Constructive Alignment for Introductory Programming

Andrew Cain

A thesis presented for the degree of Doctor of Philosophy

2013

Abstract

This thesis discusses the application of constructive alignment with portfolio assessment to the teaching of introductory programming. The goal of the work aimed to create a positive, student-centred, teaching and learning environment that encouraged, and rewarded, students to focus on deep approaches to learning.

Learning to program has been found to be very challenging. Work in this thesis investigated ways to improve student learning outcomes in introductory programming units taught at the university level. It discusses improvements in the teaching and learning environment that resulted from applying the principles of constructive alignment, including the application of constructive learning theories, aligned curriculum, and open assessment practices.

The thesis presents a systematic literature review of existing applications of constructive alignment, and argues for the need to explore approaches that aim to capture the integrated nature of the original work. It argues that this can only be achieved through adjustments to delivery and assessment practices, and proposes a set of guiding principles that can be used to create the desired learning environment.

A model of constructive alignment is presented, which encompasses the proposed principles and provides processes and guidelines for its implementation. The practicality of the resulting approach is demonstrated through the description of two programming unit exemplars.

Within this context, a concept-based, procedures-first, approach to introductory programming is also proposed, along with a range of supporting tools and resources. This approach provides students with a solid understanding of programming concepts, and experiences with a range of programming languages and paradigms, by the end of their first year of university study.

Analysis of the exemplar units, and the resulting student learning outcomes, demonstrates the positive potential for learning environments created using the proposed approach. The resulting learning environment supports a wide range of student capabilities, rewards students for deeply engaging with unit material, and encourages them to use their imagination and creativity. Using this approach, teaching staff are consistently astounded by the quality of work students are able to achieve.

While the exemplars applied the model to introductory programming, the discussion illustrates how the model can be applied to a wider range of subject areas.

Dedicated to all my students

Acknowledgements

In many ways, this work started when I was still very young and over the years the ideas that now come together have been shaped by a number of influential people, each of whom I would like to acknowledge for their formative role in this work.

I was fortunate to learn programming sitting at the kitchen table with my father. He introduced me to computing, and taught me the importance of programming concepts. Without these understandings, I would not be in the position I am today. Similarly, I would like to thank my mother for her efforts to support the scouting movement that the whole family eventually became engaged with. This helped reinforce the positive attitude of always doing your best, an attitude I now live by and have embedded within the approach presented in this thesis.

My current employer, Swinburne University of Technology, also need a special thanks for allowing me to try these radically different approaches to teaching introductory programming, but more so for introducing me to colleagues who would enable this to succeed. To Dr. Rajesh Vasa and Dr. Clinton Woodward, with whom I have shared an office at various stages, I would like to say thank you for the many discussions on teaching, and for working with me to apply these concepts to your teaching, and promote it to wider audiences. I am also indebted to the tutors who have helped me with the delivery of the programming units, I feel very lucky to have worked with such great teams over the years. To Shannon Pace for all his hard work helping prepare the various research papers we have worked on together, and to Allan Jones, Rohan Liston, and Joost Funke Kupper for the work on teh Doubtfire tool.

I would like to acknowledge with particular gratitude the assistance of my supervisors, Prof. John Grundy, and Dr. Clinton Woodward. Your support and feedback have been critical in the formation of this thesis, and I hope to continue working with you on this into the future.

Finally, I would like to thank my wife Alison for her loving support during the long period it has taken me to conduct the research and write up this thesis.

Andrew Cain, 2013

Declaration

I declare that this thesis contains no material that has been accepted for the award of any other degree or diploma and to the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of this thesis.

Andrew Cain, 2013

Publications Arising from this Thesis

The work described in this thesis has been published as described in the following list:

1. Cain & Woodward (2012), Toward constructive alignment with portfolio assessment for introductory programming, in *Proceedings of the first IEEE International Conference on Teaching, Assessment and Learning for Engineering*, IEEE, pp. 345–350.
2. Cain (2013*a*), Developing assessment criteria for portfolio assessed introductory programming, in *Proceedings of the 2nd IEEE International Conference on Teaching, Assessment and Learning for Engineering*, IEEE, pp. 55-60.
3. Cain & Woodward (2013), Examining student reflections from a constructively aligned introductory programming unit, in *Proceedings of the 15th Australasian Computer Education Conference*, Vol. 136, pp. 127–136.
4. Cain et al. (2013), Examining student progress in portfolio assessed introductory programming, in *Proceedings of the 2nd IEEE International Conference on Teaching, Assessment and Learning for Engineering*, IEEE, pp. 67-72.
5. Woodward et al. (2013), Helping students track learning progress using burn down charts, in *Proceedings of the 2nd IEEE International Conference on Teaching, Assessment and Learning for Engineering*, IEEE, pp. 104-109.

Contents

List of Figures

List of Tables

1

Introduction

The work in this thesis aimed to improve student learning outcomes in introductory programming units through the application of constructive alignment with portfolio assessment. Through changes to the assessment approach, and delivery strategy the work has demonstrated the creation of a positive, student-centred, teaching and learning environment that encouraged, and rewarded, students for adopting deep approaches to learning. The foundations of this environment are captured in twelve guiding principles, which are embodied in the approach that was developed over nine iterations of a practical action research project. Results and analysis from this project are presented and discussed in this thesis.

Programming is a critical skill in Computer Science and Software Engineering. As a consequence students in these fields are taught programming from the start of their degree programmes at many Universities. Although programming has been taught for a number of decades, learning programming remains challenging (Jenkins 2002, Lahtinen et al. 2005, Lister et al. 2004, McCracken et al. 2001, Ragonis & Ben-Ari 2007, Renumol et al. 2010, Robins et al. 2003, Rountree et al. 2002, Wiedenbeck 2005), with a general consensus that many students find programming hard. These challenges, along with the current lack of success in this critical area, have been recognised by McGettrick et al. (2005) as one of their seven grand challenges in computing education. Despite persistent efforts over many years, we are still a long way from having specific guidance on how best to approach teaching introductory programming.

Research into teaching introductory programming is an active part of the computing education research field. In their survey of literature on introductory programming, Pears et al. (2007) identified four main categories: curricula, pedagogy, language choice, and tool support. Work on curricula has examined how introductory programming fits into the wider university computing curricula, including recommendations for computing curricula by major professional computing societies (ACM/IEEE-CS Joint Task Force 2012). Computing education research on pedagogy has examined the teaching and learning of introductory programming and included topics such as approaches to adapt learning theory for computer science (Ben-Ari 2001), various views of the central concepts of programming (Denning 1989, Dijkstra 1989, Hoare 1969, Palumbo 1990, Robins et al. 2003), to work on appropriate cognitive structures (Eckerdal et al. 2005, Green & Petre 1996, Green 2000, Soloway 1986). Language and, by association, paradigm choice has also been widely studied, with various papers on which programming language to use in teaching introductory programming (Anik & Baykoç 2011, Böszörményi 1998, Bishop & Freeman 2006, Brilliant & Wiseman 1996, Howell 2003, Kelleher & Pausch 2005, Koffman 1988*b*, Maloney et al. 2010, Mannila et al. 2006, Mannila & De Raadt 2006, Mody 1991, Pendergast 2006, Roberts 1993) to debates on which programming paradigm should be used early in the curricula (Astrachan et al. 2005, Bennedsen & Caspersen 2004, Cooper et al. 2003, Ehlert & Schulte 2009, Howe et al. 2004, Lister et al. 2006, Pattis 1993, Reges 2006). Also, research regarding tool support has examined the use of software tools specifically designed to support the needs of novice programmers, including work on automated assessment (Ala-Mutka 2007, Douce et al. 2005), visualisation (Naps et al. 2002), and programming environments (Gross & Powers 2005, Kelleher & Pausch 2005, Kölling et al. 2003).

Advancements from general education literature have also provided additional advice on underlying theories and practices. This includes works on the scholarship of teaching and learning (Boyer 1990), approaches to teaching (Martin et al. 2000), approaches to learning (Marton & Säljö 1976*b*, Entwistle 1991, Trigwell & Prosser 1991, Trigwell et al. 1999, Marton & Säljö 2005), and analysis of the learner's experience (Marton & Booth 1997). While seen as beneficial, many of these general education theories need further research, and a greater inclusion in the computing education research discourse in order to determine their effectiveness in relation to teaching introductory programming.

In their study of students' experiences learning to program, Bruce et al. (2003) categorised the way students engage with learning to program into five categories, ranging from approaches focusing on "*getting through the unit*," to those aimed at discovering what it means to be a programmer. Each of these categories can be broadly classified as either a surface or deep approach to learning (Marton & Säljö 1976*b*, Ramsden 1992) to program. When engaging surface approaches, students attempt to address the outcomes with as little effort as possible. This leads to situations in which students are primarily motivated by fear of failing, and experience learning as a struggle, with the topic appearing tedious, hard and boring – adjectives that are too often associated with learning to program (McGettrick et al. 2005). Alternatively, when engaging deep approaches to learning students seek meaning in what they do, and relate their learning to the bigger picture. To succeed at learning to program, students need to engage deep approaches to learning, as surface approaches alone are unlikely to be sufficient (Bruce et al. 2003). Pedagogy that encourage students to adopt deep approaches to learning should, therefore, help address some of the issues related to this challenging topic.

Biggs' model of constructive alignment (Biggs 1996, Biggs & Tang 2007), based upon constructive learning theory (constructivism) and aligned curriculum, aims to enhance student learning outcomes by focusing on *what the student does*. Constructive alignment aims to encourage students to use *deep*, rather than *surface*, approaches to learning. The focus on the central role of the learner in building meaning is derived from constructivist learning theories (Piaget 1950, Phillips 1995, Steffe & Gale 1995, Jonassen 1991*b*, Vrasidas 2000), whilst the alignment of assessment, teaching, and learning activities, has its foundation in instructional design literature (Tyler 1969, Cohen 1987, Ramsden 1992). Biggs' model is student focused, with clear and intentional alignment of assessment, teaching and learning activities, and unit objectives.

The principles of constructive alignment were discovered through the use of portfolio assessment (Biggs 1996). In a unit on psychology, students had been presented with a set of intended learning outcomes, and asked to construct a body of work that demonstrated they had met the stated outcomes by the end of the unit. This work was then collected together and submitted as a *portfolio* for assessment. The resulting environment encouraged and rewarded students for focusing on developing deep understanding of concepts related to the stated outcomes. This original work provided a clear vision, with strong compelling arguments for incorporating student-centred approaches to unit delivery and assessment.

While there is extensive literature on constructive alignment in the general education literature, the reported work has generally focused on adapting unit delivery without changing assessment approaches. Given the central role of assessment in defining curriculum (Ramsden 1992), from the students perspective, these approaches have not been able to report the same degree of success as reported by Biggs (1996).

Constructive alignment has received little attention in computing eduction research related to teaching introductory programming. The work of Thota & Whitfield (2010) and Gaspar & Langevin (2012) have discussed the principles of constructive alignment in relation to teaching introductory programming. Gaspar & Langevin (2012) used constructive alignment to suggest a range of potential changes, while Thota &

Whitfield (2010) provided a deeper discussion on adjusting delivery methods. While these approaches indicate some potential, neither appeared able to recreate all aspects of the positive student-centred learning environment reported in the original work on constructive alignment.

Given this context, research that aims to recreate the positive student-centred learning environment, as reported by Biggs (1996), could help inform both computing education, and the wider education research literature. It is the view supported by the work in this thesis that a supportive student-centred learning environment that encourages students to adopt deep approaches to learning helps address some of the challenges associated with teaching introductory programming. At the same time, recreating the student-centred learning environment reported by Biggs (1996) has helped to identify additional principles not currently promoted in associated with constructive alignment.

1.1 Research Goals

The primary focus of this research was to improve student learning outcomes in introductory programming units through the application of constructive alignment, as originally proposed by Biggs (1996). The goal was to create a supportive, student-centred, learning environment in which students are encouraged and rewarded for engaging in deep approaches to learning, and to identify principles and guidelines that can be used by others looking to create similar environments.

Biggs' original work on constructive alignment, and the use of portfolio assessment, had been limited to small class sizes. It is our observation that introductory programming units typically involve large class sizes, with potentially hundreds of students. To enable this approach to unit delivery and assessment to be used in these introductory programming classes it was necessary for this work to determine processes that enabled the approach to scale to these larger class sizes. By improving the scalability of the general approach described by Biggs, the work in this thesis broadens the possible applications for units using constructive alignment with portfolio assessment.

The work in this thesis also examined the environment resulting from the application of the proposed approach. The resulting environment differed from traditional units in higher education, in both its method of delivery and assessment. As a result, one aim of this work was to gain a better understanding of how students learn in this new environment.

Analysing student learning outcomes also provided an opportunity to develop tools and resources to support the approach, and to examine the use of these tools within the delivery of introductory programming units. These resources helped support the identified principles and approach, which in turn aided student learning.

1.2 Research Approach

As the goals of this research work focused on improving student learning outcomes, it indicated the need for a practical and applied research method. As a result, this study used a Practical Action Research (Creswell 2008) design based on Mills' (Mills 2010) *dialectic action research spiral*. The model, and identification of its underlying principles, developed over a number of iterations, with each iteration involving a number of steps. The approach involved reflective practice, with each iteration providing insights that feed into subsequent iterations.

The Practical Action Research method used involves iteratively performing the following steps: (1) identify an area of focus, (2) collect data, (3) analyse and interpret the data, and (4) develop an action plan. In practice each iteration aligned to a teaching period and collected data from students undertaking units using the proposed approach. At the conclusion of each teaching period data was analysed and an action plan developed for subsequent iterations. The focus of each iteration varied as different aspects of the model required attention.

The final model, as presented in this thesis, is the result of nine iterations of the Practical Action Research method. Each iteration helped refine the approach, with the model stabilising after significant change in early iterations.

1.3 Key Contributions

The contributions of this thesis relate to the dual fields in computing education research: education, and computer science and software engineering. This thesis makes the following contributions to the field of education:

- A systematic literature review of applications of constructive alignment.
- A set of guiding principles for the development and delivery of units that aim recreate the "web of consistency" evident in the early work on constructive alignment.

- An approach to constructive alignment developed from the guiding principles with strong links to constructivism in both teaching and learning activities, and assessment.
- An online task tracking tool to help students track their progress on tasks designed to provide students with feedback.
- Evaluation of the resulting teaching and learning context, and tools, from an educational perspective.

This thesis makes the following contributions to computer science and software engineering:

- An introductory programming curriculum designed using the principles of constructive alignment.
- An approach to teaching introductory programming, that embodies the identified principles, with guidelines for implementing this approach.
- Example implementations of the approach presented, demonstrating its application to teaching a number of introductory programming units.
- A concept-based approach to introductory programming, together with supporting resources including a concept-based text, a game development framework, and a range of video podcasts.

As a whole, these contributions support Biggs (1996) model of constructive alignment, and demonstrate a learning context that is aptly captured in the following quote (Biggs & Tang 2007) (p54):

> All components in the system address the same agenda and support each other. The students are 'entrapped' in this web of consistency, optimizing the likelihood that they will engage the appropriate learning activities . . . but leaving them free to construct their knowledge their way.

1.4 Thesis Structure

This thesis first considers existing applications of constructive alignment, and then goes on to develop a model of constructive alignment using portfolio assessment. Following this a curriculum for introductory programming is proposed, evaluated, and discussed in detail.

Chapter 2 - Approaches to Constructive Alignment provides an extensive literature review of applications of constructive alignment. The main finding of this work indicates that applications of constructive alignment reported in the research literature tend to focus on staff aligning activities and assessment to intended learning outcomes. Constructive learning theories, when addressed, related to the design of teaching and learning activities, but not to assessment approach.

Chapter 3 - Guiding Principles outlines twelve principles that underlie this work; nine relate to *how* the teaching and learning environment should operate, and the remaining three relate to *what* is to be taught. These principles guided decision making throughout this research, and underlie the approach and example units presented.

Chapter 4 - A Model for Constructive Alignment of Introductory Programming presents an approach to constructive alignment with strong links to constructivism in both teaching and learning activities, and assessment. While the approach presented was developed for the teaching of introductory programming, general methods for its adoption are discussed. This work helps address the gap identified in the literature review.

Chapter 5 - Applying Constructive Alignment and Portfolio Assessment for Introductory Programming proposes an introductory programming curriculum designed using the principles of constructive alignment, with strong emphasis on both constructivism and aligned curriculum. This curriculum revives the "procedures-first" approach to introductory programming, and moves the focus from syntax to underlying concepts and abstractions. Through the use of graphical programming grammars, the curriculum incrementally introduces students to programming concepts and then associated syntax. This approach is based on constructive learning theories and aims to help students build viable models of the underlying machine and programming abstractions.

Chapter 6 - Supporting the Curriculum with Tools and Technologies describes a range of resources used to support the approach in teaching introductory programming: an online task tracking tool, game development framework, programming text, and video podcast series. The task tracking tool provided students with details of their progress as they worked throughout the delivery of the unit. The game development framework presented was designed primarily as a teaching tool to support the procedures-first concept-based approach to teaching introductory programming used in the example units. This is further supported by a programming textbook, and a series of video podcasts.

Chapter 7 - Evaluation of the Teaching and Learning Context describes the iterations of the action research method, and provides an evaluation of the teaching and learning context created through the implementation of the approach presented. This work examines the resulting learning environment from an education as well as a computer science and software engineering perspective. This chapter also presents the results of thematic analysis of students portfolios, with one analysis examining issues students raised and the other reporting on progress students made throughout the teaching period.

Chapter 8 - Discussion elaborates on the implications of the findings, and how these can apply to wider contexts. It also discusses the importance of each of the guiding principles, and the aspects of the approach in crafting a supportive teaching and learning environment. The overall outcomes experienced are discussed along with challenges for wider adoption.

Chapter 9 - Conclusion and Future Work provides a summary of the thesis, and argues that the findings presented can aid in the design and delivery of teaching and learning in higher education. This chapter closes with a discussion of potential future work aimed at extending the approach and curriculum presented.

The Appendix contains data gathered as part of the background literature review in Chapter A1, an illustrative list of the concepts presented in the programming text in Chapter A2, and documentation indicating Ethics approval has been granted for this work in Chapter A3.

2

Approaches to Constructive Alignment

This chapter reviews the existing literature related to constructive alignment, its theoretical foundations, applications in higher education, and relationship to published work in computing education research. The chapter aims to provide a general background on existing approaches to constructive alignment. Additional background literature is discussed in the following chapters in relation to the specific work presented.

Section 2.1 provides background on constructive alignment, and relevant associated work on constructivism and aligned curriculum. This is followed in Section 2.2 with a systematic literature review of applications of constructive alignment reported in peer reviewed research publications, which outlines how others have approached the application of constructive alignment and their findings. Section 2.3 then follows with a discussion of the challenges of teaching introductory programming, associated work from the computing education literature, and specifically looks at how constructive alignment has been applied for introductory programming. The chapter closes with Section 2.4 by briefly commenting on the opportunities for research that are addressed in the remainder of this thesis.

2.1 Constructive Alignment

Constructive alignment, as proposed by Biggs (1996), is an amalgamation of constructive learning theory and aligned instruction design. It aims to elicit deep learning approaches from all students. Biggs' model is student focused, with clear and intentional alignment of assessment, teaching and learning activities, and unit objectives. The central role of the learner in building meaning is derived from constructivist learning theories, whilst the alignment of assessment, teaching, and learning activities, has its foundation in instructional design literature.

This section briefly describes these underlying principles and related literature, each of which is later expanded upon in Chapter 3. Section 2.1.1 describes the work on approaches to learning, and discusses how the learning environment can influence students' approaches to learning. The central focus on constructive learning theories is then discussed in Section 2.1.2, with a discussion of aligned curriculum following in Section 2.1.3. The section concludes with an overview of the model of constructive alignment, Section 2.1.4, and then a brief discussion of the example implementation of these principles that was presented by Biggs (1996).

2.1.1 Approaches to Learning

The studies by Marton & Säljö (1976*b*,*a*, 2005) on student approaches to learning examined the processes and strategies students applied to learning. The work identified two different levels of processing, described as *surface* and *deep* approaches to learning, summarised in Table 2.1. When a student adopts a deep approach to learning, they study with the aim of understanding the material. They engage meaningfully with the task at hand, and use high cognitive levels in order to integrate the new knowledge into their current understanding. In contrast, when students use surface approaches to learning they engage lower cognitive levels, and aim only to be able to reproduce the material in test or exam conditions. When reading, for example, surface approaches can be characterised as focusing on the text itself, while deep approaches look for the meaning behind the text. Later work also added a third approach to learning termed a *strategic* approach by Ramsden & Entwistle (1983), or *achieving* by Biggs (1987), identified students who switch between deep and surface learning approaches in order to maximise their grade.

Table 2.1: Approaches to Learning identified by Marton & Säljö (1976*b*,*a*, 2005)

Deep	**Surface**
- Engage meaningfully	- Engage without meaning
- High cognitive level	- Low cognitive level
- Aim to integrate knowledge into self	- Aim to reproduce for assessment
- Long term understanding	- Short term memorisation

Marton and Säljö's work examined the different strategies students had used when adopting surface and deep approaches to learning, and attempted to provide activities designed to engage surface learners in similar activities to those adopting deep approaches (Marton & Säljö 2005). For example, one strategy had the student answer questions similar to those spontaneously asked by students who had engaged in deep approaches to learning. This strategy resulted in students adapting their surface learning approaches rather than in any change in the way they approached their student. Students continued to "skim the surface" of the text, but in such a way as to be able to simply mention content from various sections in a very superficial way.

An alternative strategy resulted in more positive results, as reported by (Marton & Säljö 1976*a*). In this experiment, students were asked to read three chapters from a text with the aim of answering questions after reading that required either a) precise factual information, or b) broader understanding in terms of major lines of reasoning. Students who expected to answer factual questions tended to adopt surface approaches to learning. The group who expected broader questions had a mix of different approaches, with some adopting deep approaches, but not all. The major influencing factor had been the students interpretation of what was expected of them, with only half of the group interpreting the expectations as intended.

A range of subsequent studies have associated deep approaches to learning with higher quality learning outcomes (van Rossum & Schenk 1984, Prosser & Millar 1989, Trigwell & Prosser 1991, Ramsden 1992, Marton & Säljö 2005, de Raadt et al. 2005). The work reported by de Raadt et al. (2005) related to the teaching of introductory programming, and used Biggs et al. (2001) two-factor study process questionnaire (R-SPQ-2F) to evaluate the affect of study approach on results for one hundred and seventy seven students across eleven tertiary education institutions across Australia and New Zealand. The work found that *approach to learning* had the strongest correlation to success as compared to other cognitive and demographic measures.

A student's perception of their learning environment has been found to influence the approach they take in their study. A range of investigations into students' perceptions of their learning environment, and its impact on the approach to learning, were reported by Ramsden (1992). Shallow approaches were adopted by students who perceived assessment as requiring memorisation and recall, and by those who perceived workload to be high. Perceptions of high quality teaching, freedom in choosing some aspects of what is to be learned, and clarity of goals and standards required were all related to deep approaches (Trigwell & Prosser 1991, Ramsden 1992, Trigwell et al. 1999). These findings provide additional support for the original work of (Marton & Säljö 1976*a*), that found that students adapt their learning approach based on what they perceive is expected of them.

Trigwell et al. (1999) suggested that, given the observed relations between perceived environment and approach to learning, improvements in learning outcomes may be achieved by creating an environment that encourages and supports deep approaches. Their work reported a number of important factors in creating this environment, including; teaching staff providing useful feedback, making objectives clear, motivating students and providing flexibility for students to determine what and how they learn.

In their study of students' experiences learning to program, Bruce et al. (2003) categorised the way students engage with learning to program into five categories, listed below. Each of these categories can be broadly classified as either a surface or deep approach to learning to program. When engaging surface approaches, students attempted to address the outcomes with as little effort as possible. This leads to situations in which students are primarily *motivated* by fear of failing, and experience the learning as a struggle, with the topic appearing tedious, hard and boring, adjectives that are too often associated with learning to program (McGettrick et al. 2005). On the other hand, when engaging deep approaches to learning students seek meaning in what they do, and relate their learning to the "bigger picture."

Surface Approaches where student attention is on the parts and not the whole:

Following - "getting through" the unit becomes the students primary goal, and their focus is on completing assessment tasks without aiming to develop any level of understanding.

Coding - rote memorisation of programming language syntax.

Deep Approaches attention is on programming as a whole:

Understanding and integrating concepts - students focus is on understanding programming concepts, and integrating these ideas to help with the development of programs.

Problem solving - programming is seen by the students as a means of solving problems, and their focus is on learning different ways to structure code to create solutions.

Participation/enculturation - students aim to become programmers and to engage with the software development community and its culture.

To succeed at learning to program, students need to engage deep approaches to learning, as surface approaches alone are unlikely to be sufficient (Bruce et al. 2003). Pedagogy that allows us to encourage students to adopt deep approaches to learning should, therefore, help address some of the issues related to this challenging topic.

2.1.2 Constructivism

Constructivism is a theory to explain the development of knowledge that focuses on the active role of the learner in constructing their own understanding. Dating back to Piaget (1950) constructivism exists in several forms: cognitive, individual, postmodern, radical and social constructivism (Phillips 1995, Steffe & Gale 1995). Each of these forms of constructivism has various implications for teaching and learning. Two review papers that discuss these implication are Jonassen (1991*b*) and Vrasidas (2000). These reviews outline the differences between constructivist and objectivist thinking in terms of education.

Approach to education is greatly influenced by educator's theory of teaching and learning, with constructivist and objectivist theories often being seen at two extreme ends of a continuum. Objectivism is characterised by the presence of external meaning, or knowledge, and education is therefore a task for the learner to come to know that which is real (Jonassen 1991*b*, Vrasidas 2000). Effective education can be realised through standardisation, and productivity improved through the mechanisms of management, as in business and industry (Tyler 1969, Vrasidas 2000). In contrast, constructivism is characterised by knowledge being internal to the learner, a human construction that is an interpretation of the external world (Jonassen 1991*b*, Vrasidas 2000). Constructive teaching theories relate to guiding learners, helping them to adopt viable models with the aim of enabling them to think and act like experts. This suggests that effective education can be realised by creating an environment that gives students control over their learning (Vrasidas 2000).

The differences between constructivist and objectivist thinking are illustrated in Figure 2.1. The illustration presents the characteristics of the two ends of the continuum, from constructivism on the left to objectivism on the right, related to epistemology, education, and assessment.

Epistemology : This section illustrates differences in how the two philosophical paradigms view knowledge

- Constructivism is shown with knowledge being a human construction that is informed by observation of, and constrained by, external reality.
- Objectivism is depicted with human knowledge being a reflection of the external reality, the true source of knowledge.

Education : Shows differences between the two philosophical paradigms in terms of what makes effective education

- Constructivism centres around guiding the learner's active construction of knowledge, with the objective of helping the learning think and act like an

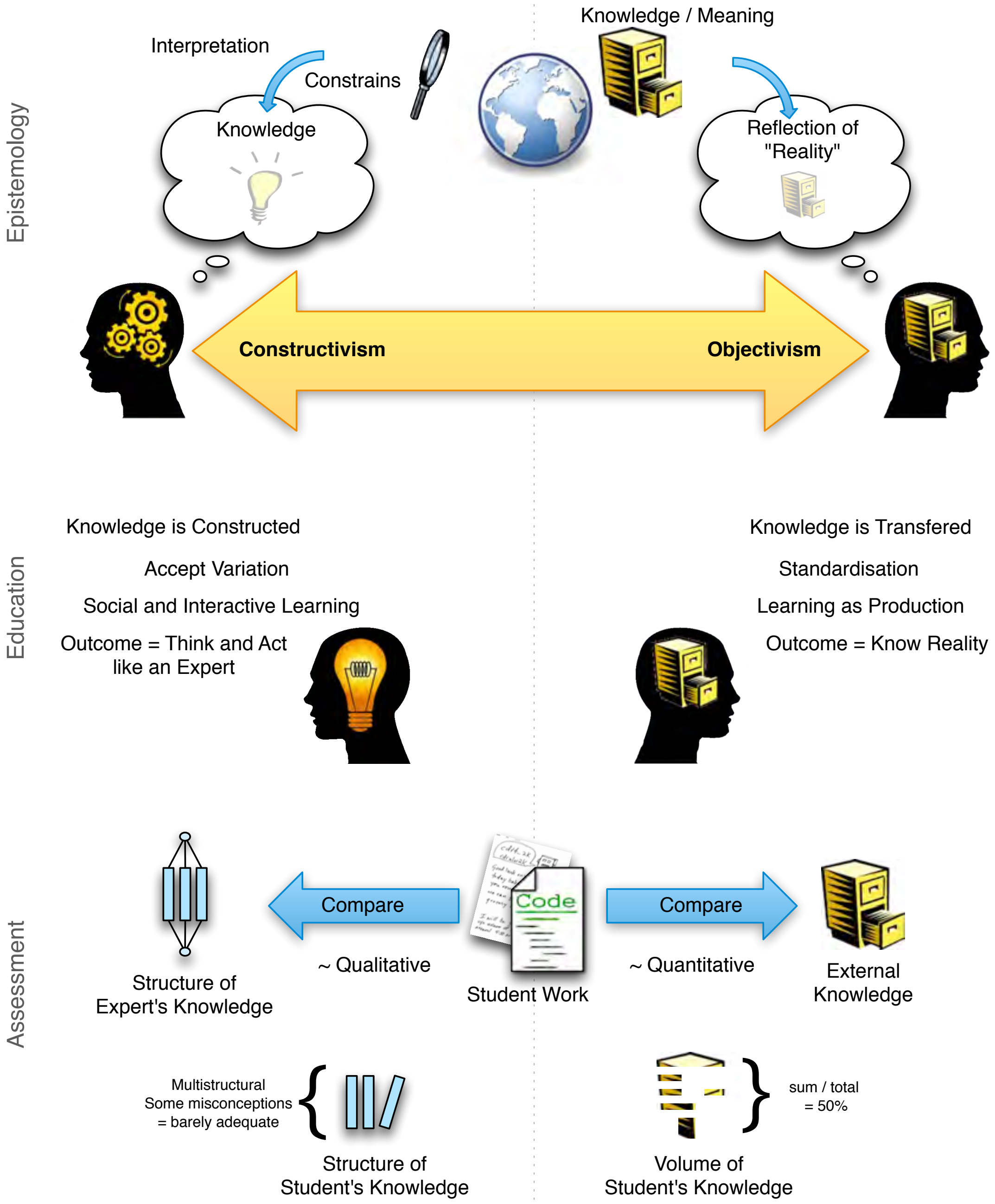


Figure 2.1: Illustration of the continuum from constructivism to objectivism, and the concepts present at levels of epistemology, education, and assessment.

expert.

- Objectivism aims to effectively transfer knowledge to the learner, with the goal of them knowing reality.

Assessment : The final section of the illustration shows the different views on how to assess student understanding.

- When guided by constructivism, assessment involves qualitatively evaluating the structure of the student's knowledge in relation to the structure of an expert's knowledge.
- Adopting objectivism, assessment involves quantitatively evaluating the amount of external knowledge the student has managed to correctly retain.

When education is guided by objectivism, instruction becomes the task of efficiently transferring knowledge to the learner. Material and tests can be standardised, using processes from business and industry to gain "productivity-like" improvements and to ensure the largest number of students are provided with access to the knowledge (Tyler 1969, Vrasidas 2000). Learning is seen as a matter of getting students to correctly conceptualise and categorise things, including the relationships between them (Lakoff 1987). It follows, therefore, that assessment is a matter of measuring the students behaviour against expectations. A focus on factual details is central to this extreme of the continuum.

With constructivism, education is seen as creating a learning environment in which students will be exposed to situations that will enable their individual construction of knowledge (Jonassen 1991*b*, Vrasidas 2000). Given that the student is constructing their own knowledge, context is important and prior experience shapes what a student learns (Jonassen 1991*a*). The goal is therefore to guide students in the construction of their knowledge, and to try to help them construct viable and meaningful conceptual structures (Jonassen 1991*b*, Vrasidas 2000). Assessment, in constructivist thinking, is divergent as each student will bring their own "reality" and learning objectives therefore become less important and possibly irrelevant (Jonassen 1992).

In relation to studies on effective teaching, constructivist learning theories – with their student-centred focus – appear to map well to features of productive learning environments. Martin et al. (2000) reported a study of twenty six university teaching staff, considering how they intended to present a topic to their students and how they subsequently delivered the topic. Their results indicated that where staff conceived of the topic as "knowledge as given", they adopted teacher-focused information transmission forms of delivery consistent with objectivist theories. Given the findings discussed in Section 2.1.1, these strategies may result in students perceiving memorisation and recall as being what is required, and therefore adopt surface approaches to learning. Where the university staff viewed the object of study as related to conceptual change, a student-focused approach was taken, which is more consistent with constructivist thinking. When such approaches were adopted, learning tended to focus on higher cognitive level activities, with a broader focus on the discipline, practice or life-long-learning and encourage students to engage in deep approaches to learning.

Early work on applying constructive learning theories to teaching computer science (Ben-Ari 1998, 2001) reported additional challenges in the application of constructivism due to the nature of computing. Beginner student in computer science have no *effective model* of a computer to start with, instead beginning their study with a limited model akin to a "giant brain." These inadequate models, and possible associated misconceptions, are likely to be exposed through interactions with the computer and possible "psychological grief" resulting as students work toward viable model. To address these issues, Ben-Ari (1998, 2001) suggested the explicit teaching of a model of the computer, as originally proposed by DuBoulay (1986).

Van Gorp & Grissom (2001) provided a range of constructivist techniques suitable for teaching introductory programming, with the view of the constructivist classroom as a problem-solving environment. The work recommended students be provided authentic, though simplified, problems to help intrinsically motivate students, and that meaningful activities be designed to enable students in the construction of their own knowledge. Suggested activities included:

- *Code walkthroughs* where students walk through existing code to predict computer behaviour.
- *Code writing* where students produce small programs, small programs or pseudocode may be provided as *scaffolding* for more complex tasks.
- Location and correction of issues, both logical and syntactic, in the form of *code debugging* activities.
- Greater interactivity was recommended for lectures, with students then being given time to *reconstruct lecture notes* as a summarising activity.

Thramboulidis (2003*a*,*b*,*c*) described a shift from teaching a second programming unit on object oriented programming from a traditional "textbook" style approach, to a design-first approach based upon constructivist learning theories. The course presented by Thramboulidis involved students modelling familiar real-world situations using object oriented programming principles and class and sequence diagrams. This utilised students understanding of "objects" in the real world in the production of object oriented simulations suitable for implementation in computer programs. Following these activities, students went on to design and implement an object oriented calculator, making use of the concepts they had learnt from the earlier tasks. Results reported in this work indicated that the more active engagement of the students had resulted in positive improvements in both the pass rate and programs the students created.

Constructivism has also been applied to teaching computer graphics, with mixed results. Taxén (2004) reported on a case study where they had applied constructive learning theories to teaching 3D graphics. The strategy applied in this work related to *discovery learning* (Duffy & Cunningham 1996) where teaching staff refrain from making explicit instruction, and instead provided a series of activities to help student discover the relevant knowledge themselves. The discussion and conclusions of Taxén (2004) indicated that students had felt the learning environment was too challenging, and they did not have sufficient background knowledge. Taxen's reflections indicated the use of an exam had contributed to a disconnect between the teaching approach and the assessment approach; the exam, in effect, encouraged students to adopt surface approaches that had been actively discouraged in the learning.

Wulf (2005) reported a constructivist-based pedagogy for teaching introductory programming and information technology that was considered to have created a more accessible environment for a wider range of students. This work recommended the active learning tasks outlined by Van Gorp & Grissom (2001), and expanded upon this with a studio-based approach to teaching introductory programming with collaborative group-based instruction to create a more student-centred learning environment for computer science units.

These various works on constructivism in computer science education indicate that the application of constructivist learning theories should be beneficial for teaching introductory programming. By focusing on the active role of the student in constructing their own knowledge it should be possible to create a unit in which students develop an appropriate model of the computer, gain an understanding of important programming concepts, and acquire associated programming skills.

2.1.3 Aligned Curriculum

Tyler (1969) defined curriculum alignment as occurring when material learnt in earlier years is built upon and supported in subsequent classes. Tyler's aligned model consisted of four major steps (1) identifying objectives, (2) choosing appropriate learning experiences, (3) organising these experiences so as to maximise learning, and (4) evaluating the resulting learning. These four steps were always aligned, with objectives matching experiences and assessment tasks. The work of (Ramsden 1992) indicates that this alignment can be achieved as long as assessment matches objectives; from the students perspective the curriculum is always defined by the assessment.

Benefits of aligned curriculum can be related to the work on approaches to learning. One of the identified requirements for positive learning environments, discussed in Section 2.1.1, was the clear communication of expectation to students. In an aligned curriculum the teaching and learning activities match the assessment tasks, and help provide the clarity of intention required to encourage students to engage in deep approaches to learning.

The impact of aligning teaching and learning activities and assessment tasks was highlighted by Cohen (1987). In discussing instructional alignment Cohen (1987) reported students achieved significantly better results when the teaching was aligned to the assessment than with non-aligned instruction. This is in line with expectations if an aligned curriculum is likely to help engage students appropriately with learning tasks.

2.1.4 The Model of Constructive Alignment

Constructive alignment brings together aspects of constructive learning theories together with aligned curriculum, with the overall aim of encouraging students to adopt deep approaches to learning.

In his original paper on constructive alignment Biggs (1996) adopted constructivism as a framework to help guide decision making in all facets of teaching and learning. Constructivism was chosen due to its central focus on the students construction of their own knowledge, and the student-centred learning environments that result from accepting this epistemology. Biggs (1996), clarified again in Biggs & Tang (2007), makes it clear that the emphasis is on student activity being central to knowledge construction, and that education should focus more on *conceptual change* than on acquisition of information. This pragmatic view places Biggs work somewhere on the constructivist side of the constructivism-objectivism continuum, but not necessarily at its extreme.

Aligned curriculum forms the second pillar of constructive alignment. Biggs (1996) proposed the alignment of teaching and learning activities and assessment tasks to a unit's intended learning outcomes. Intended learning outcomes capture unit[1] goals in terms of cognitive activities students will be able to perform by the end of the unit.

[1] A unit in this context refers to a unit of study, also called a subject or course in other institutions.

Biggs' work claimed that this use of aligned curriculum – with constructivist learning theories – results in a student-centred learning environment and a clear focus and consistent message.

Houghton (2004) described an overall model of constructive alignment, illustrated in Figure 2.2, as consisting of the following blocks:

- *Intended learning outcomes* clearly define required learning in terms of "performances of understanding."
- *Performance objectives* emerge from the desired outcomes, and can be ranked to become the assessment criteria.
- *Teaching and learning activities* are designed to place students in situations likely to elicit the required learning.
- Students provide *evidence of their learning*, that are assessed against the criteria to determine grade outcomes.

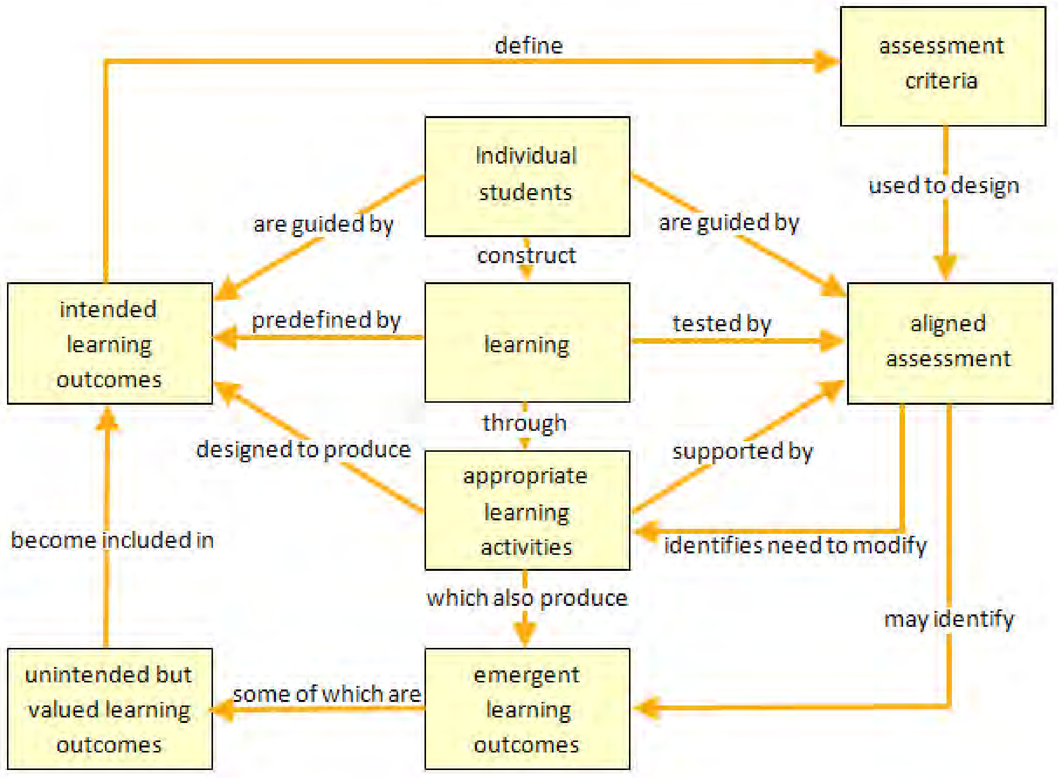


Figure 2.2: Constructive alignment model presented by Houghton (2004)

Intended learning outcomes are central to constructive alignment, and therefore their construction must ensure suitable cognitive levels are required in order to encourage deep approaches to learning. Biggs (1996) proposed the use of the SOLO taxonomy

(Biggs & Collis 1982) for this purpose. SOLO stands for *Structure of the Observed Learning Outcome* and defines five levels of outcomes: pre-structural, uni-structural, multi-structural, relational and extended abstract. These levels are illustrated in Figure 2.3 and described in the following list. Each of the five levels of outcomes can be associated with a list of verbs likely to elicit that level of cognitive activity, and these verbs can then be used in defining a unit's intended learning outcomes.

Prestructural indicates no understanding of the topic; essentially, the student has *missed the point*.

Unistructural indicates understanding of one relevant aspect of the topic; the student starts addressing the topic but does little more than *get on track*.

Multistructural indicates understanding several relevant aspects, but each is understood in isolation. The student can provide a suitable list of facts, but does not structure their response to address the topic as a whole. To adopt cliché, the student "sees the trees but not the forest."

Relational understanding indicates that the topic is seen as a whole, with the various aspects integrated into a relevant structure. The student can respond appropriately to questions, demonstrating an integrated understanding of the topic.

Extended Abstract goes beyond the present, generalising the structure into new domains or in new ways. The student should be able to demonstrate a "breakthrough" response providing new insights on the topic.

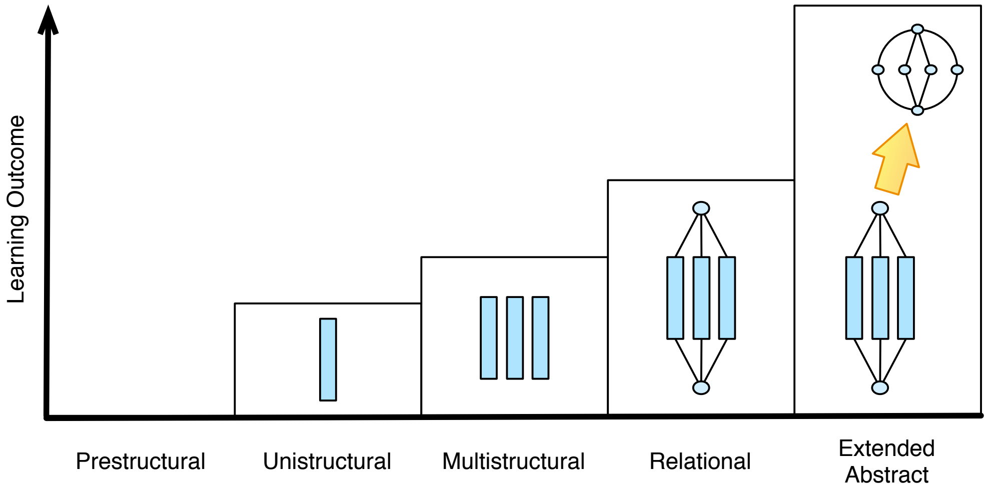


Figure 2.3: The five levels of the SOLO taxonomy, used to help define intended learning outcomes and assessment criteria, adapted from Biggs & Tang (2007).

Constructive alignment aims to encourage students to engage in deep approaches to learning by clearly stating unit intended learning outcomes, aligning teaching and learning activities and assessment tasks with these outcomes, in a student-centred environment. In this way, constructive alignment helps communicate to students the required approach to learning, with a consistent message throughout delivery and assessment. The resulting student-centred environment weaves a "web of consistency," optimising the likelihood of students *engaging appropriately* with learning activities (Biggs 1999).

2.1.5 Biggs' Example Implementation

Interestingly, the principles of constructive alignment were not conceived and then implemented, but rather *discovered* through reflection. Biggs & Tang (2007) (p.51) describes the discovery of these principles through the application of a portfolio for assessment of a third year psychology unit in the Bachelor of Education, the compelling example used in Biggs (1996) and further elaborated upon in Biggs & Tang (1997).

Biggs & Tang (1997) described a portfolio as a body of work selected by the student to demonstrate that they had met the unit's intended learning outcomes. A portfolio consisted of a number of *items*, or *pieces* of work, as well as an overall statement indicating *why* the items had been included, and how they demonstrated that the student had met the unit's intended learning outcomes. Portfolios are discussed in further detail in Section 4.1.2.

Biggs' example of constructive alignment clearly demonstrated the application of both constructive learning theories, and aligned curriculum. By using a portfolio, the unit revolved around the intended learning outcomes, with students needing to submit a body of work that demonstrated how they had met outcomes in order to pass the unit. The portfolio was assessed holistically, with different grade outcomes relating to how well the learning outcomes had been addressed. Teaching and learning activities revolved around the preparation of portfolio pieces, including a range of student centred activities and guided instruction where appropriate.

Biggs (1996) indicated that the results from the first implementation of the example application of constructive alignment had "stunning" results; portfolios included a range of rich and exciting evidence of learning, a large number of students received high grades, and student feedback was the best the unit had received. Biggs attributed this success to the design of the unit. Intended learning outcomes had defined required performance levels, teaching and learning activities had elicited them, and assessment had confirmed them. The assessment, activities, and outcomes had been in alignment, all working together to help the students construct the required knowledge. These aspects then formed the formal pillars of the constructive alignment model proposed by Biggs.

2.2 Reported Applications of Constructive Alignment

In this section we outline a literature review that examines some applications of constructive alignment in Higher Education. This review aims to identify how constructive alignment can be applied to introductory programming, and any gaps in the current research literature.

2.2.1 Review Method

Petticrew & Roberts (2008) defined a systematic literature review as a process of systematically analysing all available studies in order to answer specific research questions. The systematic nature of reviews carried out in this manner ensure that the review is thorough and fair, providing an opportunity to synthesise existing work in a scientific manner. The review presented here followed the systematic literature review process of Kitchenham (2007), and aimed to identify, to the best of our knowledge, all available studies on how Constructive Alignment has been applied in Higher Education.

Figure 2.4 shows the three phases in the systematic literature review carried out in this work. The first phase identified appropriate search and filter criteria used in locating associated literature. In Phase 2, the search criteria was used to identify potentially relevant articles from the indicated sources. The articles identified were then filtered, using the filter criteria, to identify the relevant articles to pass on to Phase 3, where relevant data was collected from the articles, and the results analysed.

Phase 1: Search and Filter Criteria

This review focused on the application of constructive alignment, and the effectiveness of the teaching and learning environment created. Petticrew & Roberts (2008) suggested that the formulation of research questions for a systematic review should consider five aspects: Population, Intervention, Comparison, Outcome and Context (PICOC). Addressing these five aspects enabled the creation of effective search and filter criteria.

Table 2.2 lists the five PICOC aspects related to this review. The Population consists of the specific target group that the study examines. In this study the population included students and academics in the context of Higher Education. This work aimed

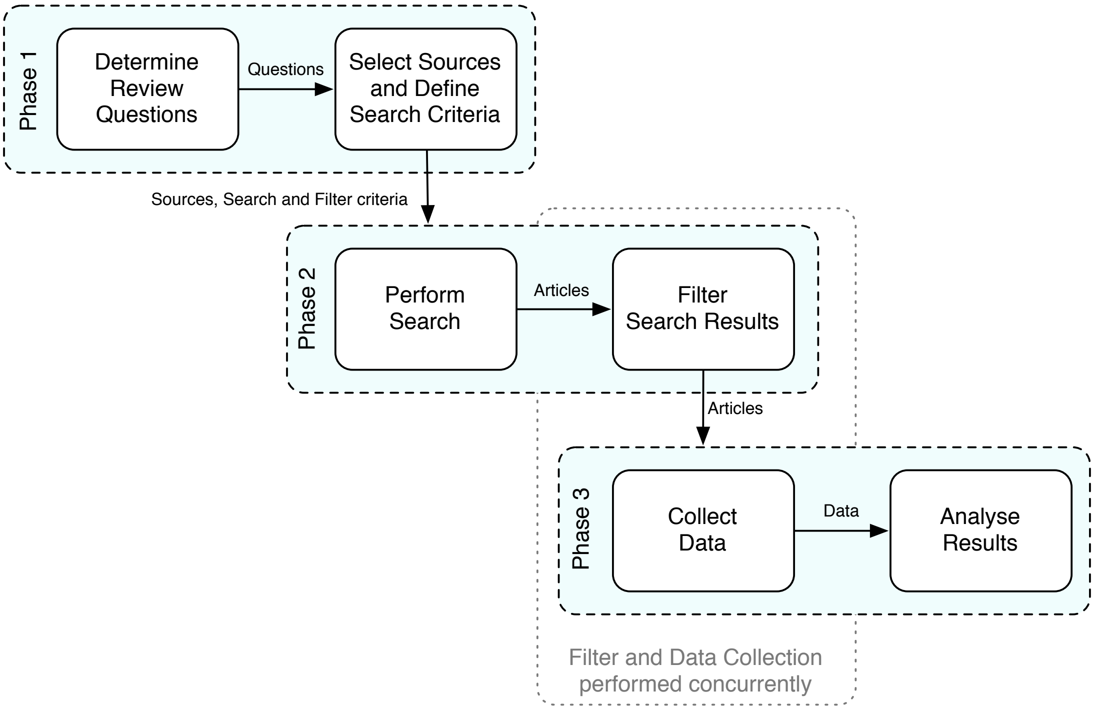

Phase 1
Determine Review Questions
Questions
Select Sources and Define Search Criteria
Sources, Search and Filter criteria
Phase 2
Perform Search
Articles
Filter Search Results
Articles
Phase 3
Collect Data
Data
Analyse Results
Filter and Data Collection performed concurrently

Figure 2.4: Systematic Literature Review processes carried out in this work, based on steps of Kitchenham (2007).

to review interventions where academics had applied the principles of constructive alignment, and any comparisons they had with existing approaches to teaching and learning. It aimed to investigate the range of approaches to delivery, and kinds of assessment used. In terms of outcomes, we were interested in examining any positive or negative impacts these changes had on either staff or students.

This systematic literature review aimed to answer the following questions:

1. What evidence is there of studies on the application of constructive alignment to teaching and learning in higher education?
2. How has the effectiveness of constructive alignment been measured in these studies, and how effective has constructive alignment been in the higher education setting?
3. What teaching and learning activities were used in conjuncture with constructive alignment?
4. What forms of assessment have been used with constructive alignment?
5. In what ways have applications of constructive alignment addressed the two main elements of constructive alignment: constructivism and aligned curriculum?

Petticrew & Roberts (2008) described the need for the search criteria to result in high

Table 2.2: Focus "aspects" for database search using PICOC

Aspect	Value
Population	Students and Academics in Higher Education
Intervention	Applications of Constructive Alignment in the design of teaching and learning activities and assessment.
Comparison	Existing approaches
Outcomes	Positive and negative impacts on student learning, as well as impacts on teaching staff.
Context	Application of Constructive Alignment to the design and delivery of teaching and learning material in a Higher Education setting.

number of relevant articles, while excluding irrelevant ones. This is referred to as the search criteria *sensitivity* and *specificity*. A search with high sensitivity returns a high number of relevant articles, with high specificity a low proportion of irrelevant articles. While an ideal search criteria would be both highly sensitive and highly specific, in practice there generally tends to be a trade-off between the two.

Kitchenham (2007) provides a number of recommendations on how to define appropriate search criteria, including the use of Boolean AND and OR conditions as well as searching for synonyms. Using this approach results in a search with higher specificity, and can result in a low number of articles being identified, see Salleh et al. (2011) for example. Therefore, for this work it was decided to start with a highly *sensitive* search criteria, and search for articles that match the term "Constructive Alignment." If this resulted in a unacceptably large number of results then more specific criteria could have been added.

Given the highly sensitive search criteria, a high number of irrelevant articles was anticipated. To address this, Phase 1 also defined a number of filter conditions. The aim of these conditions was to clearly define why papers would be excluded from the analysis in Phase 3. The following lists the criteria used.

- The full text of the paper must be available to the researchers, and must be published in English.
- The paper must appear in a peer reviewed conference, journal or workshop.
- Constructive Alignment must be discussed either as the main focus, or an important aspect, of the paper.
- Results must relate to the use of Constructive Alignment in a teaching and learning context.

Phase 2: Identification of Relevant Literature

In Phase 2, the search and filter criteria from Phase 1 were used to carry out the search on the selected online databases. The resulting articles were collected in an academic reference management system that allowed articles to be categorised using a status and a number of tags. Categories were created to indicate why a paper had been excluded, and to mark each paper's progress through the data collection.

Database searches were performed within the reference management system, which provided facilities to automate the collection of the citation data and the associated full text. Where the full text was not available from the database, a general search was performed using a number of search engines in order to ensure the full text was included for as many articles as possible. This import process also identified any references that were already included in the system, thereby avoiding the creation of duplicates in the resulting library.

The use of the categorisation tools in the academic reference management system allowed the Filter stage of Phase 2 and the Data Collection stage of Phase 3 to run concurrently. Once imported, each article was placed in a **To Be Categorised** status, thereby providing a backlog of the articles that were to be examined. Each of these articles was then examined using the filter criteria, and moved to a separate status as the filter process progressed. The stages of this process are shown in Figure 2.5.

In the first stage of the filter process the availability of full text, the language it was written in, and its refereed status was checked. If these were not met the paper was allocated to the **Ignore** status, and did not proceed to the next stage of the filter.

At the next stage a full text search was conducted on each paper. This search looked for the presence of any text related to constructivism or alignment through the search strings "construct" and "align." The associated text was then read, and the paper was allocated to **Done (no mention of Constructive Alignment)** when there was no presence of either words, to **Done (insufficient details on Constructive Alignment)** when it was mentioned briefly with no details or in depth discussion, and to **To Be Read** in all other cases.

Stage 3 involved reading the abstracts, and all sections related to Constructive Alignment from the papers in the To Be Read status. A new status was allocated to each paper: **Done (not a Teaching and Learning Context)** if they were a discussion of education theory rather than a study of a teaching and learning context, **Done (not an application of Constructive Alignment)** if they mentioned Constructive Alignment

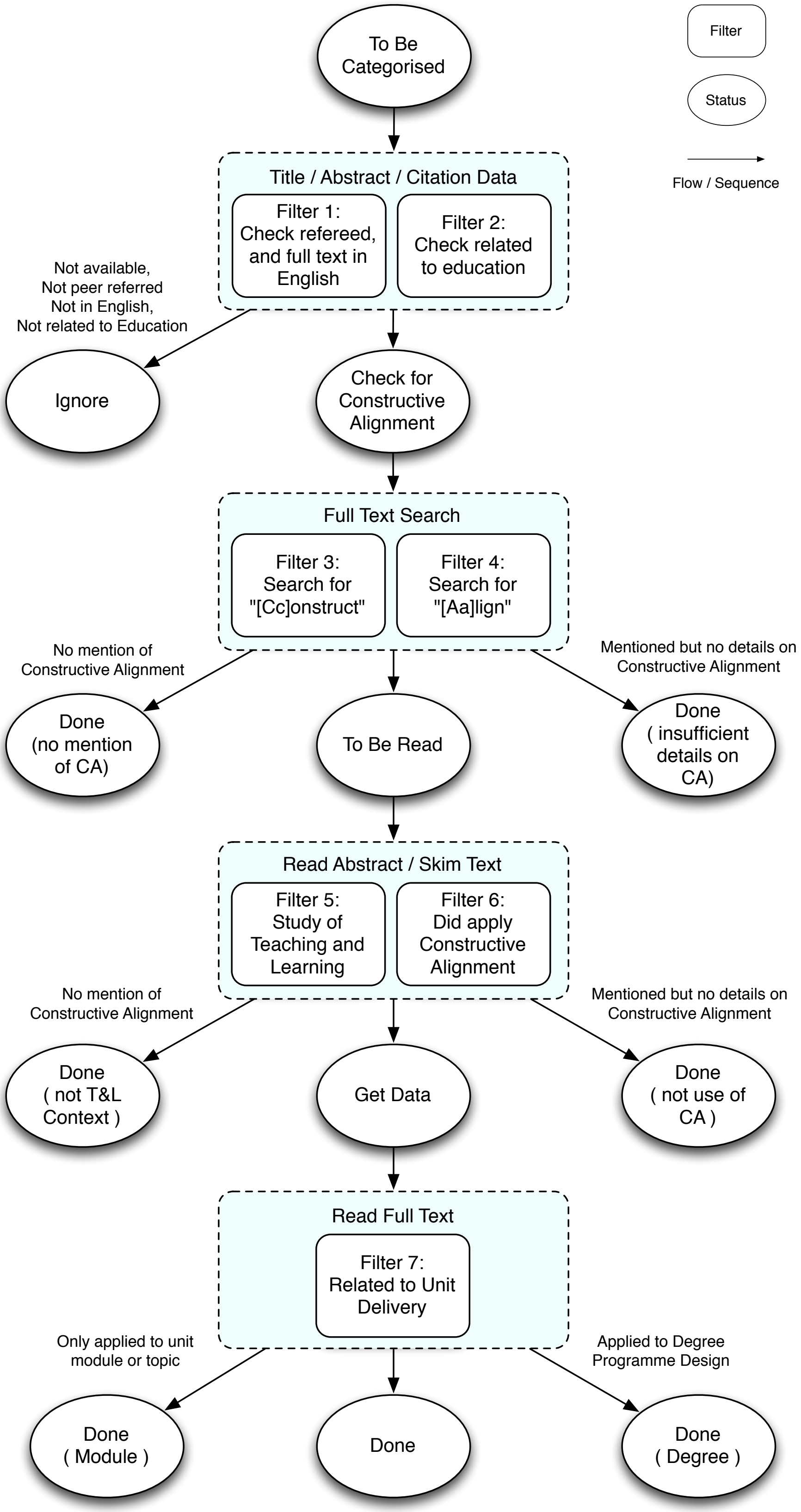

Filter
Status
Flow / Sequence
To Be Categorised
Title / Abstract / Citation Data
Filter 1: Check refereed, and full text in English
Filter 2: Check related to education
Not available, Not peer referred Not in English, Not related to Education
Ignore
Check for Constructive Alignment
Full Text Search
Filter 3: Search for "[Cc]onstruct"
Filter 4: Search for "[Aa]lign"
No mention of Constructive Alignment
Mentioned but no details on Constructive Alignment
Done (no mention of CA)
To Be Read
Done (insufficient details on CA)
Read Abstract / Skim Text
Filter 5: Study of Teaching and Learning
Filter 6: Did apply Constructive Alignment
No mention of Constructive Alignment
Mentioned but no details on Constructive Alignment
Done (not T&L Context)
Get Data
Done (not use of CA)
Read Full Text
Filter 7: Related to Unit Delivery
Only applied to unit module or topic
Applied to Degree Programme Design
Done (Module)
Done
Done (Degree)

Figure 2.5: Filter process applied to papers

but had not adopted it for the reported work, and **Get Data** in all other cases.

The final stage involved reading the full text of the papers that had been marked with the Get Data status, and classifying them based on what Constructive Alignment had been applied to. Data was collected from all of the papers at this stage, but only those related to the delivery of a unit were forwarded to the analysis phase.

Phase 3: Data Collection and Analysis

Data collection was performed by reading the indicated papers and looking for the details in the following list. Findings were recorded in a spreadsheet for further analysis, with relevant quotes from the text stored alongside the summarised information.

- Level of Unit: Undergraduate or Postgraduate, and year level.
- The Intended Learning Outcomes and associated levels from the SOLO taxonomy.
- Teaching and Learning activities used.
- Approach to assessment.
- Reported effects of constructive alignment.

In keeping with the holistic nature of the assessment approaches suggested in (Biggs & Tang 1997), the quality measure for each paper was generated by response to the following question using a five point Likert scale (Likert 1932): 5 being strongly agree, 4 agree, 3 neutral, 2 disagree, and 1 strongly disagree.

> "The paper clearly communicates how constructive alignment was applied to the unit, and the results obtained. "

Once all of the data had been collected the details were summarised in a spreadsheet, and analysed to answer the review questions.

2.2.2 Results

The search involved the use of seventeen online databases, all of which were searched for the term "Constructive Alignment." Table 2.3 lists the number of results from each of the selected databases. Google Scholar (`http://scholar.google.com`) and CiteSeer (`http://citeseerx.ist.psu.edu`) resulted in an overly large number of articles for the selected search term. In both cases the results appears to have a large number of irrelevant matches, and so the search terms were made more specific. The search in Google Scholar was updated to search for articles that included "Constructive Alignment" in the title, while CiteSeer was limited to searching abstracts for the associated text.

Table 2.3: Data sources and the number of articles located for the search term "Constructive Alignment"

Source	URL	Count
A+ Education	`http://search.informit.com.au/search`	44
Academic Search Complete	`http://ebscohost.com/academic/academic-search-complete`	32
ACM Digital Library	`http://dl.acm.org`	21
CiteSeer[2]	`http://citeseerx.ist.psu.edu`	20
EdResearch Online	`http://opac.acer.edu.au:8080/edresearch`	6
Educational Research Abstracts	`http://www.tandfonline.com`	26
Education Research Complete	`http://ebscohost.com/academic/education-research-complete`	48
eJournals	`http://ejournals.ebsco.com`	42
ERIC	`http://www.eric.ed.gov`	31
Google Scholar[3]	`http://scholar.google.com`	104
IEEE Xplore	`http://ieeexplore.ieee.org`	16
Libra	`http://academic.research.microsoft.com`	87
PsycINFO	`http://ebscohost.com/academic/psycinfo`	16
Scopus	`http://www.scopus.com/`	79
Springer Link	`http://link.springer.com`	125
VOCED	`http://www.voced.edu.au`	28
Web of Knowledge	`http://wokinfo.com`	52
	Total Unique	335

A total of 335 unique articles were identified, and were filtered using the process indicated in Section 2.2.1, resulting in 38 papers being included in the final analysis, the full details of which can be found in Appendix A1. Table 2.4 and Figure 2.6 show the number of papers excluded by each filter. Filtering for lack of full text and peer reviewed status was relatively straightforward, as was identifying papers that did not mention constructive alignment, or were unrelated to a teaching and learning context. Similarly, papers related to the use of constructive alignment of a module or degree programme were also easy to identify. Filtering for depth of discussion on Constructive Alignment was the one filter that required the most attention. In many of these

[2] Within CiteSeer the search was performed on article abstracts containing the text "Constructive Alignment."

[3] The search in Google scholar returned more than three thousand results, this was then limited to articles that included "Constructive Alignment" in their title.

cases the filter was easily able to exclude papers which lacked any real discussion of constructive alignment beyond stating it should be used, and to include papers which discussed the topic in depth. However a small number of papers had *some* discussion of constructive alignment but did not necessarily provide any depth. In these cases, the papers were initially included for analysis, and if during the data collection there was insufficient discussion to enable the data extraction then the paper was excluded and attributed to this filter.

Overview of Papers Collected

Table 2.5 and Figure 2.7 show the number of papers by field of study. This list uses the Fields of Study defined by Trewin (2000), and each of the units described was allocated to one of the associated fields. A total of sixteen (42%) of the papers were associated with the fields of Information Technology or Management and Commerce, with each field being associated with the units from eight papers. No papers were associated with Creative Arts, Food, Hospitality, and Personal Services, or Mixed Field Programmes.

In terms of level of education, the analysed papers included units at both undergraduate and postgraduate levels. Table 2.6 and Figure 2.8 show the number of papers reporting units at the undergraduate and postgraduate levels. Where reported, the data collected includes the year level for undergraduate units. Two of the reported studies involved combined undergraduate and postgraduate units, and a further two units did not include any indication of the associated unit's level.

Units in the analysed papers were primarily delivered face to face (82%), with three being delivered online (8%), and four units having a combination of online and face-to-face delivery. These results are included in Table 2.7.

Table 2.8 shows the geographic location of unit delivery in the reported work. The location of the authors university was used in the cases where the geographic location was not explicitly mentioned in the paper's text. It is interesting to note that although papers were collected from international sources, 47% of the papers analysed related to units delivered in Australiasia, with 39% from Australian universities and 8% from universities in New Zealand.

The increasing popularity of constructive alignment, at least in terms of papers reporting its application, can be seen by examining publication years. Figure 2.9 provides a visual representation of this data, also shown in Table 2.9, and indicates an increas-

Table 2.4: Counts of papers excluded by the indicated filters

Exclusion Reason Starting total	**Count** 335
No access to full text, or not peer reviewed	77
No Mention of Constructive Alignment	33
No depth of discussion on Constructive Alignment	96
Not a study of a Teaching & Learning Context	43
Not an application of Constructive Alignment	23
Earlier versions of papers, removed as a duplicate	10
Applied Constructive Alignment to Module within Unit	8
Applied Constructive Alignment to Degree Programme Design	7
Final "included" papers	38

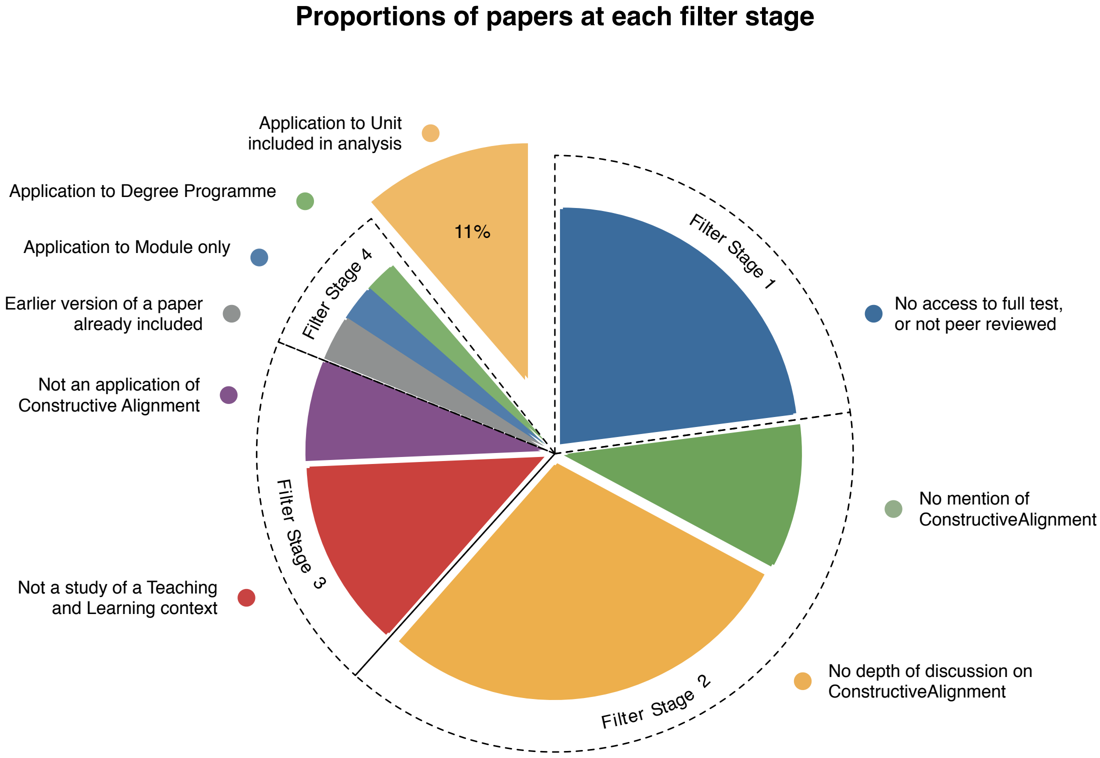


Figure 2.6: Pie chart showing the proportion of initial 335 papers in each status based on the stage excluded by the filters

Table 2.5: The number of papers, from the 38 papers included in the analysis, in each field of study

Field	Count
Information Technology	8
Management and Commerce	8
Society and Culture	6
Health	4
Agriculture, Environmental and Related Studies	3
Education	3
Engineering and Related Technologies	3
Natural and Physical Sciences	2
Architecture and Building	1
Creative Arts	0
Food, Hospitality and Personal Services	0
Mixed Field Programmes	0

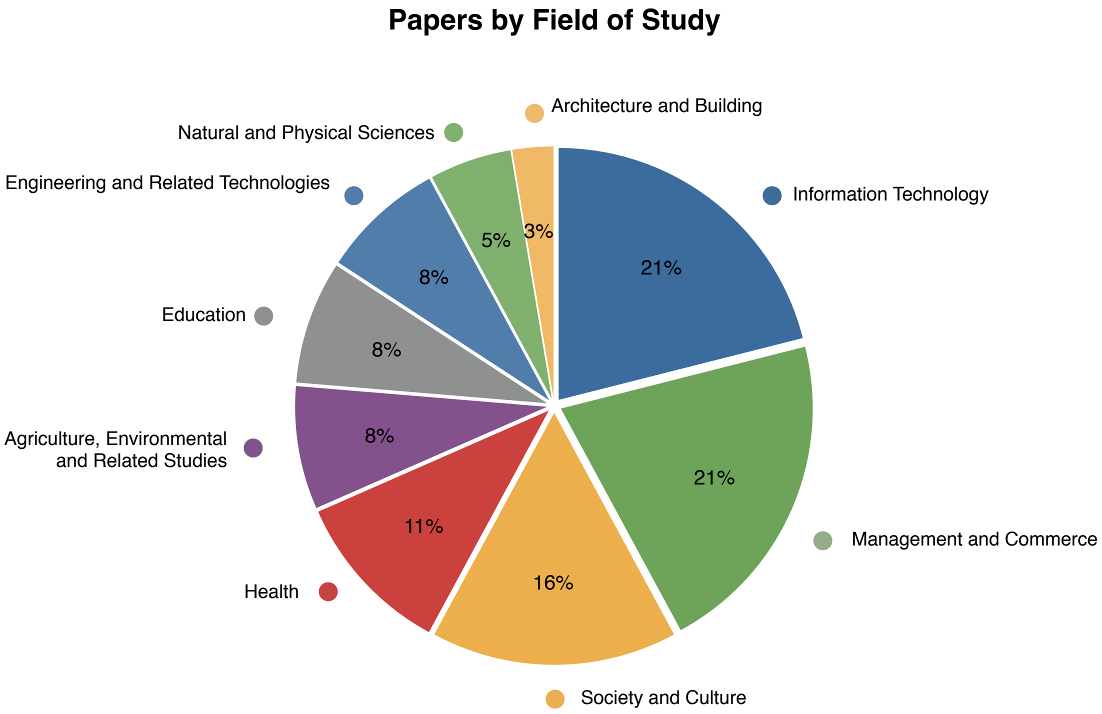


Figure 2.7: Pie chart showing the distribution of papers by field

Table 2.6: Numbers of papers reporting on units at undergraduate and postgraduate levels, and the indicated year level for undergraduate units.

Level and Year	Count
Undergraduate (Total)	33
- First Year	7
- Second Year	5
- Third or Later Years	12
- Year level not stated	9
Postgraduate	5
Level not stated	2

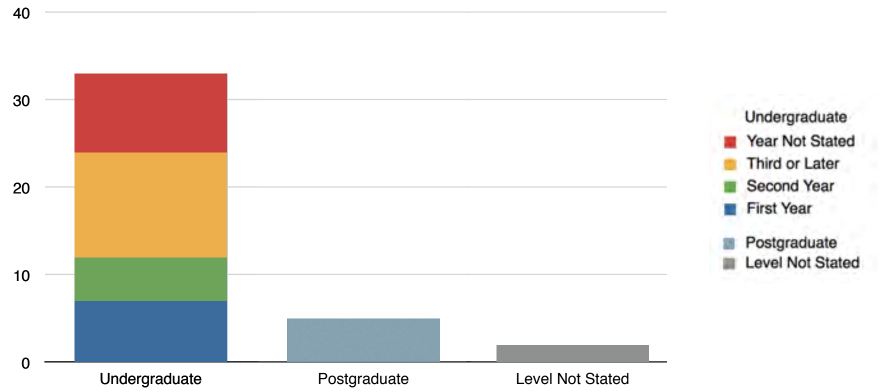


Figure 2.8: Bar chart showing the level, and year, of the units reported.

Table 2.7: Method of unit delivery

Delivery	Count
Face to Face	31
Face to Face & Online	4
Online	3

Table 2.8: Count of papers by geographic location where the unit was delivered.

Geographic Location	Count
Australia	15
United Kingdom	7
Europe	5
New Zealand	3
Hong Kong	3
China	2
United States of America	1
Canada	1
South Africa	1

ing number of papers published from 2005. It should be noted that the search was performed on the 18th of July 2012, and therefore figures for 2012 only capture papers indexed by the source databases prior to this time. The data in Figure 2.9 and Table 2.9 also includes the counts of work that was excluded as duplicate studies, where the one study had been presented in a number of papers.

Teaching and Learning Activities

In the papers that included face-to-face delivery, the data collection examined the types of classes used. The results of this are included in Table 2.10, which lists the number, and percentage, of the papers that reported using lectures, various kinds of tutorials, and other teaching and learning activities. Table 2.10 also records the number of papers in which there was no clear communication of the teaching and learning activities were used.

Lectures and tutorials remain the primary form of class used in the papers studied, with 74% of the papers reporting either or both of these class types. Four papers included the use of *Other* class types, the details of which appear in the following list. It is interesting to note that five papers (15%) did not provide details on the teaching and learning activities used.

- Szili & Sobels (2011) reported using a four day field camp to visit the scene of a serious environmental crisis, as part of a unit related to environmental management.
- Donnison & Edwards (2011) included twenty hours of community service as part of a unit in the first year of an degree in Education.
- Eight weeks of clinical practice Tang et al. (1999), as part of a nursing degree.
- Shoufan & Huss (2010) included an excursion of a fabrication plant as part of an electronics and digital systems design unit.

When collecting data on the class types used, additional information was sought as to how constructive learning theories had been incorporated into the teaching and learning activities. Three different strategies were identified: applying concepts through the use of **problem-based learning** or the examination of case studies, making the classes more **interactive**, and **group work** and discussions.

Problem-based learning, projects, and case studies, all provide a means for educators to create a setting in which students can develop knowledge through application of associated principles. The constructivist nature of these activities has been reported

Table 2.9: Year of publication for the papers analysed. Note that the count for 2012 is partial as the search was conducted on the 18th of July 2012. Numbers in parenthesis indicate the number of papers removed in the filtering process as a duplicate of later papers included.

Year	Count	Year	Count
1999	1	2006	2
2000	1	2007	3 (3)
2001	0	2008	4 (1)
2002	1	2009	6
2003	1	2010	5 (2)
2004	1	2011	4 (2)
2005	3	*2012*	*6*

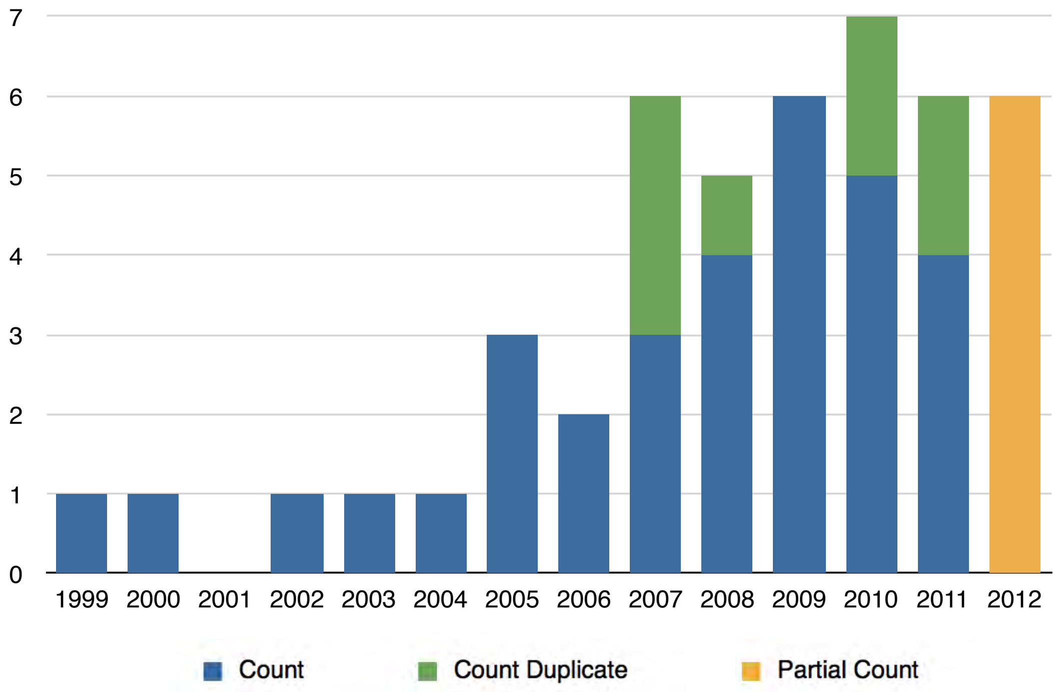


Figure 2.9: Bar chart showing the number of papers published by year, including earlier versions of work included in the analysis, and the partial count of the papers published in 2012.

Table 2.10: Class types used by units that included face to face delivery

Class Type	Count	Percent
Lecture	26	76%
Tutorial, Class, Workshop or Session	25	74%
Other	4	12%
Not reported	5	15%

Table 2.11: Method of incorporating constructive learning theories into the teaching and learning activities

Method	Count	Percent
Applying Principles: Problem Based Learning, Projects or Case Studies	18	53%
Moving beyond "knowledge transfer": Interactive Lectures and Classes	11	32%
Social Constructivism: Group Work and Discussions	9	26%
Not reported	12	35%

by a range of authors. See for example, Hendry et al. (1999), Savery & Duffy (1995) and Schmidt et al. (2009). In the articles analysed in this research, examples of this approach include the use of Problem-based Learning by Warren (2005) in teaching software design. Programming projects played an important part of the work reported by Brabrand (2008), as these provided an opportunity for students to apply their understanding of concurrency in a practical sense.

Davey & Bond (2002) reported that in their unit on pharmacokinetics, lectures were changed to examine case studies that guided student decision making in relation to drug therapy and to explore scientific rationale.

Interactive lectures and classes indicate a shift away from the objectivist ideas of knowledge transfer, and toward constructivist ideals. Shepherd (2005) provides one example of this where lectures shifted to "lectures as workshops", with case studies being discussed and the lecturer solving problems from first principles.

Palincsar (1998) indicated that from a social constructivist perspective, higher-order thinking can be enhanced through the use of social interaction, such as can be achieved through classroom discussion. A number of papers indicated a shift to group work and classroom discussions as a means of engaging with these constructivist theories. One example of this shift is reported by Israel et al. (2007), who reported using tutorials as a means of guiding group research projects in their advanced undergraduate unit on psychology.

Table 2.11 shows the number of papers that reported the use different methods for incorporating constructive learning theories in the teaching and learning activities. Again, it is interesting to note that twelve papers (35%) did not discuss the use of any method for addressing constructivism in their teaching and learning activities, which makes it difficult to understand how these papers adopted constructive learning theories, a key component of constructive alignment.

The remaining four online units used the following teaching and learning activities.

- Hoddinott (2000) students were provided with reading, optional reading, and a range of online resources.

- Raeburn et al. (2009) included work-based learning, and the use of online reflective journals.
- Terrell et al. (2011) provided online tutorials.
- Brown et al. (2006) used online "Thought Conferences" where questions were designed to promote interaction through involving case studies and problem-based inquiry.

Approach to Student Assessment

Reported approach to student assessment included a range of activities, as shown in Table 2.12. Each of the different assessment approaches is detailed in the following list.

- **Examination**: Indicates the use of a final exam in the work.
- **Assignments**: Indicates the use of classwork assignments, projects, essays, and other work conducted by students individually during the teaching period.
- **Group Projects and Assignments**: Indicates the use of group marks in the student grades, and included group assignments, presentations and projects.
- **Reflective Journals**: Indicates the use of journals to reflect on the learning process.
- **Tests during delivery**: Indicates the use of tests commonly referred to as "mid-term" tests or quizzes.
- **Portfolios**: Indicates the use of a range of assessment approaches, discussed below, involving the collection of a number of pieces of work that are submitted as a group.
- **Participation**: Indicates that a part of the students grade was awarded for participation in teaching and learning activities.

Table 2.12: Forms of assessment reported by the papers analysed

Form of Assessment Used	Count	Percent
No assessment specified	5	13%
Assessment Specified	33	
- Examination	23	70%
- Assignments	22	67%
- Group Projects and Assignments	14	42%
- Reflective Journals	8	24%
- Tests during delivery	4	12%
- Portfolios	4	12%
- Participation	2	6%

Four papers indicated the use of portfolios in the student assessment but, unlike other forms of assessment, the exact nature of the portfolios differed between the various papers. It is interesting to note that the alignment between the portfolio and the unit's intended learning outcomes was not clearly presented in any of these four papers.

- Tang et al. (1999) reported the use of a portfolio consisting of a learning diary, case study of a patient from clinical practice, and a piece of the students own choosing. This formed a part of the overall assessment along with an exam.
- Raeburn et al. (2009) indicated the use of two portfolios, each of which included a resumé, a number of blog entries, and a written report of experience in work based learning. In this context a portfolio is simply a means of collecting together separate tasks.
- Portfolios reported by Scott & Fortune (2009) required students to complete twelve tasks over the course of the year, and combine these for submission as their portfolio. The assessment of the portfolio looked for "positive contributions", but little detail was provided as to how this assessment was performed.
- So called "ePortfolios" were used in conjuncture with other assignments in the work reported by Donnison & Edwards (2011). The ePortfolios included a multimedia digital story on the communities the students were involved with, and to document the student's community service experience.

Aligning Curriculum and Assessment

All of the papers included some detail on curriculum alignment. In most cases the papers presented generic details of the importance of achieving alignment between teaching and learning activities and assessment tasks, with few actually providing any details on how this alignment was performed or specified. As indicated in Table 2.13, 61% of the papers analysed provided little, or no, explicit details on how the alignment was achieved. In many of these papers the teaching and learning activities were discussed in some detail, but the specifics of how these were intended to develop the student's understanding of the intended learning outcomes was not discussed.

Where an explanation of the alignment, the teaching and learning activities and assessment tasks were often presented in a matrix, or graphic, that showed the intended learning outcomes and the associated teaching and learning activities and assessment tasks. Table 2.14 shows an example of how one of the intended learning outcomes was presented in Terrell et al. (2011). Other papers included textual descriptions of how the alignment was performed. For example Brabrand (2008) provided an in-depth discussion of how they planned the alignment for their unit on concurrency. The numbers of papers that explained alignment using a matrix, other graphical means, or textually is shown in Table 2.13 and graphically in Figure 2.10.

Table 2.13: How alignment of teaching and learning activities, and assessment, was achieved as reported in the papers analysed.

Aspect related to Alignment	Count	Percent
Alignment performed entirely by staff	35	92%
Little, or no, explicit details about how alignment was performed	23	61%
Explained alignment using matrix, graphic	9	24%
Explained using examples or textual explanations	6	16%

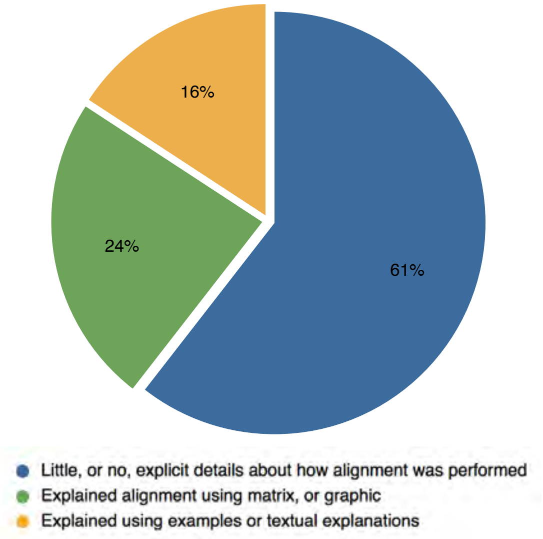


Figure 2.10: Method used to explain the alignment of the curriculum

Table 2.14: Example of alignment matrix from Terrell et al. (2011)

Outcome	Activities	Assessment Criteria
...	...	...
Using communication and team working skills to promote productive and cohesive relations among employees.	Read and comment on each other's blogs. Use Twitter to explore its use as a communication medium. Share resources using social bookmarking tools...	Communication, Execution and Reflection
...	...	...

Thirty-five of the thirty-eight papers indicated that alignment was performed by staff. The remaining three papers included the use of portfolios where students were required to include evidence that aligned to the unit's intended learning outcomes. The relevant details from these three papers are described in the following list.

- Hoddinott (2000) described a unit on Biology where students were required to align their portfolio submissions with the intended learning outcomes and assessment criteria. Unfortunately insufficient details were provided to enable deeper understanding of how this was subsequently assessed.
- Portfolios were also used in the unit described by Scott & Fortune (2009), although no explicit details were provided about how alignment was achieved.
- Tang et al. (1999) provided a third example of portfolio assessment, but in this case the portfolio's objectives were not directly linked to unit intended learning outcomes.

Reported Methods of Evaluation

The papers analysed indicated the use of one, or more, of a number of different data sources as a basis for their evaluation of the teaching and learning environment created. Table 2.15 lists the number of papers that included a single and multiple evaluation sources, and the number that did not include any evaluation.

Table 2.15: Number of papers using a single, multiple, and no evaluation source.

Number of Sources	Count
Single Evaluation Source	12
Multiple Evaluation Sources	22
No Evaluation	4

Evaluations sources identified in the papers analysed are listed in Table 2.16, with each of the sources being described in the following list.

- **Student Feedback** typically included feedback received as part of official unit evaluation questionnaires, but also included comments related to assessment tasks, and other informal sources of student feedback.
- **Results** indicated the use of unit grades, or results from individual assessment tasks.
- **Student Work** included document analysis of student assessment work, including assignments, examinations and portfolios.
- **Interviews and Focus Groups** additional feedback was received in response to focus groups or interviews in eight of the papers analysed.

- **Reflections on Teaching** staff reflections on teaching were included as a data source in four of the analysed papers.
- **Use of Online Tools** usage data from online learning management systems was used as a proxy for student engagement.

Reported Effects of Constructive Alignment

Table 2.17 lists the number of papers that indicated some positive results, or issues, in relation to the unit delivered. Of the thirty eight papers analysed, four did not include any evaluation of the delivery of the unit as noted in the previous section. Of the thirty four papers that included some results, 97% indicated positive aspects and 21% indicated issues.

In relation to positive results, there were three common themes: positive learning outcomes, student satisfaction and student engagement. The number of papers reporting each of these is shown in Table 2.18. Papers that reported other positive results are also included, and outlined in the following list.

- Hill (2009) reported benefits related to the ongoing refinement of the intended learning outcomes, and learning environment in general. The work also indicated that staff found the formative feedback was easier to deliver and stimulated wider discussion in class.
- Morton (2008) identified positive aspects related to improvements in written communication and understanding of the rigour required in laboratory work.
- Improvements in student confidence was reported by Scott & Fortune (2009), along with their analysis of portfolios showing students had very deep and meaningful learning experiences.
- Schaefer & Panchal (2009) reported that the use of real-world problems, as compared to textbook learning, had helped better prepare students for careers in mechanical engineering.
- Vanfretti & Milano (2012) indicated the use of free and open source software had

Table 2.16: Evaluation sources used in the papers analysed

Source Kind	Count
Student Feedback	26
Results	13
Student Work	10
Interviews of Focus Groups	8
Reflection on Teaching	4
Use of Online Tools	3

helped to engage students in deep approaches to learning.

- Vogel et al. (2007) indicated some positive effects for students using constructively aligned learning support over those who had chosen not to use these options.

Positive learning outcomes were reported in the form of higher pass rates, higher percentage of students passing on the first attempt, or as demonstrations of high quality student learning outcomes. Szili & Sobels (2011), for example, indicated that student learning journals had exceeded staff expectations, with student work showing capacity to evaluate information, learning to reason and construct arguments, engagement with and learning from the wider world, and other positive learning outcomes. Other examples include the work of Warren (2005), where students who sat the exam demonstrated the ability to apply material covered to solve new problems.

High student satisfaction was also reported in a number of papers, and typically refer to some form of unit evaluation survey. Shepherd (2005), for example, indicated that students regarded their learning experience in the associated unit as uniquely, or unusually, satisfying. Their work also indicated that students did not enjoy rote memorisation but often feel it is required or the only possible approach in some units. Similarly, Scott & Fortune (2009) reported very high satisfaction rates, with overall feedback being very positive. Students identified the interactive lecture approach and online resources as hugely beneficial.

Student engagement was also reported as a positive result in a number of papers. Davey & Bond (2002) reported a significant shift with students being more interested in their learning experiences and how the unit encouraged these. The work noted that students' comments shifted away from concerns of assessment requirements, to interest in what they were learning. Survey results indicated student interest in the subject matter, enjoyment of learning, and challenge and motivation to learn were significantly different after the reported initiative. Another good example of student engagement was reported by Szili & Sobels (2011), whose work showed students had a strong motivation for active participation in study and learning. Szili and Sobels cited their most important insight as the way that constructive alignment had worked

Table 2.17: Positive results and issues related to the teaching and learning environment created

Results	**Count**	**Percent**
No Evaluation	4	11% of papers analysed
Results related to delivery	34	89% of papers analysed
- Some Positive Results	33	97% of papers with results
- Some Issues	7	21% of papers with results

seamlessly to deliver students who were motivated to pursue lifelong learning and engagement with their communities.

Table 2.18: Papers that reported different kinds of positive results, and percentage compared to total number of papers reporting some positive results.

Positive Result Related To	**Count**	**Percent**
Learning outcomes (grades, or otherwise)	19	58%
Student satisfaction	17	52%
Student engagement	9	27%
Other positive aspects	6	18%

Problems arising from the application of constructive alignment were reported by only a few papers, with seven papers reporting issues related to the teaching and learning environment created. One common issue was additional staff and student workload as a result of the initiatives, as shown in Table 2.19. Staff workload related to the development of new material, and the provision of feedback. Student workload issues were reported via the student satisfaction surveys. The remaining two issues, classified as other in the table, are detailed in the following list.

- Norton (2004) indicated that the less capable students may have found the problem-based learning approach used too complex and challenging.
- Tang et al. (1999) reported students not enjoying the portfolio process, and not seeing it as contributing to their learning in the unit. This was attributed to either lack of clear instruction and support throughout the process, or to students prior expectations of assessment.

Table 2.19: Papers that reported different kinds of issues, with percentage compared to total number of papers reporting at least one issue.

Negative Result, or Issues, Related To	**Count**	**Percent**
Staff Workload	2	29%
Student Workload	5	71%
Other	2	29%

2.2.3 Discussion

Papers Identified

The use of a highly selective search criteria identified a wide range of papers, that were subsequently filtered to identify the papers analysed in this work. While every care was taken in filtering these papers, it is possible that a small number of papers were inappropriately excluded. To help mitigate this risk, the papers excluded at each stage were examined at least twice before filtering proceeded to the next stage. At the end of the process we have reasonable confidence in the papers included in this analysis.

It is concerning to note the number of papers that were excluded as a result of insufficient discussion on constructive alignment. Many of these papers included some details of constructive alignment in the introduction and background, and again in the conclusions, but provided no detailed discussion in the body of the work. For example, Penaluna et al. (2009) indicated opportunities for constructive alignment in the implications of their research on assessment strategies for entrepreneurship education, but provided no details on either constructivism or alignment in the remainder of the paper.

One possible explanation for this in the wider field is the growing expectation that work in education should include the principles of constructive alignment. This message is, for example, clearly communicated in the work of Haigh (2013) on successfully writing for the Journal of Geography in Higher Education. Given the large number of papers in this category it would be interesting to examine them in more detail to see how they do relate to constructive alignment.

The papers identified in this work provided a reasonable sample across a number of fields of study, but is predominantly related to undergraduate education. A greater balance between undergraduate and postgraduate had been expected, so it would be interesting to find out why constructive alignment has not been discussed more in relation to postgraduate education.

Examining the geographic location of the work resulted in a couple of interesting observations. Firstly the high number of papers from Australiasia, and the relatively small number from Northern America. The high number of papers from Australiasia may be a result of the choice of databases, with three of the databases containing work predominantly from Australia and New Zealand. However, the largest number of papers were located from databases with broad focus from international contributions, so this may indicate a greater focus on education research in this region, and a wide adoption of the principles of constructive alignment. The reverse could therefore also be posed in relation to the relatively small number of papers from Northern America. This could indicate that constructive alignment has not gained popularity in that region, or that relatively little research is published on the application of principles in Education. Both cases provide interesting questions for future research.

Effective Evaluation Sources

A clear message from the analysed papers was the value of using multiple data sources in evaluating outcomes. Student satisfaction surveys and unit results appear to be readily available sources of information, and provide valuable evidence of learning outcomes and student perceptions. This can then be further enhanced by analysis of student work and staff reflections.

One topic not often discussed in the papers analysed is the ethical implications of using student work for purposes other than assessing the students learning. Issues related to the power imbalance of the student-teaching relationship, and issues associated with the perception of coercion need to be addressed. If student work is to be used then an appropriate ethical protocol needs to be developed, and students given the opportunity to give informed consent for their participation in the research.

Opportunities for Misalignment

There appears to be significant opportunities for misalignment due to the lack of student engagement with the alignment process. Many of the papers analysed used a traditional assessment strategy involving the use of one, or more, assignments and a final exam. In this situation the teaching staff are entirely responsible for ensuring alignment between the unit's intended learning outcomes and the outcomes assessed in student submissions.

Figure 2.11 tries to capture the situation where staff perform the alignment, for the purpose of illustrating areas for potential misalignment. Staff use intended learning outcomes in the design of assessment tasks for the students to perform. These tasks may aim to elicit outcomes that require high level cognitive activities such as explaining or reflecting on the application of some principles. Staff intentions must then be communicated to students via the assessment tasks, a complex activity that can be unintentionally undermined in a variety of ways. Even if the intentions can be captured perfectly, student interpretations of these expectations are likely to weaken any requirements. As indicated in Section 2.1.1, a number of students will aim to perform assessment tasks with as little effort as possible. These students will employ surface level approaches to learning where possible, and will not engage high level cognitive activities if possible.

In addition to misalignment in the design of the assessment tasks there is also opportunities for misalignment on the assessment side of the process. Assessment tasks require assessment criteria that indicate how marks are awarded for the various aspects of the task. Where this is carried out quantitatively there is significant potential for misalignment, with the contents of knowledge needing to be broken down into identifiable units. As indicated by Biggs (1996) this results in situations where assessment encourages piecemeal identification of details, and provides no encouragement for combining these details into a coherent whole.

Given the widespread and dominant use of assignments and exams and the potential for misalignment, it is particularly concerning how few papers provided any in-depth discussion on how the alignment was performed. Where details of the alignment were provided, the papers indicated how staff aligned topics in the intended learning outcomes with the tasks in the assignments and questions in the exams. However, few papers provided any reflections on the approaches that students had engaged, and how well these aligned with staff intentions. It should be mentioned that two of the papers analysed did provide good details on this aspect. Brabrand (2008) provided a very detailed discussion of how tasks had been aligned to outcomes, and yet still indicated that their work was not an ideal example of alignment in recognition of these challenges. Hill (2009) provided significant details on the challenges, and iterative improvements, with aligning student activity to outcomes and staff intentions with the delivery of their unit over a six years period.

These challenges indicate that any genuine attempt to implement the principles of constructive alignment will need to be performed over a number of iterations. Each iteration should aim to improve alignment based on evidence gathered, and reflected upon, as part of unit delivery.

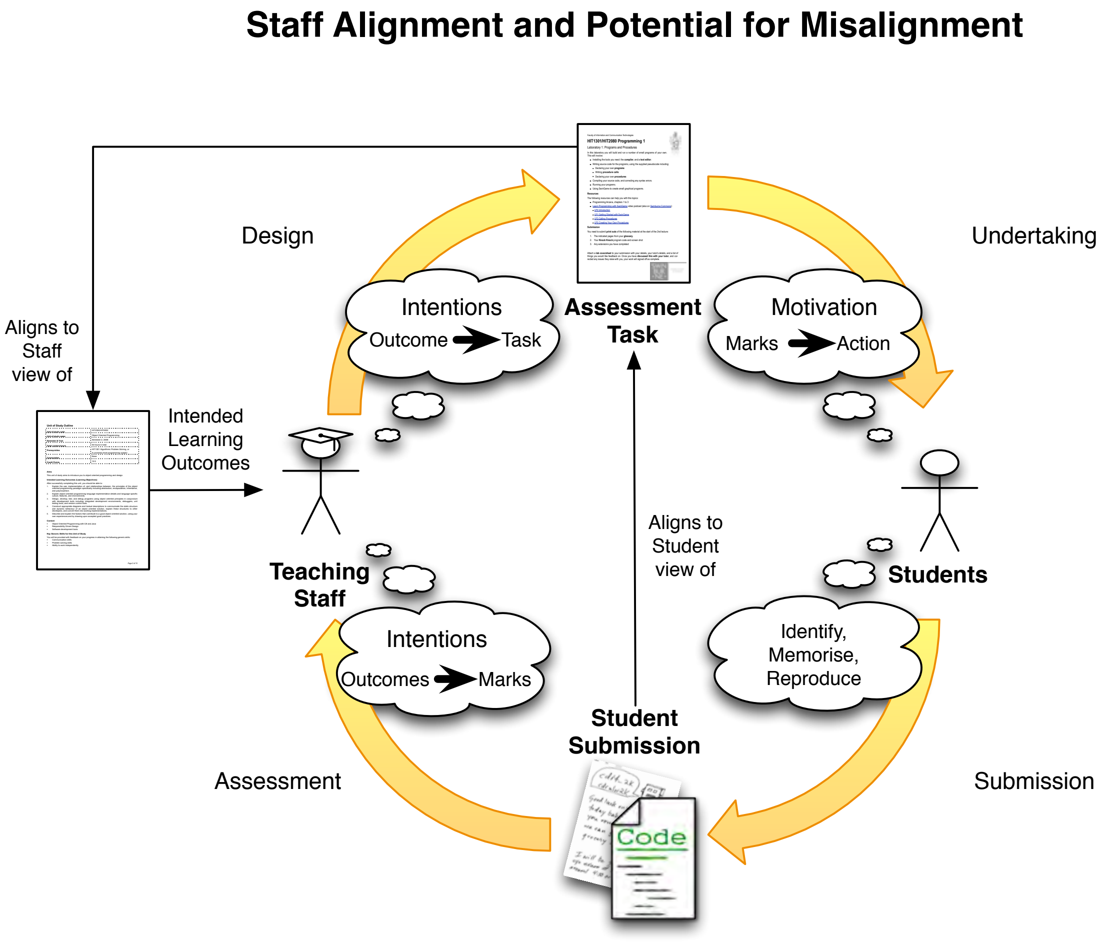


Figure 2.11: When using a traditional assignment and exam assessment strategy, particular attention is needed to ensure Staff intentions carry through to student actions.

Business as Usual

In proposing Constructive Alignment, Biggs (1996) describes a teaching and learning environment which is student-centred. Biggs argued for the use of a range of teaching and learning activities, and encouraged active student engagement. With assessment, Biggs argued against examinations, short answer or multiple-choice questions. He indicated that these approaches to assessment are likely to give credit to lower level performances than intended, and thereby encourage students to adopt lower cognitive levels than desired. Biggs (1996) strongly implied the use of portfolios, which was further elaborated upon in Biggs & Tang (1997). Portfolio assessment in this context had referred to a body of work demonstrating how the student had met the intended learning outcomes, with a justification for how the selected pieces related to the unit objectives. The principles of Constructive Alignment had been discovered by Biggs as a result of the environment created through this clear focus on the intended learning outcomes, and the "web of consistency" that had resulted.

The clarity and vision provided by Biggs (1996) does not appear to have resulted in many changes in the reported work. In many ways the papers analysed indicated a mostly "business as usual" approach. Constructive Alignment involved staff thinking about how their lectures, and the tasks they asked students to perform, related to the unit's intended learning outcomes. Some of the ideas of constructivism have made their way into teaching as either group work, and its associated group assessment, or attempts to make lectures and other classes more interactive. Similarly, approaches to assessment have remained primarily unchanged from the standard assignments and final exam. The ideas of learning through mistakes, and centring activities on students appears to have received little attention. Portfolios, when used, did not strongly follow the principles of Constructive Alignment, or had little details on how they had been implemented.

2.3 Constructive Alignment in Introductory Programming

This section examines the topic of introductory programming in computing education research. It starts with an outline of challenges associated with introductory programming in Section 2.3.1. Following this is a description of published research perspectives on teaching introductory programming in Section 2.3.2. The section concludes with a summary of two existing approaches to applying constructive alignment to introductory programming in Section 2.3.3.

2.3.1 Challenges in Introductory Programming

Introductory programming is a central skill for all students studying computer science and software engineering, and is often also taken by students in other fields. Units on introductory programming typically occur in students first year of university, and are widely recognised as being one of the most challenging units for these students. McGettrick et al. (2005) described the need to better understand programming process and programmer practice so as to deliver more effective educational outcomes as one of the seven grand challenges of computing education in recognition of the challenges related to teaching introductory programming.

Introductory programming can itself be viewed from a variety of perspectives. Dijkstra (1989) (see Denning (1989) for associated commentary) proposed a mathematical view of introductory programming, based upon formal methods and imperative programming. Palumbo (1990) discussed introductory programming from a problem solving perspective, with a focus on problem solving skills. DuBoulay (1986) likened programming to learning to control a notional machine, one that represents an ideal computer in which the programming constructs being taught are realised. However, in practice introductory programming units tend to follow a textbook style approach (Robins et al. 2003), an approach that tends to focus on language syntax and illustrative examples.

The goals of introductory programming have changed little in the last forty years. Gries (1974) indicated that introductory programming units should aim to teach students to solve problems, to describe an algorithmic solution to the problem, and verify that the algorithm is correct. These same concepts are at the core of all introductory programming units. This position was echoed by Marion (1999), who commented that the advent of object oriented programming did not fundamentally change the objectives of introductory programming. Marion included objectives related to software design, algorithm design and analysis, problem solving, and language syntax necessary for expressing these ideas in a modern programming environment. In their study on what was taught in introductory programming, Schulte & Bennedsen (2006) reported that most teachers taught such "classic" topics to the same extent, regardless of the approach they were using to teach the topic overall.

McCracken et al. (2001) reported on an investigation of programming skills of first year computer science students across four universities. The investigators reported that results of the work were disappointing, and indicated that many students did not know how to program at the end of their first unit in introductory programming. A total of 216 students completed a *Charette*, a short assessment carried out in a fixed-time laboratory session. The results included an analysis of the programs the students had completed, and indicated that in many cases students were not even able to produce code that compiled. Later work by Lister et al. (2004) found similar results when examining students ability to determine output from small code snippets, known as "code tracing tasks," and when selecting the code necessary to finish an almost complete piece of code. Given these poor results, gaining better understanding of the difficulties students face is an important avenue of research in the computing education research field.

Studies on novice programmers reported in Soloway & Spohrer (1988) indicated that novices have issues with many aspects of programming including concepts such as variables, loops, conditions, arrays, pointers and recursion, as well as having shortcomings related to planning and testing program code. The review of a number of psychological studies on programming by Winslow (1996) concluded that novices lacked appropriate "mental models" of computing, and so were limited to a surface understanding. This meant that student lacked sufficient structure to enable relevant aspects to be called upon when needed, with student approaching problems via control structures and line-by-line rather than by grouping into logical units. It is interesting to note that the work of Lahtinen et al. (2005), a survey 559 students across six universities, found that students also overestimate their understanding. This is likely to be related to their limited "mental model" and surface understanding of the subject, with many students failing to gain a relational level of understanding of the associated concepts.

In reporting on their overview of research related to the psychology of programming, Robins et al. (2003) indicate that novices face various problems, including issues related to; program design, ability to address algorithmic complexity, and the "fragility" of their knowledge, with learning to program requiring students to develop viable models of the problem domain, notional machine, and program structure. They found that students must then also develop skills necessary to map aspects from the problem domain to a the program structure, in such a way that this structure can effectively control the notional machine and generate the required outputs. The work also identified students as either "effective novices", who were able to learning to program with a reasonable effect, and "ineffective novices" who needed to expended an inordinate amount of effect to learn the required material.

Differences between effective and ineffective programming novices was reported by Renumol et al. (2010). The work used verbal protocol analysis to examined the cognitive processes used by nineteen novices. It was found that both effective and ineffective novices had used the same set of cognitive processes. A total of forty two cognitive processes were identified, and the work concluded that the use of so many cognitive processes made programming difficult, and the programming process complex to learn and practice.

A common theme across the various papers on introductory programming is that students struggle more with how and when to use programming structures than with programming language syntax. For example, Winslow (1996) indicated that students typically found it easy to generate syntactically correct statements once they understood what was required. Similar results were reported by Lahtinen et al. (2005), who indicated that the biggest problem students faced was not language details, but applying such details to solve new problems and in the creation of larger programming structures.

Approach to learning also appears to be recognised as a factor in students success in learning to program. The work of de Raadt et al. (2005) and Bruce et al. (2003), previously discussed in Section 2.1.1, indicated that approach to learning does effect results in introductory programming units, and that students need to engage in deep approaches to learning. This is further supported by the work of Jenkins (2002) on difficulties students face in learning to program, which indicated that students face a range of challenges including cognitive factors related to learning styles and motivation. Jenkins and colleagues noted that many students' approach to learning was inadequate for them to succeed in introductory programming units, with motivation being a contributing factor (Jenkins 2001, Forte & Guzdial 2005).

2.3.2 Research Perspectives on Introductory Programming

Given the challenges involved in teaching introductory programming, this is a very active area in the field of computing eduction research. In their survey of literature on introductory programming, Pears et al. (2007) identified four main categories: curricula, pedagogy, language choice, and tool support.

Work on curricula has examined how introductory programming fits into the wider university computing curriculum, including recommendations for computing curricula by major professional computing societies (ACM/IEEE-CS Joint Task Force 2001, 2008, 2012). Other work in this area has included tailoring introductory programming content to meet the needs of students not enrolled in computer science majors, such as that reported in Guzdial & Forte (2005) and Forte & Guzdial (2005).

Computing education research related to pedagogy of introductory programming examines a range of aspects related to teaching and learning. The work of Ben-Ari (2001), Denning (1989), Dijkstra (1989), Hoare (1969), Palumbo (1990) and Robins et al. (2003), discussed earlier in this chapter, all relate to pedagogy. Other work in this area includes the work of Soloway (1986) who discussed the use of plans and schemas in problem solving and program design, with additional details on the use of these in teaching programming being reported by Rist (2004). Green & Petre (1996) and Green (2000) discussed issues of cognitive load associated with various programming language elements and environments. While Eckerdal et al. (2005) described five different levels of understanding related to what it means to learning programming; from a surface level where programming is seen as a study of a programming language and the program text, to a deep level where programming is seen as involving problem solving skills that can be taken beyond the programming unit.

Language and, by association, paradigm choice is also widely studied. There exists a wide variety of papers on which programming language can and should be use in teaching introductory programming, including the work from Koffman (1988*b*), Mody (1991), Roberts (1993), Brilliant & Wiseman (1996), Böszörményi (1998), Howell (2003), Kelleher & Pausch (2005), Bishop & Freeman (2006), Mannila et al. (2006), Mannila & De Raadt (2006), Pendergast (2006), Maloney et al. (2010) and Anik & Baykoç (2011). There are also a large number of papers on the debate as to which programming paradigm should be used early in the curriculum. See for example Pattis (1993), Cooper et al. (2003), Bennedsen & Caspersen (2004), Howe et al. (2004), Astrachan et al. (2005), Lister et al. (2006), Reges (2006) and Ehlert & Schulte (2009). These aspects, and the associated work, are considered in more detail in Chapter 5.

Research on tool support for introductory programming examines the use of software tools specifically designed to support the needs of novice programmers, including work on automated assessment by Ala-Mutka (2007) and Douce et al. (2005), visualisation (Naps et al. 2002), and programming environments (Gross & Powers 2005, Kelleher & Pausch 2005, Kölling et al. 2003, Mason & Cooper 2013).

2.3.3 Applying Constructive Alignment to Introductory Programming

Constructive alignment has also been seen as an avenue for improving outcomes in introductory programming. The work of Armarego (2009) concluded that traditional teaching approaches did not align well with the requirements of the computer science and software engineering disciplines, and may actually inhibit students, both current and potential, from engaging with associated topics. In agreement with this we suggest that an effective means of applying constructive alignment to the teaching of introductory programming is, therefore, imperative.

Thota & Whitfield (2010) and Gaspar & Langevin (2012) have each aimed to use the principles of constructive alignment to enhance the teaching of introductory programming. Thota & Whitfield (2010) described the development and delivering of an object oriented introductory programming unit that aligned with cognitive and affective learning outcomes. While the work of Gaspar & Langevin (2012) discussed how the principles of constructive alignment represented a need to explicitly cover programming thought processes in teaching introductory programming, but did not discuss an overall application of these principles to the development of an associated unit of study.

The work of Thota & Whitfield (2010) proposed a holistic and constructively aligned approach to introductory programming that aligned teaching and learning activities and assessment tasks with cognitive and affective learning outcomes. The described unit involved five intended learning outcomes related to object oriented programming, listed in Figure 2.12. Each of the intended learning outcomes was mapped to programming concepts and techniques, and then to the assessment activities using a tabular matrix as discussed earlier in Section 2.2.

Constructive learning theories were embedded by Thota & Whitfield (2010) through the use of group work and pair programming. Teaching and learning activities included interactive lecture and laboratory classes, and the use of role plays and oral presentations. Adaptive quizzes, peer feedback, and lecturer feedback provided students with a range of useful resources to learn from. The use of class diagrams and an

ILO-1: Demonstrate knowledge and understanding of essential facts and concepts, relating to OOP.

ILO-2: Deploy appropriate theory, practices and tools for problem definition, specification, design, implementation, maintenance and evaluation of programs.

ILO-3: Use object-oriented design as a mechanism for problem solving as well as facilitating modularity and software reuse.

ILO-4: Work productively as part of a pair/team.

ILO-5: Demonstrate ability for organisation and internalisation of values.

Figure 2.12: Intended learning outcomes from Thota & Whitfield (2010).

interactive debugger also helped students comprehend critical aspects of objects and classes.

Assessment in the unit reported by Thota & Whitfield (2010) was divided up into a number of tasks based upon the intended learning outcomes, and included the use of quizzes (5%), an exam (10%), programming assignments (30%), group project (50%), classwork, and a journal (5%). With an additional stipulation that all intended learning outcomes had to be met to a satisfactory level, though no indication of how this was checked was provided.

Thota & Whitfield (2010) reported higher mean scores for deep approach to learning than surface approaches, using Biggs et al. (2001) two-factor study process questionnaire (R-SPQ-2F). A positive relationship between deep approach to learning and course grades, and a negative relationship between surface approaches and grades, was reported but found to be not statistically significant. Informational resources were considered more important by students as a source of their learning, than collaboration, which group work being cited as a source of conflict in the unit.

Overall the work of Gaspar & Langevin (2012) and Thota & Whitfield (2010) are representative of the work on applying constructive alignment in general. Gaspar & Langevin (2012) refers to the principles of constructive alignment (Biggs 1996) as a means of suggesting a range of changes, but did not discuss embedding the principles more deeply. While Thota & Whitfield (2010) provide a more genuine attempt at embedding these principles throughout their unit, the use of traditional assessment approaches could easily have resulted in unintended misalignment, which can be further eroded through significant use of group marks in students final grades. In these contexts it is hard to see how any given student's learning outcomes relate to a unit's intended leaning outcomes.

2.4 Summary

Biggs (1996) vision of constructive alignment provided a strong and compelling argument for focusing on **student-centred** approaches to learning and assessment. Biggs' example unit demonstrated clear alignment between the intended learning outcomes and assessment, with learning activities providing students with the means of addressing these outcomes in the preparation of their portfolios. In contrast, the reported applications of constructive alignment have remained primarily **teacher-centred**, with teaching staff defining teaching and learning activities that are then assessed using traditional weighted assignments and exams.

While existing approaches examined in the literature review presented in this chapter can be argued to be applications of "Constructive Alignment", they do not appear to have captured the dramatic shift in thinking reported by Biggs (1996). The student-centred approach of the portfolios, with alignment to intended learning outcomes at their core, does not appear to have gone beyond Biggs' original work. If the papers analysed in this review are representative of "Constructive Alignment", then the work reported by Biggs goes beyond the core principles to capture additional aspects that are beneficial to student learning. Rediscovering these principles and trying to recreate the "web of consistency" is a worthwhile pursuit, and a goal of this research.

3
Guiding Principles

This chapter describes twelve principles that underlie our model for delivering constructive aligned introductory programming. These principles act as guidelines for decision making, and in many ways underpin the model in the same manner as a unit's intended learning outcomes underpin constructively aligned teaching. Each aspect of the model, the associated curriculum, teaching and learning activities, and assessment tasks are aligned with one or more of these principles.

The principles cover both *how* the teaching and learning environment should operate, and *what* should be taught. Originally derived from constructive alignment, the *how* principles centre on constructivism and aligned curriculum. In relation to *what* should be taught, the principles draw upon computing education literature and our own experiences as educators and software developers.

Reflective practice played an important part in the formation of these principles, and both sets of principles have developed over the course of this research. This chapter presents the current working principles we use to guide the development and delivery of introductory programming. While most were present throughout the research, their individual emphasis and relationships have developed through our reflective practice.

The chapter first outlines the principles related to *how* we aim to teach introductory programming, in Section 3.1. These principles relate to the core principles of constructive alignment, described in Chapter 2, but also aim to more tightly integrate constructivist theories, with their strongly student-centred focus. Following this, Section 3.2 presents the principles related to *what* we aim to teach, specifically covering details on what we aim to focus on in teaching introductory programming. The full set of principles is this summarised in Section 3.3.

3.1 Principles to Guide HOW We Should Teach

Interventions that impact on the learning environment, also referred to as the teaching or academic environment, have been shown to have the potential to positively influence student learning outcomes (Trigwell & Prosser 1991). Learning environments have also been found to influence students' approach to learning (Entwistle & Tait 1990, Entwistle 1991, Kember & Leung 2007) and perceptions of teaching environments have been shown to directly, and indirectly, influence learning outcomes (Meyer & Muller 1990, Lizzio et al. 2002). The aim of these first principles is therefore to create a positive learning environment for students; one that is demanding of students but supports and rewards their efforts to understand the concepts required of the curriculum.

The following list outlines these principles for educators:

P1 : Recognise that students construct knowledge in response to activity.
P2 : Align activities and assessment to intended learning outcomes.
P3 : Assess learning outcomes, not learning pace or product outcomes.
P4 : Focus on important aspects, while providing access to necessary details.
P5 : Communicate high expectations.
P6 : Actively support student efforts.
P7 : Trust and empower students to manage their own learning.
P8 : Be agile and willing to change in response to measurable indicators.
P9 : Embed reflective practice in all aspects.

These first nine principles relate to *how* the teaching and learning environment should operate and could, therefore, be applied to a range of teaching and learning contexts and topic domains.

Individually each principle has its own merits but they are designed to work together. As a whole, each principle interacts with the other principles to create a productive student-centred learning environment. Figure 3.1 shows the key interactions between these principles. The students active construction of knowledge is central, with various aspects of this being supported by the other principles. Each principle is discussed in detail in the following sections, with the various relationships being discussed along with associated literature.

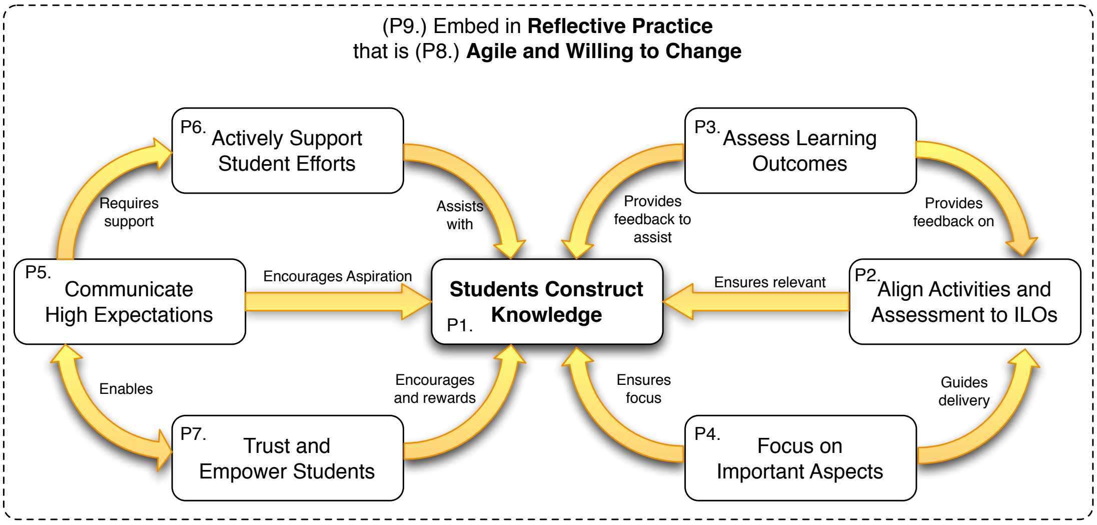

(P9.) Embed in **Reflective Practice**
that is (P8.) **Agile and Willing to Change**
P6. Actively Support Student Efforts
P3. Assess Learning Outcomes
Requires support
Assists with
Provides feedback to assist
Provides feedback on
P5. Communicate High Expectations
Encourages Aspiration
Students Construct Knowledge P1.
Ensures relevant
P2. Align Activities and Assessment to ILOs
Enables
Encourages and rewards
Ensures focus
Guides delivery
P7. Trust and Empower Students
P4. Focus on Important Aspects

Figure 3.1: Key interactions between proposed principles for educators

3.1.1 Recognise Students Construct Knowledge in Response to Activity

Decisions about curriculum, teaching and learning activities, and assessment tasks are all guided by the educator's theory of teaching and learning (Argyris 1976, Ramsden 1992). While constructivism is often promoted by educators, Phillips (2005) observed that constructive learning theories have not transitioned to common education practice, resulting in a "dissonance" between the elements of effective learning and the characteristics of typical university learning environments. This is symptomatic of the disconnect between educators espoused theory and their theory-in-use (Argyris 1976). To successfully implement constructive alignment it is, therefore, important to identify and adopt the key aspects from constructivism, as outlined by Biggs (1996), Biggs & Tang (1997) and in Biggs & Tang (2007). By adopting constructivism as our theory-in-use we aimed to create an educational setting that was "in harmony" with the principles of constructive alignment.

Central to all forms of constructivism is the principle that learning is an *active process* requiring the leaner to construct their own understanding through individual and social activity (Biggs 1996, Cobb 1994, Duffy & Cunningham 1996, Duffy & Jonassen 1992, Glasersfeld 1989, Jonassen 1991*b*, Steffe & Gale 1995, Vrasidas 2000). To incorporate central ideas from these writings on constructive learning theories the following aspects of constructivism are actively embedded in the model presented in the next chapter:

- Knowledge is constructed, not transmitted via communication alone.
- Teaching involves creating a context in which learners are able to construct ap-

propriate cognitive models through individual and social activities.
- Errors in understanding are opportunities for further learning, as these help indicate the students' current level of development and can be used to guide future learning activities.

Biggs (1996) reason for adopting constructivism as a central philosophy was due to its emphasis on the students active role in constructing their own knowledge. When taken to an extreme this can result in approaches that rely upon students building their own understanding from "first" principles, possibly isolated concepts and without structure. Such approaches are promoted in discovery learning (Bruner 1961) and in some constructivist writings, such as in Glasersfeld (1989) and Duffy & Cunningham (1996). The unstructured nature of these teaching and learning environments have received strong criticism.

Anderson et al. (1998) criticises constructive learning theories when "pursued to unproductive extremes", as in the case with discovery learning, and argue that there is significant evidence of the benefits for guided instruction from the area of cognitive psychology. Mayer (2004) also argues against discovery learning, instead suggesting that constructivist views of education may be better served through cognitive activity, instructional guidance, and curricular focus. Furthermore, Kirschner et al. (2006) argue against discovery learning, indicating that in highly complex environments, such as software development and introductory programming, free exploration may generate a heavy workload and detrimentally affect learning.

Embedding constructivism in the model proposed in this research required accepting the central role of the learner in constructing their knowledge, while avoiding detrimental aspects associated with taking these ideas to their extreme. This approach will temper constructivism with certain practical details, an approach we feel is in line with the principles of constructive alignment as originally proposed by Biggs (1996). These details include the following:

- Communication remains a valuable tool for educators to help shape the learning context, but should not be seen as a means of knowledge transfer.
- Guided instruction is valuable and ensures student activity is likely to be productive, though students should also have opportunities to explore content in a context meaningful to them individually.
- Deliberate practice provides students with opportunities to engage with principles in action, but these activities should include opportunities to reflect on important aspects learnt.

3.1.2 Align Activities and Assessment to Intended Learning Outcomes

In order to implement constructive alignment we need to work iteratively toward achieving a "web of consistency" (Biggs 1999) in which we optimise the likelihood of students *engaging appropriately* with learning activities and assessment tasks. Biggs (1996) indicated that this can be achieved through aligning learning activities and assessment tasks to the unit's intended learning outcomes.

The alignment of activities and assessment to intended learning outcomes is critical for our model. As shown in Figure 3.1, this alignment is seen as supporting student construction of knowledge. By aligning teaching and learning activities to the intended learning outcomes we ensure that students are constructing the required understanding. Similarly, by aligning assessment with these same intended learning outcomes we ensure that students are adequately prepared for this assessment and that the assessment is evaluating students' attainment of the stated learning outcomes.

One potential issue identified in Chapter 2 is relying too heavily on staff to perform this alignment. This has the effect of placing the intended learning outcomes outside the process, interacted with only by staff when reporting their alignment to activities and assessment. This was shown visually in Figure 2.11, with the students always two steps from the intended learning outcomes, which provided additional opportunities for misalignment.

Figure 3.2 presents an altered version of Figure 2.11 from Chapter 2. This illustrates how the intended learning outcomes become central to this process when students are included in the alignment process. The central role of the intended learning outcomes reduces the opportunities for misalignment, as now they form a central aspect shared by both staff and students. Staff and students now need to develop a shared understanding of the intended learning outcomes, and collaboratively work toward students being able to demonstrate they have met all outcomes. In this way, it should be possible to achieve the "web of consistency", and hopefully thereby improve the chances that students engage appropriately with learning activities and assessment tasks.

3.1.3 Assess Learning Outcomes, Not Learning Pace or Product Outcomes

Assessment in education is often seen as serving one of two possible purposes: supporting learning, or evaluating outcomes. These two purposes are known as *formative assessment* and *summative assessment*, as proposed by Scriven (1967) and Bloom (1969).

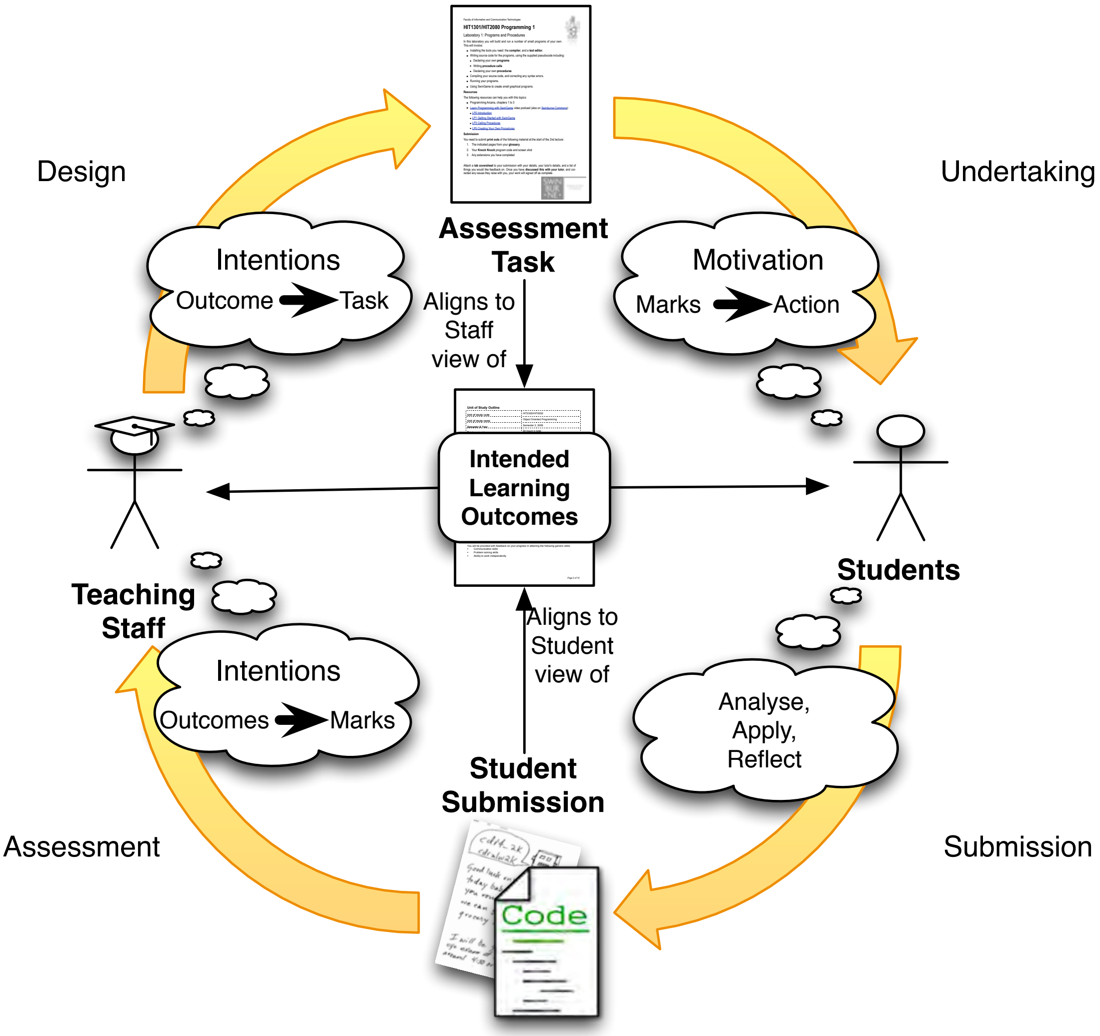


Figure 3.2: An altered version of Figure 2.11 with students and staff now actively involved in aligning work to the unit's intended learning outcomes

Formative assessment aims to assess student learning for the purpose of providing *feedback*. This is distinct from summative assessment where the aim is to assess how well students have performed on a certain task, typically to determine a final *grade*. Biggs & Tang (2007) suggest that for clarity the two forms of assessment are best referred to as *formative feedback* and *summative grading*.

The important role of formative feedback in education is widely reported. Ramsden (1992) indicates that, of all items on the Course Evaluation Questionnaire (Ramsden 1991), the one that most clearly distinguished between the best and worst courses related to the provision of helpful feedback. Wiliam (2006) described the use of formative feedback in short, medium and long cycles to improve student learning, and indicated that – to be formative – outcomes of the assessment must be used by students to make adjustments to better meet their learning needs. Black & Wiliam (1998) showed that substantial learning gains can be achieved by innovations designed to strengthen frequent feedback students receive. Furthermore, Black & Wiliam (1998) report that students pay more careful attention to feedback when there are no associated marks, or put another way "marks" reduced student attention to formative feedback. In discussing assessment for learning Brown (2004) stated that "Formative feedback is critical" and that "feedback must be at the heart of the process" if we are to make assessment an integral part of learning.

Gibbs & Simpson (2004) listed ten conditions under which assessment assists with student learning. These ten conditions can be grouped into three main points, as shown in the following list.

- Assessment tasks are aligned with intended learning outcomes, and provide students with sufficient work to ensure they engage appropriately with the required learning.
   1. Tasks provide students with enough work to require sufficient time on the task.
   2. Tasks direct students to spend appropriate time and effort on the most important aspects of the unit.
   3. Completing tasks is likely to engage appropriate kinds of learning.
- Feedback is constructive in nature, providing information that students will be able to use to develop their understanding of associated concepts.
   4. Feedback is both timely and sufficiently detailed.
   5. Feedback focuses on demonstrated learning outcomes, and on actions students can control.
   6. Students receive the feedback while it is still relevant, and they are able to incorporate the feedback or seek further assistance.

7. Feedback needs to be in line with the purpose of the assessment tasks, and relate to its criteria for success.
8. Feedback should also relate to the student conception of the task, and their understanding of what they are supposed to be doing.

- Students utilise the feedback.

9. Students must receive, and pay attention to, the feedback.
10. Feedback should influence students future actions.

The experience of Smith & Gorard (2005) indicated that shifting to formative feedback requires more than simply removing summative marks. In their case study, Smith & Gorard (2005) reported significantly worse results for the group of early secondary student who received only formative feedback. In discussing their results, they indicate that, in this case, feedback comments were often not constructive, were misunderstood by students, and were not integrated back into the teaching and learning context. It appears that in the case examined by Smith & Gorard (2005) the assessment remained primarily summative in nature with marks being replaced by comments, resulting in many of the conditions raised by Gibbs & Simpson (2004) not being met.

Interestingly, in proposing the use of formative feedback in education, Bloom (1969) indicated that an assessment item can play both formative and summative roles, though he suggested that in this case the formative feedback will be less effective. The issue is that the different purpose behind the two forms of assessment will result in students approaching each in a different way. To make the most of formative feedback, the ideal strategy for students is to highlight their misunderstandings and draw attention to what they do not know. By doing this, students will receive feedback that is more relevant to their current situation and it provides them with the advice they need to make progress. In contrast, misunderstandings result in lower grades when the assessment is summative. As a result, summative grading encourages students to hide their misunderstandings. In extreme cases students plagiarise others work in an attempt to hide their own misunderstandings, something that does not make sense when the work is formative in nature.

If used effectively, formative feedback can be used to focus students on gaining required levels of understanding. With summative assessment marks for assessed work is typically final, meaning that students have little incentive to incorporate feedback they may receive in addition to the summative marks. The focus is on the marks gained, not on discovering opportunities to learn from mistakes. With formative feedback the process can be seen as ongoing, with the assessment just one step toward gaining understanding. Student can then use the provided feedback to engage in additional learning, helping them to address any misunderstandings. Figure 3.3 provides an illustration of these two different views on assessment, highlighting the ongoing nature of formative feedback.

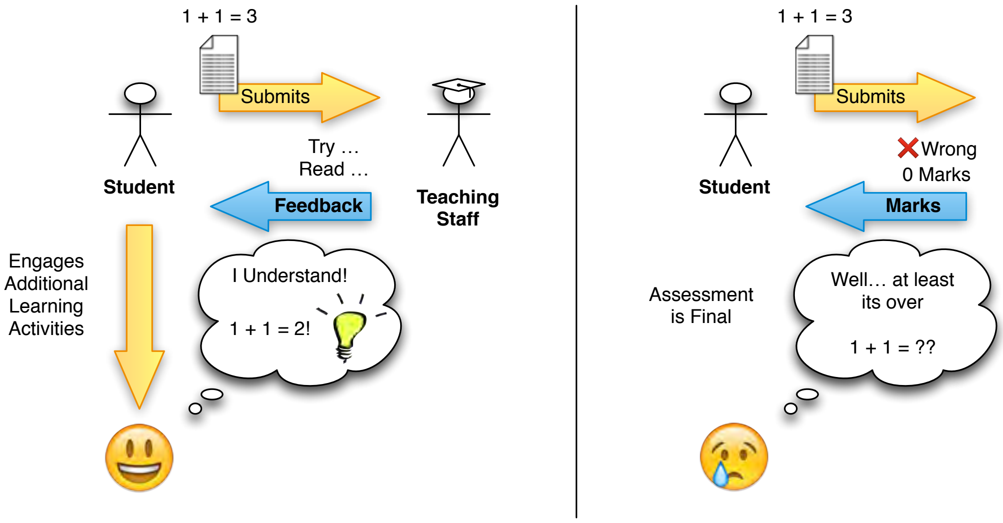


Figure 3.3: Formative feedback enables an ongoing learning process, with feedback providing details on how work can be completed rather than being an end in itself.

Figure 3.4 highlights another issue with the use of summative assessment during the teaching period. This figure shows three hypothetical students, Student "A", Student "B" and Student "C", and their depth of understanding over the teaching period. At the start of the teaching period each student comes in with existing knowledge, and during the teaching period they construct additional knowledge. When Assessment 1 (A1) is performed each has achieved some level of understanding that is compared against an expected level of achievement. When summative grading is used at this point Student "C" has not made sufficient progress and would receive a low grade: they have learnt too little. At the same time Student "A" has not been challenged by this assessment, and there is little recognition for this students advanced understanding: they have learnt too much. Student "B" is somewhere between these two, and therefore receives a "good" grade: they have learnt just enough. The first assessment is punishing Student "C" for learning these topics too slowly, and is discouraging Student "A" by not recognising their current level of understanding. There is also little distinction between Student "A" and Student "B", from this perspective they are the same.

Similar patterns occur for the second assessment (A2) later in the teaching period. Student "A" is still not begin challenged, Student "B" is progressing nicely, and Student "C" is struggling. For Student "C" this negative reinforcement may possibly discourage them from attempting to master the concepts, and reinforce any negative opinion they have of the field in general. In this case, however, Student "C" more fully grasps the concepts after the completion of the second assessment. The concepts start to come

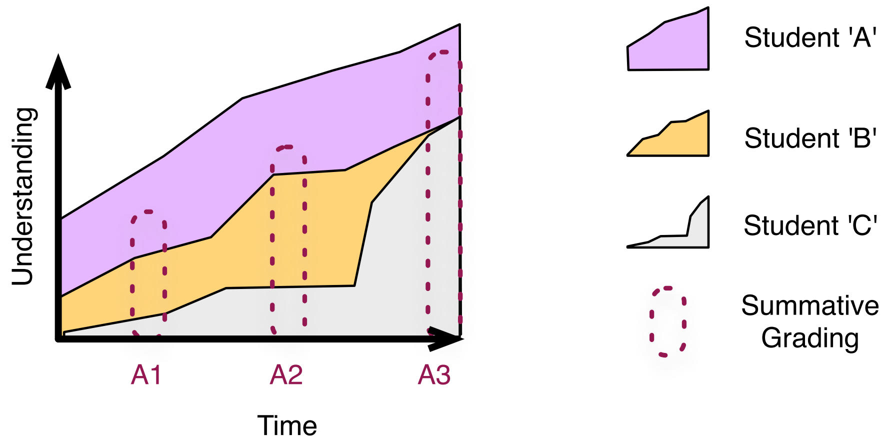


Figure 3.4: A hypothetical scenario, showing summative grading measuring pace of learning.

together, and by the end of the unit Student "B" and Student "C" have a comparable depth of understanding. This result will, however, not be reflected in their results. Given that each of the assessment items in our hypothetical unit were summative, Student "C" will have lost significant marks from the first two assessment items. In effect, the summative assessment is not only assessing the final level of understanding, but also the pace at which the student was able to achieve this understanding.

This grading during the semester is also not effective for helping the high achieving Student "A." At no point has their advanced knowledge been recognised, and the constrained nature of the assessments are not likely to have helped their development.

Figure 3.5 shows an alternative picture, one that makes use of formative feedback and delays all summative assessment to the end of the teaching period. At each of the assessment points during the semester the students each receive formative feedback based on their individual level of understanding demonstrate at that point. The advanced standing of Student "A" can be recognised, and the student can be encouraged to further their understanding with additional resources and advice. Student "B" can be congratulated for making good progress, their misunderstandings can be addressed and they can be advised how best to proceed with the upcoming material. Student "C" can be offered additional support or directed to useful resources, their lack of progress is not punished but used to indicate the student needs additional help. This more personal help and attention should result in improved learning, but even when the progress remains the same the summative assessment at the end of the semester is still a better representation of the students' learning outcomes.

To best support student construction of knowledge the model presented in the next chapter aims to maximise the use of formative assessment: assessment that supports

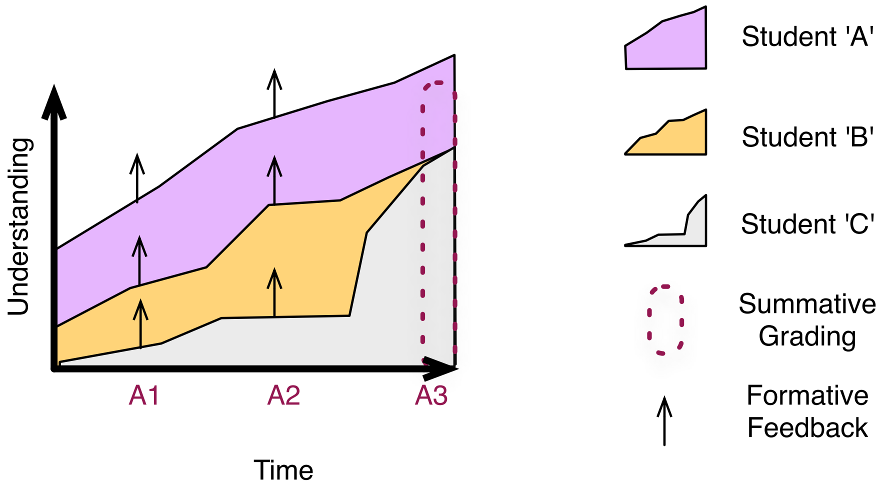


Figure 3.5: An alternative to Figure 3.4, showing formative feedback supporting learning during delivery with summative grading at the end.

learning. This involves the use of frequent formative feedback during the semester to aid students in developing appropriate understanding, and delaying summative assessment until after unit delivery.

Our assessment should, as much as possible, focus on providing feedback on student understanding and ability to meet the intended learning outcomes. We want to focus on more than just the "product" outcomes from the teaching and learning activities. Assessment tasks need to include aspects that require students to articulate their current understanding of concepts. This can then be used to help determine the students current level of understanding, and errors evident in this work provides opportunities for students to learn from their mistakes.

This principle relates to both students construction of knowledge and to the alignment of activities and assessment, as shown in Figure 3.1. Formative assessment of learning outcomes during delivery helps students in the construction of knowledge, providing opportunities to learn from their mistakes without fear of losing marks. These formative tasks also help both staff and students with the alignment of teaching and learning activities and assessment tasks. Staff can use the identified misunderstandings to help guide students individually, and to change or adjust teaching and learning activities where needed. For students, this ongoing focus on articulation of understanding and receiving feedback will ensure they are suitably prepared to demonstrate how they have met all of the intended learning outcomes in the final summative assessment.

The summative assessment also contributes to both students construction of knowledge and to the alignment of activities and assessment. For final unit grades students will need to demonstrate how their understanding aligns with the units intended learning outcomes. The assessment needs then to aim to assess the learning outcomes, evaluating how suitable the students level of understanding is at the end of the unit. The SOLO taxonomy (Structure of the Observed Learning Outcome) proposed by Biggs & Collis (1982) provides effective guidelines for performing such an assessment.

To summarise, the model presented in Chapter 4 aims to enhance learning outcomes through the use of frequent formative feedback. This will meet the requirements taken from Gibbs & Simpson (2004), thereby avoiding the issues raised by Smith & Gorard (2005). To be effective, formative feedback must be communicated effectively to students and provide them with clear means of addressing any shortcomings.

3.1.4 Focus on Important Aspects, while Providing Access to Necessary Details

Adopting constructivist learning theories requires a recognition that ideas cannot be directly communicated, and that teaching is therefore not about presenting the required material but guiding students to actively construct their own understanding of the required topics. To accommodate this change in approach, teaching staff need to carefully plan communication so that it focuses students appropriately on the most important aspects. In communicating concepts it is important to focus on only core aspects, ignoring unnecessary details. To borrow an idea from communication theory, we aim to have as high a signal to noise ratio (Shannon 1949) as possible, ensuring students will not miss important details amongst the noise, no matter how interesting teaching staff find particular side issues.

Presentation of teaching and learning resources need to take into consideration their purpose when determining what is communicated. For example, lectures are not a suitable means of communicating details, but could be useful for inspiring students, and motivating them to learn a particular topic. In these situations, the focus should therefore be on providing only key aspects necessary for students to get started with a topic, or important concepts to guide their thinking. Further resources can then provide students with access to relevant details as they are required, and when they will be appropriate for each student.

With limited resources, primarily time, a classic depth-vs-breadth trade off also needs to be considered, when thinking about the focus on a particular unit. Given that we have fixed time, it is important to aim to cover an appropriate breadth and depth of topics, as illustrated in Figure 3.6. Different units could aim either for a wide breadth and shallower depth, or for a narrower focus with a deeper depth.

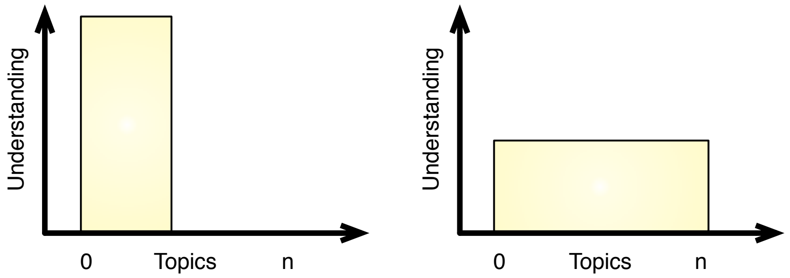


Figure 3.6: Given a fixed teaching "volume", a unit can cover either a breadth of topics or fewer topics in depth.

The SOLO Taxonomy (Biggs & Collis 1982) provides several levels that can then be used in defining appropriate intended learning outcomes, as described in Biggs (1996) and further elaborated upon in Biggs & Tang (2007). These have been found to be effective means of communicating intended learning outcomes across a range of fields and units (Brabrand & Dahl 2007, 2009). Biggs recommends that university level degrees should aim for relational levels of understanding, indicating that in many regards a focus on depth is more appropriate than a shallow understanding across a wider range of topics.

Interestingly, the study conducted by Schwartz et al. (2009) found that high school students who reported studying a major topic in depth earned higher grades in college than those who reported covering no topics in depth. If this translates to undergraduate computing education, then a depth of understanding in programming may help students succeed with other computing units. The strong correlation generally observed between programming skills and other computing skills (McGettrick et al. 2005) supports this idea.

As programming is central to the discipline of computing (McGettrick et al. 2005) it is important to focus on building depth of understanding, over covering a breadth of topics. This is particularly challenging as external parties seek to inject more content into curriculum. There is a constant pressure to include details on contemporary topics such as developing for mobile devices, web application programming, building graphical user interfaces, using certain programming languages, development environments, or software tools. Introducing such topics may have some merit, but introduces the risk of inappropriately focusing students attention. A successful introductory unit should focus on building sufficient depth of understanding such that additional topics can be learnt by the student on their own. In contrast, where a breadth approach is taken students may learn contemporary tools at the risk of missing the underlying concepts that would enable them to move beyond what they have learnt.

By focusing on important aspects the teaching and learning activities and assessment tasks will help guide students in the construction of their knowledge, and ensure that they align to sufficiently deep cognitive levels. As tasks define what students will do, this in turn helps to ensure that students are developing appropriately deep knowledge in relation to the intended learning outcomes.

3.1.5 Communicate High Expectations

Communicating high expectations is an important part in the development of the model. In listing their principles for good undergraduate education, Chickering et al. (1987) include *communicating high expectations* as one of their seven principles. Chickering et al. (1987) state "expect more and you will get more" and indicate that high expectations are important for everyone – from those who are poorly prepared or unwilling, to exert themselves to those who are bright and motivated. Similarly, Klem & Connell (2004) reported that elementary and secondary school students were likely to be more engaged with learning if they perceived their teachers as using a well structured environment with high expectations.

Believing in students' potential is also key to the approach presented by Soetanto (2003, 2012). Soetanto's approach aimed to improve students' discipline, confidence and belief in their potential, and units delivered with this approach have gained in popularity despite their technical difficulty and being delivery in a foreign language (English).

By having high expectations of our students we aim to build student confidence and to get them to aspire to excellence. High expectations require students to work hard throughout the unit's delivery, providing encouragement to spend sufficient time on the teaching and learning activities. This then requires both time and energy from students, which should improve outcomes: as stated by Chickering et al. (1987) "time plus energy equals learning."

3.1.6 Actively Support Diverse Student Efforts

Both Chickering et al. (1987) and Soetanto (2003, 2012) indicate high expectations should also apply to teaching staff. If the students are to meet our high expectations, they will need our active support. This will need to extend beyond providing formative feedback, to providing active support throughout the process. Given the technical nature of introductory programming and the exacting nature of a compiler, students typically face numerous challenges.

This requires the recognition of student differences, and preferred learning styles. The various works on learning styles (Coffield et al. 2004) indicate that individuals have different preferences with how they approach their learning. By providing support for a range of learning approaches, and through using different ways of communicating the same ideas, the model aims to help students in the construction of their knowledge. Providing a range of resources will enable students to approach concepts and topic from a variety of angles.

3.1.7 Trust and Empower Students to Control their Own Learning

Student motivation has a significant impact on learning. In terms of strategies for improving student motivation, McGregor's work on motivational strategies in business (McGregor 1960) provides some insight into similar strategies that could be applied to education, these ideas are summarised in Table 3.1. In regards to personnel management, McGregor identified two categories of managers perceptions of their employees: named "Theory X" and "Theory Y."

- Traditional businesses were seen to use coercion or persuasion as a strategy to motivate employees to achieve required levels of productivity. These strategies are used when managers adopt the view that employees *do not want to work* and *cannot be trusted*, McGregor (1960) named this understanding of human motivation as Theory X.
- In contrast, Theory Y assumes that, given the right conditions, *people want to work*, that they *can be trusted* and will do their best work when they are.

While originally applied to business organisation management, these views can also be applied to an educational setting. Markwell (2004) categorised Theory X and Theory Y positions for educational settings, as summarised in Table 3.1. In the educational context Theory X can be categorised as:

- Being dominated by a negative view of students and their motivation.
- Seeing staff as central to the distribution of knowledge.
- Believing that students must be coerced into learning.

The role of the teacher is seen more as a *sage on a stage*. These attitudes are unlikely to lead to student-centred approaches to learning, and conflict with constructivist thinking.

The contrasting Theory Y position takes a more positive view of students, their potential and willingness to learn. Students are viewed as being naturally inquisitive, willing to learning, and capable of engaging appropriately with the learning activities. The role of the teacher is seen more as a *guide by the side*. The resulting teaching and learning context is likely to lead to student-centred approaches that are more in keeping with constructivist thinking.

These two perspectives represents the two extremes, and no one individual is likely to hold a pure Theory X or Theory Y position. Markwell (2004) differentiated between a Hard and Soft form of Theory X. Hard Theory X teachers focus on the punitive aspects of the assessment, focusing on the punishments for not following the rules. While Soft Theory X uses marks for encouragement, with the use of bonus marks or similar motivations. In contrast, Theory Y focuses on providing opportunities and resources for students to learn from.

In order to achieve many of the principles listed here it is necessary to adopt a predominantly Theory Y stance. The formative nature of the assessment tasks together with the high expectations will both require a level of trust in students that cannot be achieved with a predominantly Theory X stance. High expectations of students is a natural repercussion of a Theory Y position, and enhancing motivation in this way will ideally help students in the construction of their knowledge.

Table 3.1: Comparison of "Theory X" and "Theory Y" attitudes in education, adapted from Markwell (2004)

Theory X	**Theory Y**
Students have little desire to learn new material.	Students want to learn, learning is as natural to students as play or rest.
Students are inherently lazy and will attempt to get the material dumbed-down; the teacher must use a controlling environment to force students to learn and prevent cheating.	Students are not lazy; threats of diminished grades are not necessary to motivate students. The self-satisfaction from learning is sufficient to commit students to achieving the educational objectives.
Students prefer to be directed and do not want to be responsible for their own learning.	Students will naturally accept responsibility for learning.
The teacher must act as the source of information and actively transmit it to the students.	The intellectual potential of most students are being only partially utilized in the classroom.
Many students are not capable of learning the necessary material and can be expected to earn a low grade.	Imagination, ingenuity, and creativity are widely distributed within the student population and will be willingly applied to the learning process.

3.1.8 Embed Reflective Practice In All Aspects

In education the idea of reflective practice is to periodically look back at our teaching, and consider how things can be improved. The foundations of this idea can be traced back to Dewey (1933), though reflective practice itself was originally proposed by Schön (1983). Farrell (2007, 2008) identified two forms of reflective teaching practice: a strong form and a weak form. In its weak form, reflective practice involves informal evaluation of various aspects of professional practice that Farrell (2008) likens to a "thoughtful practice." The alternative strong form of reflective practice involves systematic reflection on teaching and taking responsibility for teaching and learning activities. For reflection to be effective, Richards & Lockhart (1994) state it must be used "hand-in-hand" with critical self-examination with reflection being the basis for decision making, planning and action.

Reflection plays an important role for both staff and students in the model presented in Chapter 4. Students undertaking units taught using this model will graduate and move into professional practice. Engaging them with reflective practice throughout their education will help ensure they are adequately equipped for lifelong learning (Field 2006). The active incorporation of frequent formative feedback provides a direct means of encouraging students to reflect on their work throughout the delivery of the unit, and the summative assessment should include some reflective aspects where students can reflect on what they have achieved in the unit.

The role of reflection, with respect to the teaching and learning activities, for students is shown in Figure 3.7. When each learning activity is concluded, students are encouraged to reflect on their learning. In this process students identify any areas they would like feedback on, and can use this to ensure their formative feedback is relevant to their current situation. By reflecting, and through the subsequent formative feedback, students reinforce the construction of their knowledge, and are able to inform their actions for upcoming teaching and learning activities.

At the end of the teaching period for a unit, students are encouraged to reflect on their unit as a whole. Students can reflect on their achievements, challenges overcome, work habits and other aspects they felt influenced their learning. This process of reflection should help students consolidate their knowledge, drawing into clear focus exactly what they have achieved and, hopefully, ways they can improve their learning in future semesters.

Figure 3.8 shows the role of reflection for teaching staff. During the teaching period staff reflect upon the delivery of the teaching and learning activities and student

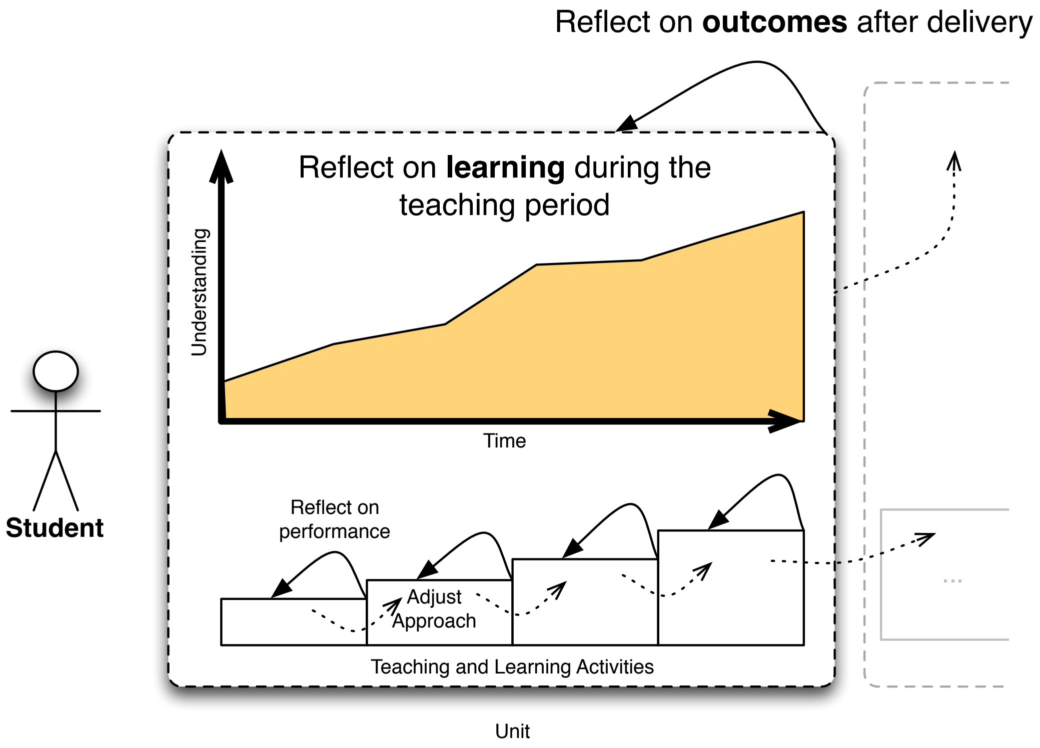

Reflect on **outcomes** after delivery
Reflect on **learning** during the teaching period
Understanding
Time
Student
Reflect on performance
Adjust Approach
...
Teaching and Learning Activities
Unit

Figure 3.7: Students reflect on their learning during the teaching period, and on the outcomes they have achieved after the teaching period.

progress. Common misconceptions of students identified in formative feedback can be used to update delivery during the teaching periods. After the teaching period students' results – both in terms of grade distributions and quality of evidence demonstrated in the final summative assessment – can be drawn upon to suggest changes for subsequent unit deliveries. Reflections on teaching should be shared amongst all teaching staff related to the unit to encourage reflective practice, and to facilitate collaborative improvements.

Reflection underpins all of the principles presented in this work. Students engage in reflection as a tool to help them construct their knowledge, and the formative feedback activities enable student and staff reflections. Staff reflect on teaching and learning activities, their alignment to learning outcomes, and the depth and breadth of coverage. Reflections enable educators to realistically manage expectations of students, the support offered, and to help balance trust with mechanisms to avoid inappropriate behaviour. Even the very nature and composition of these principles have been the focus of ongoing reflective practice.

To ensure ongoing improvements for both students and staff, the model incorporates reflective practice across all aspects. Students engage in reflection throughout the learning process, using their reflections to direct formative feedback and consolidate

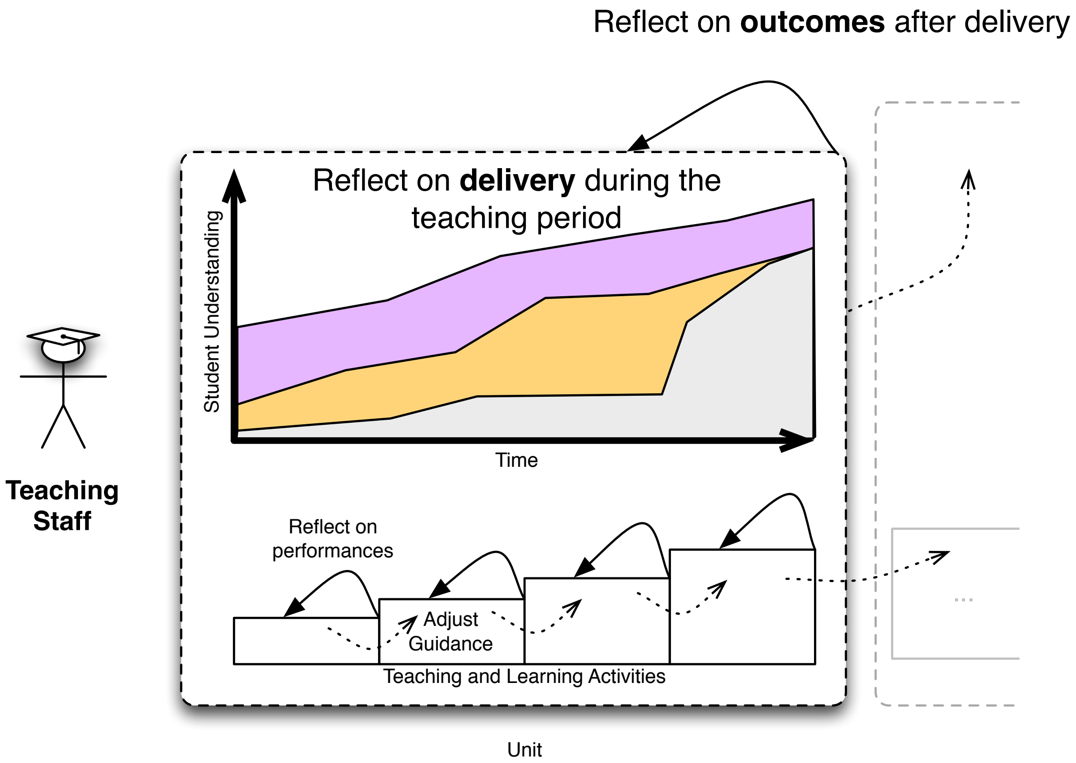

Reflect on **outcomes** after delivery
Reflect on **delivery** during the teaching period
Student Understanding
Time
Teaching Staff
Reflect on performances
Adjust Guidance
Teaching and Learning Activities
...
Unit

Figure 3.8: Staff relect on delivery during the teaching period, and on the outcomes students achieved after the teaching period.

knowledge. Teaching staff reflect on delivery, progress, and outcomes to improve the teaching and learning environment to better meet student needs.

3.1.9 Be Agile and Willing to Change

For reflective practice to actively enhance teaching educators must embrace change, focusing on aspects that will deliver the most value for students for the effort required. Similarly, as software developers this emphasis on delivering value by "*focusing on the things that matter most*" is reminiscent of the agile software development principles (Martin 2003). The agile software development community aimed to move away from heavyweight, documentation driven, software development processes toward a more *agile* approach focused on outcomes. Beck et al. (2001) documented the Agile Manifesto; the key priorities from a wide range of agile software development processes. The Agile Manifesto states the value of:

- **individuals and interactions** *over* processes and tools,
- **working software** *over* comprehensive documentation,
- **customer collaboration** *over* contract negotiation, and

- **responding to change** *over* following a plan.

In many ways the evolution of software development process from the traditionally heavyweight processes to lightweight agile processes can be likened to a shift from a predominant Theory X environment to one which is predominantly Theory Y. The focus on documentation and rigid control over the development process is being relaxed, and developers are being trusted and expected to deliver value.

If we are to affect change in education from a mark-driven Theory X environment, to a student-centred Theory Y environment, then the agile software development principles provide useful guidance to inform our model. To realise all of the principles presented in this chapter it is necessary to adopt similar priorities in our teaching, valuing things that help students construct their knowledge over other less valuable activities.

To help manage change effectively we view the education environment as consisting of an a) overall strategy, b) teaching and learning resources, and c) teaching and learning activities, as shown in Figure 3.9.

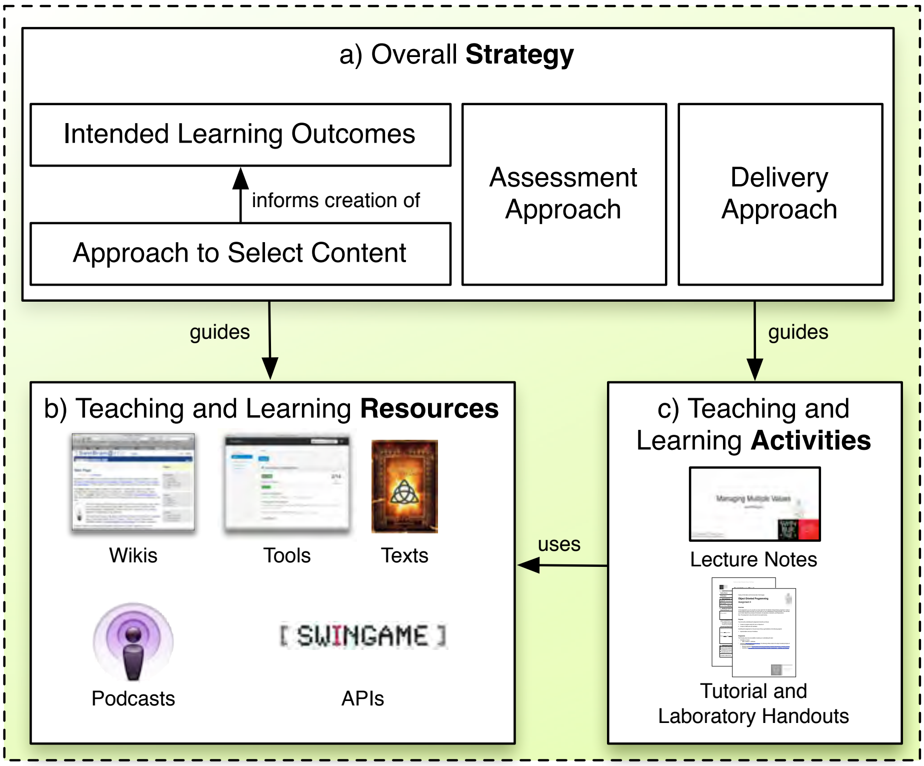

a) Overall Strategy
Intended Learning Outcomes
informs creation of
Approach to Select Content
Assessment Approach
Delivery Approach
guides
guides
b) Teaching and Learning Resources
Wikis
Tools
Texts
Podcasts
[SWINGAME]
APIs
uses
c) Teaching and Learning Activities
Lecture Notes
Tutorial and Laboratory Handouts

Figure 3.9: Relationship between a) overall strategy, b) resources and c) activities used to manage change effectively.

The strategy for delivering unit content informs the approach to material delivery, the choice of assessment approach, approach to selecting unit content, and the development of the intended learning outcomes. The central role of the intended learning outcomes means that the overall strategy should not change unless significant issues are identified in the approach overall. Teaching and learning *resources* can then be separated from the teaching and learning *activities*. In this way we can create reusable resources that are independent of the activities that are used in, meaning that activities can be adjusted more freely to better help direct student efforts. An overview of this approach is summarised in the following list, and each of these points are expanded upon in Chapter 4 and Chapter 5.

- Develop an *overall strategy* for delivering the unit content. Then:
    - Determine appropriate content, by focusing on important aspects and using principles related to *what* we aim to teach.
    - Derive intended learning outcomes, using appropriate verbs for the intended levels of the SOLO taxonomy.
    - Select an appropriate approach to unit assessment, and delivery.
    - Focus teaching and learning resources and activities on engaging students actively with the concepts related to the intended learning outcomes.
    - Avoid changing overall strategy, unless there are significant issues.
- Create teaching and learning *resources*.
    - Deliver material following the direction from the overall strategy.
    - Make resources generic and self contained.
    - See resources as supporting teaching and learning activities with additional details.
    - Invest in initial development for long term value; actively reuse and enhance over time.
- Design teaching and learning *activities*.
    - Use activities to focus student activity.
    - Actively review each semester, and rework based on reflection.

In developing the overall strategy several aspects need to come together: the intended learning outcomes, unit content, and approach to assessment and material delivery. Each of the principles presented in this chapter needs to help shape these aspects to ensure the overall learning context will enable the desired student centred focus. Chapter 4 includes a discussion on the application of these principles to select appropriate approaches to assessment, in Section 4.1.1, and approach for material delivery, in Section 4.1.3. Using these approaches, a number of approaches to content delivery are possible. The selected approach for the examples presented in this working are discussed in Section 5.1, with the two example units then presenting their associated intended learning outcomes in the remainder of Chapter 5.

Details related to the development and delivery of teaching and learning activities and resources also accompanies the two example units discussed in Chapter 5. This discussion illustrates how these principles helped shape the material developed for the example units, and lead on to a more in depth discussion of the supporting resources in Chapter 6. Details of the iterative development and delivery of these resources and activities is then discussed in Chapter 7.

3.1.10 Summary

This section presented nine principles to guide decisions that together should produce a student-centred, constructively aligned, learning environment. Each principle is backed by a range of education theories, and, if valid, together the principles should enable the creation of an environment that is demanding but supportive, focused on students building knowledge, agile, constantly improving through reflections, and accepting of various individual strategies and pace of learning.

3.2 Principles to guide what we should teach

Principles specific to teaching introductory programming are presented in the following list. While the general principles, presented in Section 3.1, helped shape the overall teaching and learning environment, these principles helped shape specifics in the curriculum, activities, and assessment tasks.

P10 : Set the strategy, and structure learning, around a programming paradigm.
P11 : Focus on programming concepts *over* language syntax.
P12 : Use programming languages as they were designed to be used.

Principle 10 is an implication of Principle 8, the need to be agile and willing to change, which proposed guiding the development and delivery of the unit by an overall strategy. This strategy will guide decisions on the topics to be included as well as the approach to delivering these topics. In terms of introductory programming, a key part of this strategy will be the choice of programming paradigm. Choosing this will help determine appropriate intended learning outcomes, and will guide the creation and delivery of teaching and learning activities and resources. Changing paradigm would require significant changes to teaching and learning activities, and resources, and so once chosen it will have a lasting impact.

Whichever paradigm is chosen, the constructive learning theories, encompassed in Principle 1, indicate that the focus of the delivery should be on guiding students, rather than trying to transfer the knowledge related to programming. This implies a focus on the underlying concepts, as these are fundamentally more important than details of the language syntax. This idea is further supported by Principle 4, with the goal of focusing on the most important aspects. This intention is captured in Principle 11, with the aim of using concepts as the central focus for unit delivery and assessment.

Lastly, Principle 12 indicates that we should aim to use programming languages in the way they were intended to be used. The goal here is to ensure we are always guiding students in appropriate ways of using programming languages, and designing programs. Good programming practices should be used throughout the unit, ensuring that there is consistency as students attempt to construct their knowledge of this challenging topic.

Each of these principles is expanded upon in the following sections, and linked to associated research.

3.2.1 Set the Strategy, and Structure Learning, Around a Programming Paradigm

Programming paradigms define fundamental programming styles and abstractions, and therefore have a profound impact upon the structure and outcomes of introductory programming units. In designing a new introductory programming unit, a programming paradigm – such as the procedural, functional or object oriented paradigm – needs to be chosen. This can then form the core of the overall strategy for a unit, and guide the development of its intended learning outcomes, teaching and learning activities and resources.

Which programming paradigm should be taught first is a popular, sometimes emotive, topic in computing education research. The following list indicates the range of approaches from imperative-first to objects-first approaches.

- Imperative programming first such as with Koffman (1988*a*).
- Cooper et al. (2003) suggest a course working with graphics prior to introductory programming to help students with problem solving skills.
- Felleisen et al. (2004) described an approach to teaching introductory programming using the functional paradigm (using the Scheme programming language).
- Howe et al. (2004) presented a components-first approach.
- Bennedsen & Caspersen (2004) argued for the use of a model-first approach using the object oriented programming paradigm.

With the predominant role of object oriented programming in industry, a common trend has been for academic institutes to move from imperative-first approaches to objects-first approaches.

In their discussion on learning and teaching of programming, Robins et al. (2003) summarise a range of research on both imperative-first and objects-first approaches to teaching introductory programming, and indicate that there is not likely to be one “best” approach. Lister et al. (2006) provided an in depth look at the research perspectives on the topic of objects-first, but in general there is no consensus on which approach should be taken. The objects-first approach has “failed” according to Astrachan et al. (2005) and Reges (2006), who report on shifting back to a procedural paradigm after having taught objects-first for a number of years. While Ehlert & Schulte (2009) reported no significant difference between approaches using objects-first and objects-later, their later work (Ehlert & Schulte 2010) indicated the objects-later approach had a greater comfort level for students.

During the design of the introductory programming units developed in this research work there was a need to choose which x-first approach to use. The common decision appears to be between the imperative-first, objects-first or functional-first approaches. Whichever approach is taken, the choice guides which concepts are covered in the unit and the order in which these can be tackled.

3.2.2 Focus on Programming Concepts

There are various views of programming in the literature. One perspective views programming from a *mathematical basis* (Denning 1989, Dijkstra 1989, Hoare 1969), others see it as an *exercise in problem solving* (Palumbo 1990), or as *modelling concepts* (Bennedsen & Caspersen 2004). Interestingly, most textbooks approach the topic through language syntax and features (Robins et al. 2003). In this work we avoided the predominant standard textbook approach, and instead focused on *programming concepts*.

A similar idea was expressed in Goldman (2004) as a *concept-based* approach. Their work introduced students to a number of "big ideas" related to software development using the JPie interactive programming environment. Our approach differs in that we focused on the "small ideas" that programs are build upon. In this way we build depth across these ideas and focus on the core programming abstractions, in line with our general principles (Principle 4).

Topics such as variables, procedures, control flow, etcetera are each approached as a concept. By teaching staff focusing on these concepts, it is hoped that students will engage meaningfully in understanding programming as a whole. Bruce et al. (2003) indicated that when teaching staff focus on developing student understanding and integration of programming concepts, it likely to lead to greater student satisfaction, with students being better prepared for subsequent programming units.

This concept-based approach is similar to the model-based approach of Bennedsen & Caspersen (2004), but applied to procedural programming concepts. By focusing on concepts we aim to provide students with reasons why the various programming features should be used, when different abstractions should be used, and help students develop a means of understanding abstract ideas. This concept-based approach can be applied to procedural programming concepts, object oriented programming concepts, and other programming paradigms.

Winslow's comparative study of expert and novice programmers (Winslow 1996) indicated that expert programmers tend to "abstract from a particular language to the general concept." By focusing on the concepts first we hope to instil similar expert-like ideas in students. Students are encouraged to create *conceptual programs*, which can then be mapped to code using the programming language syntax rules. Focusing first on concepts should encourage a depth of understanding, with students going beyond surface learning of the syntax used and thinking conceptually about what they are trying to achieve.

Programming concepts are tightly interrelated, and yet there is a need to provide a sequence of activities that can be introduced to students without overloading them initially. In designing these activities we aimed to ensure that concepts could be introduced to students in such a way as to reduce the amount of "magic" to a minimum. That is, limiting the cases where students have to do something without being able to reason why with the concepts they understand.

The aim of this concept-based approach is to enable students to explore language features. At each stage in the process students should have sufficient concepts to be able to understand the programs they are asked to create. To extend students further, they can be asked to experiment with features using the concepts covered to create programs they are personally interested in. Ideally this would enhance student motivation, and ensure they spend sufficient time on the task.

The focus on programming concepts means there is a need to:

- introduce programming concepts incrementally;
- provide students with time to put concepts into practice;
- see syntax as a means to an end, not an end in itself;
- avoid using language features before concepts that can explain their use; and
- map concepts to code using programming language grammars.

3.2.3 Use Programming Languages as they were Designed to be Used

While we agree with the "back to basics" approach of Reges (2006), we want to emphasise using programming languages in the way they were designed to be used. Our emphasis on programming concepts has deliberately relegated language to a secondary role, and one that can be changed by learning new syntax rules. Given this, changing between languages should be less problematic, and therefore we should not use a language simply for its industry relevance, as has been done by many institutions. Rather, language choice should be based on the programming concepts it supports, its availability across computing platforms, and the level of support it offers novices.

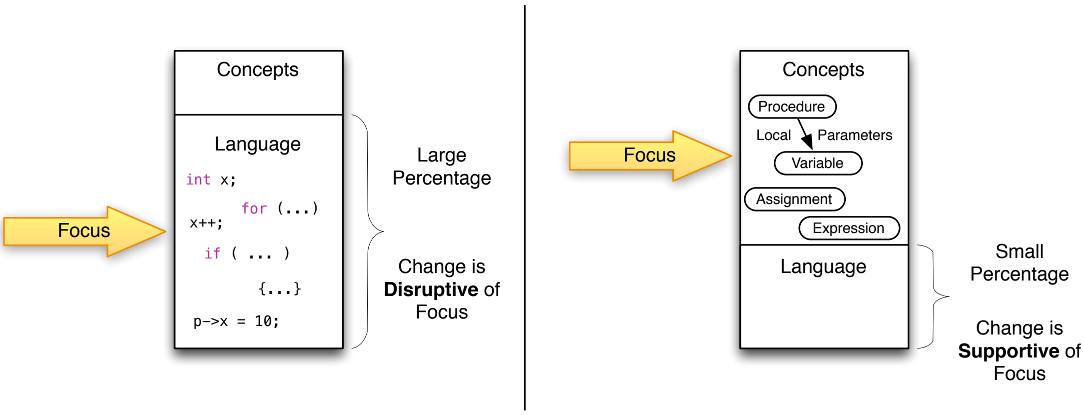
Programming language choice for introductory programming is as popular a topic in the Computing Education Research literature as the choice of programming paradigm. The following list covers a select number of well-cited papers on the topic of which programming language to use in teaching introductory programming. It is ordered by year, and shows the general shift from procedural languages to object oriented languages.

- Koffman (1988*b*) argued for Modula-2 over Pascal, PL/1 and Ada.
- Mody (1991) argued against C, and C++ for its lack of coherence, simplicity, understandability and implementability.
- Roberts (1993) discuss Stanford's shift from Pascal to C, addressing common issues with C by providing libraries to encapsulate complex features, emphasising procedural and modular abstractions, and focusing on the discipline of software engineering.
- Brilliant & Wiseman (1996) discussed programming paradigm and language selection for introductory programming. They concluded that there was no clear advantage to starting with either procedural programming, object-oriented programming, or functional programming paradigms. In terms of language they discussed moving from Pascal to Ada, C, C++, or Scheme.
- Böszörményi (1998) argued for Modula-3 over Java, discussing features that are useful to be taught in introductory programming yet missing from Java.
- Howell (2003) claimed that through structured labs, comprehensive grade sheets, in-class grading and frequent feedback any programming language could be used.
- Gupta (2004) suggested that a first language should strike a balance between being easy to grasp and supporting advanced concepts needed for later units.
- Kelleher & Pausch (2005) provided a detailed examination of a wide range of programming languages used for teaching introductory programming. Their work discussed various efforts to help make programming more accessible for novices, including systems that range in support from look at the way programs are expressed, how programs are structured, understanding of program execution, to systems that embed learning support. While the work reported on a range of systems it does not provide any recommendations on which language to use.
- Bishop & Freeman (2006) presented the pros and cons, from their experiences, in using the C# programming language, and provided some recommendations for those looking at using the language.
- Mannila et al. (2006) argued for the use of Python due to its simplicity, with examples comparing Python to Java.
- Mannila & De Raadt (2006) provided a set of criteria for comparing language

features for introductory programming. The study then compared several languages, with Eiffel, Java and Python achieving the highest scores.

- Pendergast (2006) provided a reflection on teaching introductory programming with Java over a number of years, highlighting some of the issues encountered and suggesting mechanisms to avoid them.
- Maloney et al. (2010) described the Scratch programming environment, a visual programming language designed primarily for students aged between 8 and 16.
- Anik & Baykoç (2011) used the Analytic Network Process methodology to help guide the decision of which language should be used first. The relative weightings given to the various aspects resulted in Java being ranked above the other languages.

Interestingly, underlying many of these papers is the idea that switching language is difficult. For example, Brilliant & Wiseman (1996) indicated that teaching multiple languages increases the overhead necessary to cover language details and peculiarities. While true, the relative proportion and disruption of this depends upon educators perspective as illustrated in Figure 3.10. When the focus is on using a language to achieve a task, switching language is highly disruptive as students have focused on learning the language itself. If the focus can be brought to the associated concepts, the perceived change is smaller, as the focus has been on underlying concepts that will apply in the new language. In this case the change is actually supportive of the focus, helping students to more clearly see the underlying concepts.



```
int x;
      for (...)
x++;
   if ( ... )
         {...}
p->x = 10;
```

Figure 3.10: When the focus is on the language, changing it changes a large portion of what has been covered and the change is seen as disruptive. When the focus is on the concepts, changing the language is now seen as supporting the focus on the underlying concepts.

By having a focus on programming concepts over language syntax this work aimed to tackle this problem from a different angle. In our previous teaching we noticed that many students tended to focus on, if not "cling" to, a particular syntax. Shifting language was a major effort as it was syntax they had learnt. Students felt they were "Java programmers" or "C/C++ programmers"; they did not see that they were learning something far more important; they were not learning a specific language, they were *learning to program*.

This focus also becomes evident when you examine the title of many introductory programming units. Introductory programming unit titles like "Introduction to Programming with C" and "Software Development in Java" give the impression that the language is very important, raising its role from supportive to central in the unit.

Choosing a language for a *concept-based* approach to introductory programming should be guided by the following principles, stated in a manner similar to the agile manifesto (Beck et al. 2001). We value…

- **Focus on programming concepts** *over* language syntax.
- **Teach students how to learn the language** *over* teaching the language explicitly.
- **Value languages that clearly support the concepts to be learnt** *over* the languages that are the current industry trend.
- **Use languages that support multiple platforms (operating systems)** *over* those tied tightly to a single platform.
- **See multiple languages as an important part of learning** *over* focusing on a single language.
  - Use multiple languages to encourage students to focus on concepts over programming language syntax.
  - Expose students to language differences enabling them to see different strengths and weaknesses.
  - Encourage students to see that they are *learning to program*, not learning *a programming language*.
  - Foster the attitude that language is a choice – students should not feel constrained to one language and should be open to possibilities other languages offer.

As with the Agile Manifesto, we agree that there is value in the items on the *right*, but we value the items on the *left* more.

A key aspect in this work that is different from other published work is the **deprecated** importance of the use of the language in industry. If a concept-based approach is successful, students will be able to quickly acquire skills in industry relevant languages in later units. Early programming units can focus on using languages that best meet educational requirements, while later units can cover language specific details and peculiarities concisely knowing students have an understanding of underlying concepts and the ability to learning new languages themselves.

3.2.4 Summary

This section proposed the use of a *concept-based* approach to teaching introductory programming. This approach focuses on teaching programming concepts directly, with use of the programming language grammars to help students map concepts to code. The aim of this approach is to help students focus on concepts, while providing them with tools they can use to learn any relevant programming language. The expected outcome of this approach is that programming language choice is much less important than which programming paradigm choice.

3.3 Summary of Guiding Principles

This chapter outlined twelve principles related to both *how* and *what* we aim to teach in our constructively aligned introductory programming units.

- Nine principles describe *how* the teaching and learning environment should operate:
  1. Recognise that students construct knowledge in response to activity.
  2. Align activities and assessment to intended learning outcomes.
  3. Assess learning outcomes, not learning pace or product outcomes.
  4. Focus on important aspects, while providing access to necessary details.
  5. Communicate high expectations.
  6. Actively support student efforts.
  7. Trust and empower students to manage their own learning.
  8. Be agile and willing to change in response to measurable indicators.
  9. Embed reflective practice in all aspects.
- Three principles help guide *what* we aim to teach:
  10. Set the strategy, and structure learning, around a programming paradigm.
  11. Focus on programming concepts, not language syntax.
  12. Use programming languages as they were designed to be used.

Chapter 4 will outline a model for constructively aligned introductory programming units. The model was created through the application of these principles, and the model's ability to embody these principles is discussed.

4

A Model for Constructive Alignment of Introductory Programming

Chapter 3 outlined twelve principles, nine for guiding *how* to create a student-centred learning environment, and three for guiding decisions on *what* should be taught in introductory programming. This chapter proposes a model for applying constructive alignment for teaching introductory programming that is in agreement with the twelve principles.

Section 4.1 describes an application of the principles in defining the overall strategy for teaching introductory programming, outlining the assessment approach used and the approach taken to deliver this material in a student centred manner. This section argues for the user of portfolio assessment, and provides some guidelines for designing and delivering lecture and tutorial classes. Following this, the general model for constructive alignment is presented in Section 4.2, which describes the overall model, the processes within it, and means for addressing plagiarism. The chapter concludes with a brief summary in Section 4.3.

4.1 Overall Strategy

One of the overarching principles from Chapter 3 is the requirement to be agile and willing to change (Principle 8). In discussing this principle, Section 3.1.9 outlined that teaching and learning resources need to be guided by an overall strategy. The overall strategy informs, and is shaped by, the assessment approach, approach to material delivery, and the approach to the selection of unit content. This section describes the application of the principles from Chapter 3 to the selection of an assessment approach and approach to material delivery. The discussion of approach to selecting content, and specific intended learning outcomes that follow, are presented in Chapter 5.

4.1.1 Assessment Approach

Assessment plays an important role in defining what students learn. Rowntree (1977) indicated the central role of assessment procedures in understanding any education system. This is supported by Ramsden (1992), who stated that "from our students' point of view, assessment always defines the actual curriculum", and further supported by Biggs & Tang (2007) who indicated that "students learn what they *think* they will be tested on." In presenting their conditions for effective assessment, Gibbs & Simpson (2004) discussed the dominant influence of assessment in defining what students focus on, indicating that students are able to distinguish between what assessment requires them to pay attention to, and what is likely to result in effective learning. So the selection of an assessment approach will have a significant impact on the overall strategy for both students and staff.

Consider a traditional introductory programming, taught using a number of assignments and a final examination. This approach, while commonly used, is not in keeping with several of our guiding principles, as outlined in the following list.

1. This approach to assessment is teacher-centred, and does not easily incorporate aspects from constructive learning theory, as discussed in Chapter 2. As a result, some adjustments are required in order to address Principle 1.
2. The teacher-centred nature of the assessment also excludes students from the alignment process, Principle 2, with staff performing the alignment of assessment tasks with intended learning outcomes resulting in additional opportunities for misalignment, as discussed in Chapter 2.
3. Similarly, the use of summative assessment during the semester goes against Principle 3 with its goal of assessing outcomes, and using frequent formative feedback to aid student learning.
4. The use of assignment marks to motivate students is also contrary to Principle 7, with marks being used for motivation in either a hard or soft form of Theory X strategy.
5. Marked assignments also indicate a finality, with marks being lost or gained when the assignment is assessed. These results are typically final, and therefore provide no incentive for student to reflect on their approach to learning, and to ensure they have more fully understood concepts before proceeding. Alternate assessment strategies, with a greater emphasis on formative feedback,

could better encapsulate the ideals of reflective practice, and thereby better meet Principle 9.

As a result, the principles from Chapter 3 requires us to consider other forms of assessment.

One strategy to address this would be to abandon coursework assignments, delaying final summative assessment to an examination worth 100% of the student's grade. While this would address Principle 3, such a heavy weight examination is very much a hard Theory X approach, and fails to recognise the value of coursework assignments, which have been strongly argued for as in Gibbs & Simpson (2004) which provided several strong arguments for coursework assignments, including the following:

- Units that include coursework assignments, in addition to an exam, resulted in higher average marks than compared with units that included no coursework assignments (Chansarkar & Raut-Roy 1987).
- Students prefer coursework assignments over examinations. A position that is also supported by Kniveton (1996) who stated that students prefer coursework assignments to exams as assignments assessed a better range of their abilities and enabled them to organise their time to a greater extent.
- Students attain better results from coursework than examinations. Gibbs & Lucas (1997) indicated a strong positive correlation between the proportion of coursework assignments and average marks. Though, James & Fleming (2004) indicated that individual students do not consistently perform better, or worse, on any one form of assessment.
- Coursework assignments are at least as valid a form of assessment as examinations:
  - Exams are a poor predictor of future performance (Baird 1985, Gibbs & Simpson 2004).
  - Coursework assignments are a better predictor of long term learning than exam results (Conway et al. 1992). This supports the idea that students adopt surface approaches to preparing for exams Marton & Säljö (1976*b*), Tang et al. (1999).
  - The quality of learning is deeper in assignment-based units, when compared to exam-based units (Tynjala 1998, Gibbs & Simpson 2004).

Examinations have also been found to be at odds with constructive learning theories. In discussing constructivism for computer science education, Ben-Ari (2001) indicated that test performances, and group work, were poor indicators of the conceptual models students had constructed about computer science. Ben-Ari (2001) suggested that the ideal constructivist assessment would involve observation and questioning of students as they worked on authentic problem solving tasks.

In reporting on a constructivist approach to teaching computer graphics Taxén (2004) described the use of a final examination as causing a disconnect between the teaching approach and the assessment approach. The work indicated that while the teaching approach had focused on problem solving, and developing a deep understanding of associated graphics concepts, the use of an examination had rewarded students for adopting surface approaches to learning. Taxén (2004) indicated that it was quite possible for students to pass an exam through memorisation of facts, without developing the necessary depth of understanding.

When considered in terms of introductory programming, written examinations are seen to be an ineffective means of assessing student knowledge. A recent study by Sheard et al. (2013) interviewed programming unit lecturers from eight universities on the processes associated with setting final exams, the underlying pedagogical reasons, and how academics sought to prepare students. The work found that there was general agreement that a written examination was not ideal for assessing programming knowledge, though it was felt to be necessary to address issues of plagiarism and to assess conceptual understanding. Similar concerns were also echoed by Bennedsen & Caspersen (2006) who advocated for practical laboratory examinations.

The challenge, therefore, is to define an assessment approach that enables students to construct their knowledge, uses coursework assignments in a formative manner, and enables 100% of each student's grade to be determined after the end of the teaching period, with a strong alignment to unit intended learning outcomes.

4.1.2 Portfolio Assessment

In proposing Constructive Alignment, Biggs (1996) advocated strongly for the use of an assessment portfolio, and indicated that the principles of constructive alignment had evolved with the decision to use portfolio assessment. This work was extended in his later work, presented in (Biggs & Tang 1997), which outlined suggestions for implementing portfolio assessment and a generalised model for instruction design. Further advice and details of the generalised model were presented in Biggs & Tang (2007) book on quality learning at university.

Assessment involves three components, all of which are typically under the control of the teacher (Biggs & Tang 1997). These include *setting criteria*, *selecting evidence* and *making a judgement*. With the assessment portfolio the students take control of *at least* the selection of evidence, as illustrated in Figure 4.1. The portfolio is then a collection of work that the student puts forward for assessment against the unit's intended learning outcomes, which helps avoid the teacher-selected *sampling* effect of exams.

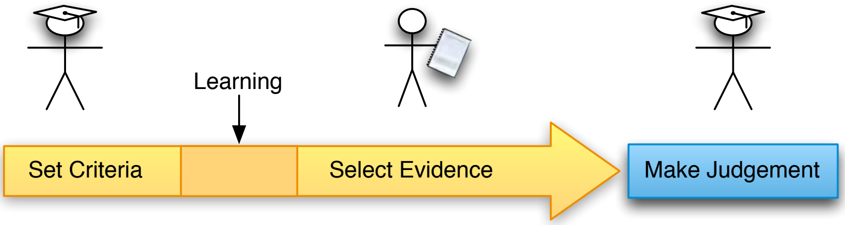


Figure 4.1: With portfolio assessment the student is responsible for, at least, the selection of evidence in the assessment process.

Smith & Tillema (2001, 2003) identified four different kinds of portfolios evident in the research literature: *dossier*, *reflective*, *training* and *personal development*. These reflect the combination of two identified factors: (a) the purpose of the portfolio as either for selection/promotion or learning, and (b) whether the portfolio is self-directed or mandated. The details of these are described in the following list, and shown in Figure 4.2.

Dossier [selection/promotion, mandated] is a portfolio for the purpose of selection or promotion that contains a mandated collection of work to demonstrate achievement.

Reflective portfolios [selection/promotion, self-directed] contain a self-selected collection of work that demonstrates growth or accomplishment for the purpose of admission or promotion. The portfolio is accompanied by a self-appraisal, with the justification for the selection of pieces being as important as the evidence itself.

Training portfolios [learning, mandated] are a mandated collection of work performed in a learning context. The portfolio has a fixed format, and contains representative work from the student demonstrating acquired skills, knowledge and competencies.

Personal development portfolios [learning, self-directed] are a self-selected collection of work, and reflective account of personal growth over an extended period.

Biggs' use of an *assessment portfolio* clearly fits with the *Training* portfolio classification, being a mandated part of the unit assessment for the purpose of evaluating learning outcomes. In the study of a small group of professionals, Smith & Tillema (2001) found the training portfolio to be highly rated. Their findings indicated that students found the training portfolio confusing initially, but that once they had understood its func-

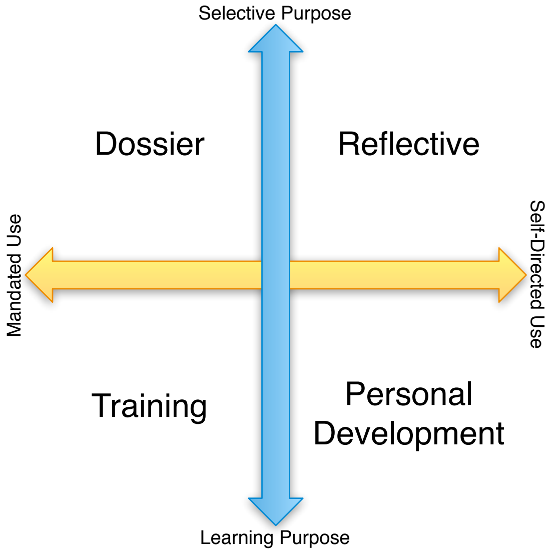


Figure 4.2: The four kinds of portfolio based upon purpose and use from Smith & Tillema (2001).

tion and rational they liked the approach, were easily able to construct their portfolio, and felt it was a fair way to assess their learning.

Tang et al. (1999) provides further evidence of the value of portfolio assessment. In evaluating how students approach study, they found that students tended to have a narrow, surface approach to studying for tests. These same students were found to adopt wider, more cognitively challenging, approaches when preparing for a portfolio assessment.

Portfolios have been used to assess introductory programming. (Plimmer 2000) reported the successful use of portfolio assessment in an introductory programming unit in which the portfolio contributed between 25% and 60% of the students' final grades. Programming portfolios were also discussed by (Jones 2010), where students submitted a number of portfolio assignments during the semester. These two approaches represent interesting applications of portfolio assessment, but do not use the portfolio as a means of performing a holistic assessment of the students ability to meet the intended learning outcomes, as is proposed in this work.

We argue that when used as a holistic assessment of student performances, portfolio assessment enables a shift from a Theory X "sage on the stage" view, to a Theory Y "guide by the side" view of education. The traditional approach of setting assignments and exams, in which educators test student ability, becomes inverted with portfolio assessment. Details of the assessment are no longer hidden, as is the case with exams, but is shared with students as the goal they need to achieve. Using portfolio assessment, the emphasis is on the students, and it is their responsibility to demonstrate how they have met a unit's intended learning outcomes. This frees educators to help students and to guide them in the preparation of their evidence. As illustrated in Figure 4.3, we are now working side by side with the students, helping them to achieve the unit's intended learning outcomes.

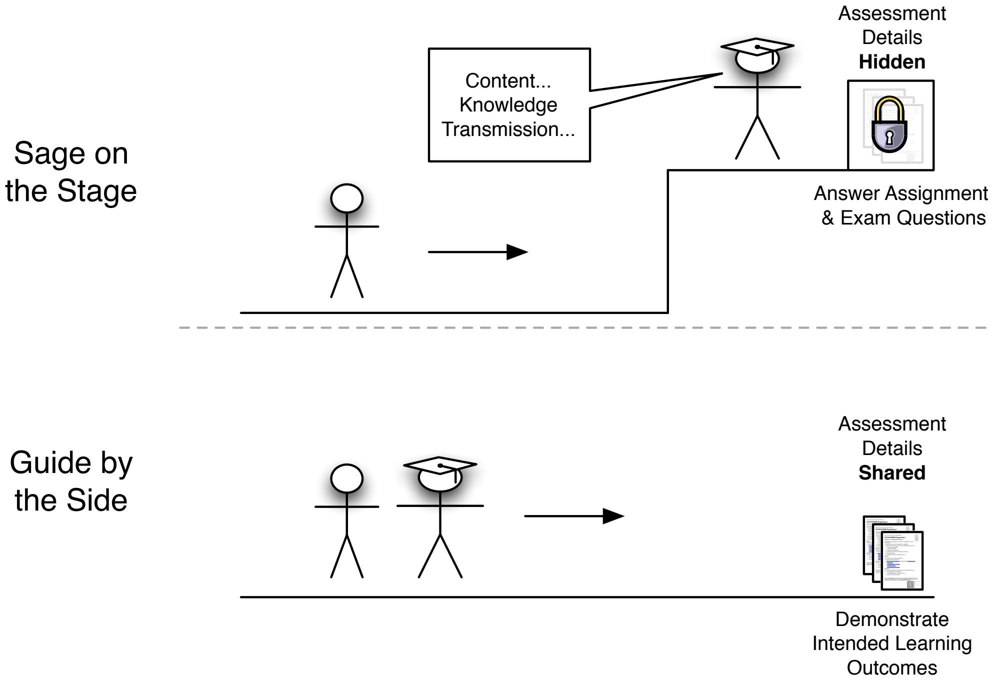

Sage on the Stage
Content... Knowledge Transmission...
Assessment Details **Hidden**
Answer Assignment & Exam Questions
Guide by the Side
Assessment Details **Shared**
Demonstrate Intended Learning Outcomes

Figure 4.3: Portfolio assessment helps enable the view of teaching staff as acting as a "guide by the side", rather than a "sage of the stage"

Portfolio assessment aligns well with all nine "*how*" principles listed in Chapter 3.

- The portfolio consists of a collection of work the student feels demonstrates the depth of their knowledge. (P1)
- Assessment criteria for the portfolio can be aligned with the unit's intended learning outcomes. (P2)
- The portfolio can be used as the sole form of summative assessment, with students being able to take advantage of formative feedback throughout the teaching period. (P3)
- A clear focus, and use of verbs from the relevant levels of the SOLO taxonomy, will ensure the portfolio requires students to engage appropriate cognitive levels, requiring them to explain, justify and reflect in cases where a depth of understanding is required. (P4)
- By communicating high expectations, students will strive to create high quality evidence for their portfolios. (P5)
- Students are able to develop evidence for their portfolio from day one; every-

thing they do could be of value. This will require active support from teaching staff. (P6)
- Without marks for motivation, an entirely portfolio assessed unit empowers students in the learning process, and they must be trusted that they are able to effectively manage their own learning. (P7)
- Portfolio assessment, with frequent formative feedback, is very much akin to agile software development processes. In addition, student portfolios provide a wealth of evidence for educators to guide change. (P8)
- Incorporating a reflective component in the portfolio encourages students to engage in reflective practice. (P9)

Given its role in the formation of constructive alignment, and its clear alignment with the principles from Chapter 3, the approach to constructive alignment for introductory programming presented in this chapter uses portfolio assessment.

4.1.3 Delivery Approach

The second aspect of the overall strategy is to define an approach for selecting unit content, with Principle 10 indicating that the overall strategy needs to be defined around a programming paradigm. Rather than focusing on the specific question of which programming paradigm (objects-first or objects-later) was chosen in this work, this section will demonstrate how the principles from Chapter 3 guide the design and delivery of teaching and learning activities. These guidelines apply equally to both approaches, and a range of other teaching and learning contexts. The specifics of the chosen paradigm for the example implementations is presented in Section 5.1 of Chapter 5.

A range of teaching and learning activities are appropriate for helping students develop an understanding of introductory programming. In the following sections we illustrate how the principles from Chapter 3 can be used to focus lecture away from knowledge transmission, and provide structured learning in laboratory sessions.

Focused Lecture Slides

Principle 11 and Principle 12 proposed a concept-based approach to teaching introductory programming content. These principles, along with Principle 4, guided the choice to use the "Beyond Bullet Points" style of presentation for lectures (Atkinson 2007). This approach draws upon the work of Richard Mayer (see Mayer (2005)), and structures presentations using a storyboard that guides the audience toward a stated “solution.” Using this structure helps to enable a shift in focus, from presentations as providing information to presentations as providing cognitive guidance, where the aim of the presentation is to guide the audience to build appropriate knowledge. An example of the storyboard template, outlining a presentation on arrays, is shown in Figure 4.4. This story guided the audience to the solution “Use arrays to store multiple values in a single variable.”

The first five slides, termed Act 1, that set up the “story”, telling the audience why they are there and centring them as the main character in the story. This helps motivate discussion, and focused teaching staff on clearly communicating the motivation behind the topic. The first five slides contain the following details:

Slide 1: **The setting**, sets the *context*, positing the story at a relevant place within the content of the unit.

Slide 2: **The protagonist** indicates the story is about the students, they are the focus not the teaching staff.

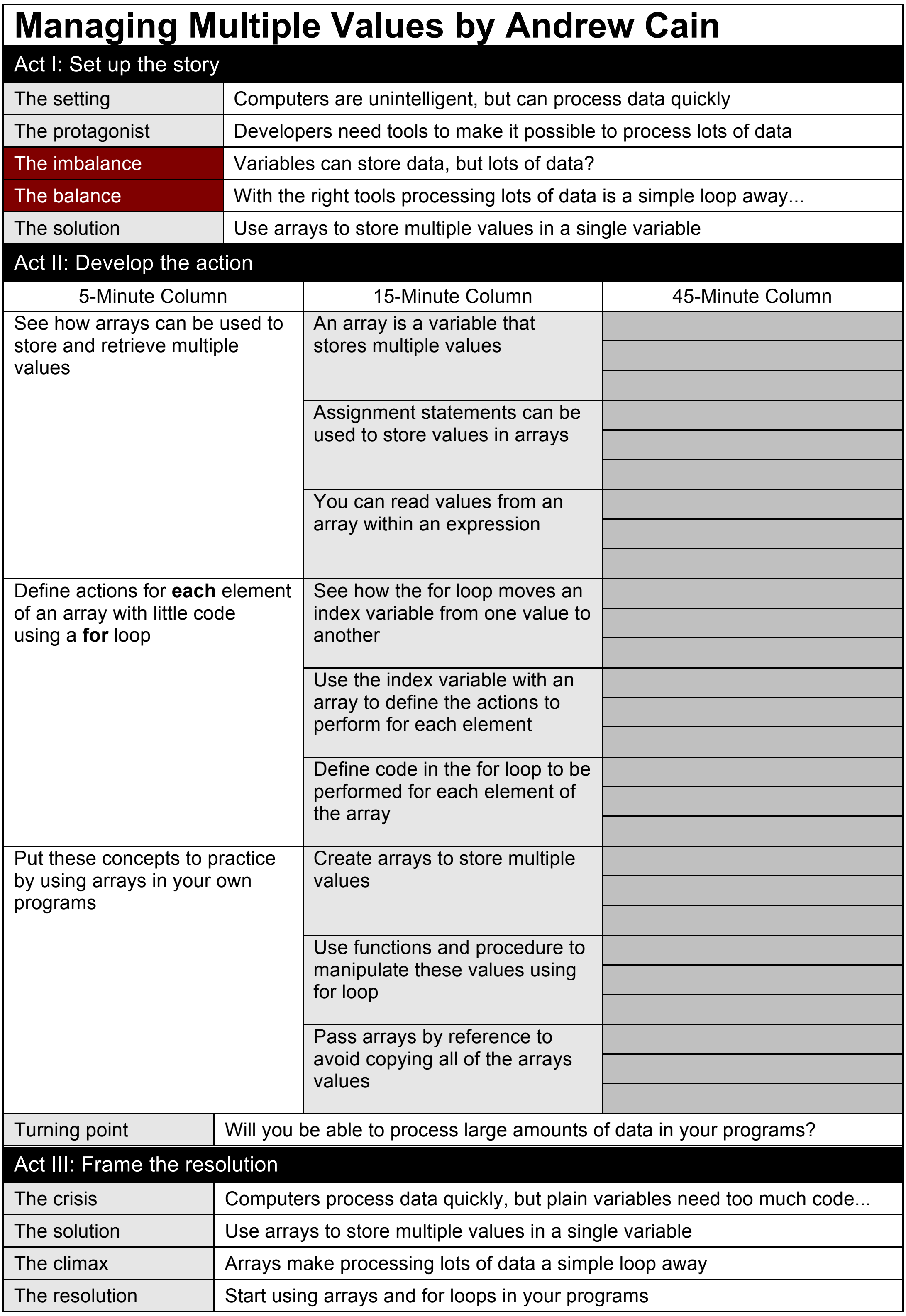
Slide 3: Indicates **the imbalance**, stating a current problem, challenge or opportunity that motivates the need to find a solution.

Slide 4: Juxtaposing the imbalance, **the balance** presents the goal, the situation in which the problem or challenge has been addressed, or the opportunity has been realised.

Slide 5: Presents our **solution**, indicating how students (the protagonist) can get from the current *imbalance* to the desired *balance*.

Slides contain a full sentence, written in an active voice. Ideally the slide text is accompanied by iconic visuals that set a theme or metaphor for the presentation. The time and resources necessary to prepare slides in this manner is, however, contrary to Principle 8, being agile and willing to change these teaching and learning activities. If significant effort is spent developing slides in this way it is likely to increase resistance to change by staff if adjustments to the topics message are desired. Instead, we suggest a minimalist approach with slides clearly showing the sentence text and using images to help communicate concepts central to the topic, as can be seen in the example slides shown in Figure 4.5.

For a fifteen minute presentation, the body of the presentation contained three main points, supported by three sub-points. Each addressed a “why” or “how” point, and supported the main solution. Once again these required a single short sentence in an active voice. The composition of these slides usually drew upon visuals created as part of the teaching and learning resources for the unit.



Managing Multiple Values by Andrew Cain

Act I: Set up the story	
The setting	Computers are unintelligent, but can process data quickly
The protagonist	Developers need tools to make it possible to process lots of data
The imbalance	Variables can store data, but lots of data?
The balance	With the right tools processing lots of data is a simple loop away...
The solution	Use arrays to store multiple values in a single variable

Act II: Develop the action

5-Minute Column	15-Minute Column	45-Minute Column
See how arrays can be used to store and retrieve multiple values	An array is a variable that stores multiple values	
	Assignment statements can be used to store values in arrays	
	You can read values from an array within an expression	
Define actions for **each** element of an array with little code using a **for** loop	See how the for loop moves an index variable from one value to another	
	Use the index variable with an array to define the actions to perform for each element	
	Define code in the for loop to be performed for each element of the array	
Put these concepts to practice by using arrays in your own programs	Create arrays to store multiple values	
	Use functions and procedure to manipulate these values using for loop	
	Pass arrays by reference to avoid copying all of the arrays values	

Turning point	Will you be able to process large amounts of data in your programs?

Act III: Frame the resolution	
The crisis	Computers process data quickly, but plain variables need too much code...
The solution	Use arrays to store multiple values in a single variable
The climax	Arrays make processing lots of data a simple loop away
The resolution	Start using arrays and for loops in your programs

Figure 4.4: An example of lecture material developed for one of the units using the storyboard template, provided by Atkinson (2007) and available from `http://beyondbulletpoints.com`, that outlines the stages in a Beyond Bullet Points presentation.

The last five slides conclude the presentation, reminding students of the overall solution and the new balance it brings about. These slides contain the following details:

1. The first slide in the conclusion was the *turning point*. This is a question that asks a question of the students, indicating the presentation has turned to the conclusion. In effect it asks them if the concepts presented have shown them how to address the imbalance from the introduction.
2. Next the *crisis* slide restates the balance and imbalance, reminding the students of where the presentation started and where it was aiming to go.
3. This then leads nicely to a restating of the *solution*. This is an exact duplicate of the solution from the introduction, but now the student have been though the "story" and it should have more meaning.
4. The *climax* brings all of the pieces of the story together in summary and give the teaching staff a final opportunity to motivate students to study the topic further themselves.
5. The final slide is the *resolution*, and indicates the end of the presentation.

Many of these guidelines from the Beyond Bullet Points approach were applied in the development of the lecture slides for the units discussed in Chapter 5. This included the general structure of the storyboard, though in some cases the three points were stretched to four but not beyond. The body of the presentations did include some information on "what" can be used, rather than purely focusing on "how" and "why." The use of active sentences, and visual communication, free from lists of bullet points, were also applied.

Focusing lecture slides in this manner helps to address the following principles from Chapter 3:

- Presentations aim to provide cognitive guidance, supporting students construction of knowledge. (P1)
- Completing the story board requires a focus on the most important aspects for each topic. Where this focus is appropriately targeted, this also helps support alignment with intended learning outcomes. (P2 and P4)
- Shifting details from presentation to other resources requires a trust in students willingness to learn, supporting the need for a Theory Y attitude to motivation. (P7)
- Using visuals for communicating core concepts, and minimalist themes for scaffolding slides, helps ensure presentations can be changed to respond to student needs. (P8)
- The presentation style encourages a focus on concepts as little (if any) syntax

> would be used in the slides themselves. Instead, slides focus on visual representations to communicate programming concepts, leaving lower level details to be communicated via other means, as is discussed in Chapter 6. (P11)

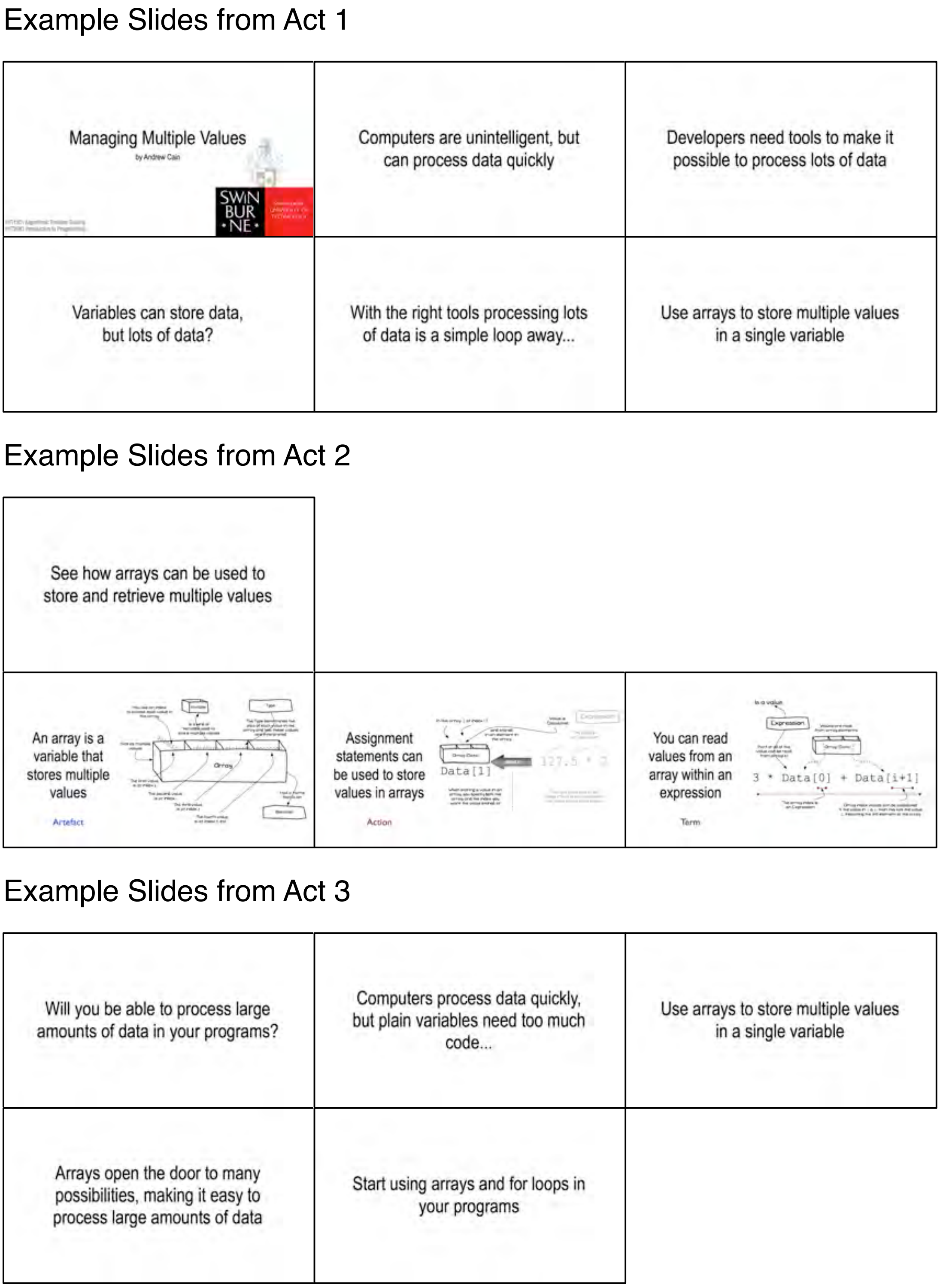
Interactive Lecture Demonstrations

Delivering one short, fifteen minute, presentation for each week's topic leaves a significant portion of a two hour lecture for other purposes. This time can be used to demonstrate the application of the concepts, providing students with a first experience of how these concepts can be used to create working programs. Similar approaches have been found to be an effective means of engaging students by Gaspar & Langevin (2007) and Rubin (2013), and help to implement the constructivist approaches to teaching introductory programming discussed by Ben-Ari (1998, 2001) and Van Gorp & Grissom (2001) which were reviewed in Section 2.1.2.

The interactive sessions involve writing programs from scratch, guiding students through the whole software development process. This involves questioning students about what we should do, and which concepts we needed to apply. Program's are developed together with the students, enabling discussion as they evolve. This enables a focus on the thought processes behind program creation, taking students through the decisions that need to be made in crafting the design of the program being creating.

One important strategy we applied in these interactive coding sessions was for staff to become selectively forgetful, typically forgetting the syntax related to the current concept, for example. In this way, staff had to make use of the resources available to the students to *find* the necessary details. For example, when focusing on how to declare variables this could be achieved by saying "We want to create a variable here, but I can not remember the syntax. Where could I look to find the syntax for this?" This triggers a discussion, and enables staff to demonstrate how to locate the associated resources, and lookup the relevant sections. Switching back to the code would be another point to "forget" the syntax, "So what do I type first?". The aim of this is to demonstrate to the students how these resources could be used to learn the language's syntax. In essence, providing an example of how to solve problems.

Depending on the topic, lectures typically follow one of two formats. Either the presentation was delivered in its entirety and was followed by the interactive session, or the interactive session was interwoven with the presentation itself. These two strategies can be applied at different times throughout the teaching period, depending on the topic.



Example Slides from Act 1

Managing Multiple Values

by Andrew Cain

Computers are unintelligent, but can process data quickly

Developers need tools to make it possible to process lots of data

Variables can store data, but lots of data?

With the right tools processing lots of data is a simple loop away...

Use arrays to store multiple values in a single variable

Example Slides from Act 2

See how arrays can be used to store and retrieve multiple values

An array is a variable that stores multiple values

Artefact

Assignment statements can be used to store values in arrays

Action

You can read values from an array within an expression

Term

Example Slides from Act 3

Will you be able to process large amounts of data in your programs?

Computers process data quickly, but plain variables need too much code...

Use arrays to store multiple values in a single variable

Arrays open the door to many possibilities, making it easy to process large amounts of data

Start using arrays and for loops in your programs

Figure 4.5: Sample slides from the lecture created from the template shown in Figure 4.4. Act 1 and 3 use minimalist sentences, with visual representations of programming concepts being used to communicate the important concepts.

Topics in the first part of a unit are likely to have the interactive sessions interwoven with presentations. At this stage each of the concepts was totally new to the students. Presentations focused entirely on concepts is likely to make the topic very abstract. By interweaving live coding demonstration with presentations we aim to address student apprehension about how concepts are realised in practice.

Later topics, which reinforce earlier concepts, are likely to benefit from progressing more quickly through the slides and using a longer interactive demonstration. This would have the benefit of allowing a wider range of concepts to be demonstrated in a single session.

The use of interactive lecture demonstrations helps to address the following principles from Chapter 3:

- These sessions help provide students with a demonstration of applying unit concepts, aiding them in the construction of their own knowledge. (P1)
- In introductory programming, many of the intended learning outcomes relate to applying programming concepts to the development of small programs. These demonstrations are, therefore, directly aligned with activities we expect students to be able to demonstrate by the end of the unit. (P2)
- Guiding students through the use of supporting resources helps keep the focus on the most important concepts, while also helping students learn how to use available resources. (P4, P6 and P11).
- Demonstrations also provide an opportunity to illustrate good programming practice, and to show students examples of programs they are expected to be able to create. (P5)
- Incorporating student input into the live demonstrations helps to motivate students, and requires a willingness to adapt to student requirements and needs. (P7 and P8)

Laboratory Sessions

Laboratory sessions provide an opportunity for students to try applying the concepts covered in the lecture classes themselves. To help structure these sessions we organised these tasks into three groups:

Laboratory tasks were designed to be a guided exercise that was to be completed in class, providing detailed instructions on how to approach the problems presented.

Core tasks had to be completed and submitted for feedback, helping students to develop pieces they can include in their portfolios.

Extension tasks provided students with optional exercises they could use to extend themselves, requiring greater levels of independence and a better understanding of the concepts.

Tutors guide students through each week's laboratory tasks. These tasks were designed to give students their first hands-on experience with each week's concepts. Laboratory notes provided detailed step-by-step instructions to enable students to work through these on their own if they wanted, or need, to. At the end of the laboratory exercises students should be sufficiently prepared to undertake the core tasks.

Students then apply their understanding of each weeks concepts to complete the core tasks. It is important that these tasks ask the students to perform actions related to the unit's intended learning outcomes, as these tasks will help them create evidence they can include in their portfolios. For example, this could include tasks such as creating one, or more, small programs, or performing code reading exercises that ask students to hand execute programs and explain the program's behaviour, or identify issues in the presented code.

Core tasks also form an integrated part of the formative feedback process. Once these tasks are completed students submit this work for feedback. Staff can then assess the work, and provide guidance on identified issues, and highlight possible misconceptions. If the work has issues, it can then be returned to the student with instructions on what needs to be corrected. Students can then work to address these issues, and associated misconceptions, and resubmit the work at a later stage. Where the task has been performed to a sufficient standard it can be signed off as complete, indicating that the student appears to have understood the associated concepts.

Extension tasks provide students with extra activities they can perform each week to demonstrate a deeper understanding of the associated concepts. These tasks are typically loosely defined, requiring students to explore the concepts in a more independent manner. A range of challenging tasks can be provided to support different student interests, and to provide students with ideas for activities that are likely to help them create pieces that will demonstrate valuable learning in their portfolios.

Laboratory sessions are highly student-centred, so organising these sessions in this way helps to address many of the principles from Chapter 3.

- Activities help students develop their knowledge of the concepts being covered. (P1)
- Staff help ensure that these tasks relate to the unit's intended learning outcomes, ensuring students will be able to demonstrate how they have met these outcomes when they submit their portfolios. (P2)
- Core tasks are used to provide students with formative feedback during the semester. (P3)
- Students are able to focus on the most important aspect for them at their current stage of development. Laboratory tasks focus on getting students started, core tasks focus on problems that demonstrate passable knowledge, while extension tasks can support the demonstration of more advanced levels of understanding. (P4)
- The formative process, with changes to core tasks being required before they are signed off, helps to clearly communicate the high standard expected of students. This is further supported by the list of extension tasks provided each week. These communicate the extra tasks the students *should* be completing to demonstrate a deeper understanding of the concepts. (P5)
- The different laboratory task levels each provide support for students at different stages of capability, while options in the extension tasks also help to support a range of student interests. (P6)
- Using this approach students take a greater responsibility for their own learning. (P7)

4.1.4 Summary

The overall strategy is defined by two approaches: the assessment approach, and the approach to content selection. Decisions related to these two approach were guided by the principles from Chapter 3, and resulted in the selection of a **portfolio assessment** approach to units that are taught using a range of student-centred teaching and learning activities. Figure 4.6 shows an updated version of Figure 3.9, showing the selected approaches discussed in this section. The next section describes the model for constructive alignment that developed from this overall strategy.

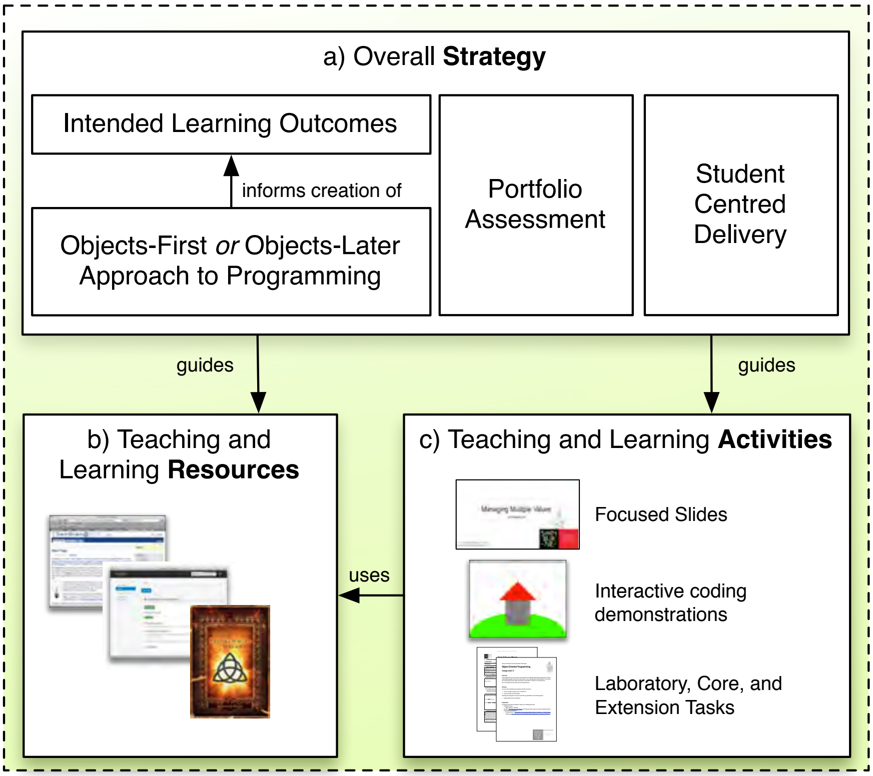

a) Overall **Strategy**
Intended Learning Outcomes
informs creation of
Objects-First *or* Objects-Later Approach to Programming
Portfolio Assessment
Student Centred Delivery
guides
guides
b) Teaching and Learning **Resources**
uses
c) Teaching and Learning **Activities**
Focused Slides
Interactive coding demonstrations
Laboratory, Core, and Extension Tasks

Figure 4.6: An updated version of Figure 3.9 showing the selected assessment approach, and the interactive teaching and learning activities.

4.2 Constructively Alignment with Portfolio Assessment

4.2.1 Model Overview

Having decided upon an overall strategy, the next stage of our research was to determine how Biggs' model of constructive alignment (Biggs 1996), and the details on using portfolio assessment suggested by Biggs & Tang (1997), could be used to guide the creation of an introductory programming unit. This involved the examination of the practical advice from Biggs & Tang (2007), which further elaborates on Biggs' model of constructive alignment and portfolio assessment. Using this together with the principles from Chapter 3, a model of constructive alignment for introductory programming was defined. The resulting model was documented in Cain & Woodward (2012), and captured staff and student processes and the artefacts generated and exchanged throughout the learning process, as illustrated in Figure 4.7.

Processes within the model are performed either by *students* or *teaching staff*. These processes are distributed across three stages of unit development and delivery: being either *prior* to the start of the teaching period, *during* the teaching period, or *after* the teaching period. Figure 4.7 illustrates this with separate columns for teaching and student processes, and rows for the stages in which these processes occur.

Prior to the start of the teaching period (a) the teaching staff *define intended learning outcomes* and *construct assessment criteria*. Together these aspects form a critical component of unit, defining what students will be able to achieve after successfully completing the unit, and how well they must perform in order to achieve different grade outcomes. Both the intended learning outcomes and assessment criteria are documented in the Unit Outline, a document that is a common practice in university environments. The Unit Outline forms the central focus for subsequent processes, and informs and guides staff in the *development of teaching and learning activities*. Unit outlines may also be used by students in evaluating units to select in their course of study.

The developed teaching and learning activities and their associated resources are then used during the teaching period (b) by staff to *deliver the unit*. Students follow the guidance of teaching staff, and ideally these activities aid students as they *construct knowledge*. The work that students produce can then be *submitted for formative feedback*, which provides an opportunity for teaching staff to *provide feedback and guidance*. The process of students undertaking activities, constructing knowledge, and receiving formative feedback is designed to be an ongoing iterative process throughout the teaching period.

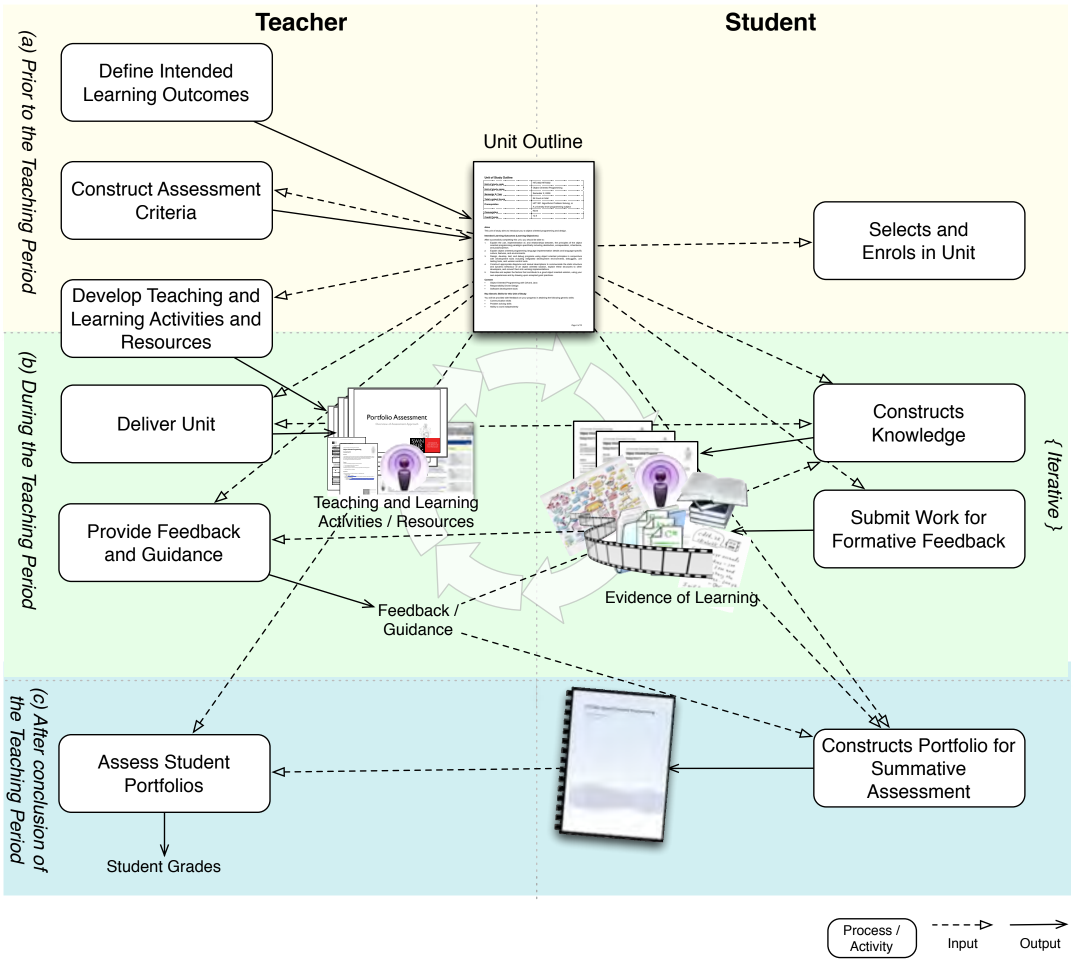

Teacher
Student
(a) Prior to the Teaching Period
Define Intended Learning Outcomes
Construct Assessment Criteria
Develop Teaching and Learning Activities and Resources
Unit Outline
Selects and Enrols in Unit
(b) During the Teaching Period
Deliver Unit
Portfolio Assessment
Teaching and Learning Activities / Resources
Constructs Knowledge
Provide Feedback and Guidance
Submit Work for Formative Feedback
{ Iterative }
Evidence of Learning
Feedback / Guidance
(c) After conclusion of the Teaching Period
Assess Student Portfolios
Constructs Portfolio for Summative Assessment
Student Grades
Process / Activity
Input
Output

Figure 4.7: An overview of teacher and students roles (columns), and iterative delivery, in the constructive alignment model developed for the introductory programming units.

After the conclusion of the teaching period (c) students prepare their work for summative assessment through the *construction and submission of their portfolios*. These portfolios are then *assessed by* the teaching staff against the intended learning outcomes and assessment criteria prepared prior to the unit's delivery.

Each of these processes is described in more detail in the following subsections.

4.2.2 Defining Intended Learning Outcomes

Intended learning outcomes are central to the concept of constructive alignment as a statement of what students will be able to achieve at the end of the unit. Aligned curriculum in constructive alignment indicates that teaching and learning activities and assessment must *align* to these intended learning outcomes. The findings in Chapter 2 indicate that alignment is typically performed by staff who indicate how teaching and learning activities and assessment tasks are aligned. Alignment is a matter external from the actual teaching itself. There is little involvement of the student in this process.

With portfolio assessment the alignment becomes intrinsically entwined with unit delivery and assessment. The teaching staff relinquish control of this aspect and the outcomes themselves take on their true purpose: as a statement of what students will be able to achieve at the end of the unit. All other aspects of the unit must now align to this purpose. Teaching and learning activities must prepare students to demonstrate that they have achieve these outcomes. Assessment aims to verify the extent to which students have reached these outcomes.

One way to conceptualise the central role of the intended learning outcomes is to picture this situation as a very long examination. The intended learning outcomes are the questions, the things students need to demonstrate they can do by the end of the "exam." The intended learning outcomes have become the assessment, a direct realisation of the fact that "assessment always drives the curriculum" (Ramsden 1992). In this arrangement there is little opportunity for misalignment between the unit objectives and assessment, but a greater importance on the exact nature of the intended learning outcomes. This critical role of the intended learning outcomes means that they are instrumental in the success of the unit.

For the introductory programming units it was important to design the intended learning outcomes so that, as a group, they cover the required programming competencies as well as the associated conceptual knowledge. The intended learning outcomes play a central role in driving the processes of both students and teaching staff. As a result, it is important they are expressed clearly and simply so as to be understood by all involved.

Development of unit outcomes has a variety of input sources. Thota & Whitfield (2010) propose inputs related to pedagogic theory (constructivism and phenomenography) as well as student factors such as approach to learning, learning styles, and prior knowledge. Armarego (2009) highlights the needs for inputs from industry, such as the Computer Science and Software Engineering Curriculum from professional standards bodies and associated Bodies of Knowledge Abran et al. (2001). For curriculum recommendations from professional standards bodies see Lethbridge et al. (2006), Cassel et al. (2008), and ACM/IEEE-CS Joint Task Force (2012).

In defining the intended learning outcomes for an introductory programming unit using constructive alignment with portfolio assessment we suggest drawing upon these sources, as well as the guiding principles from Chapter 3, overall strategy, resourcing factors, and accreditation requirements as shown in Figure 4.8. Resourcing factors provide additional constraints on what intended learning outcomes can include. Factors such as staffing, availability of texts, required tools, and others all need to be considered to ensure that students will be able to engage in activities associated with demonstrating the expected outcomes. Accreditation standards, such as the Australian Qualifications Framework (Australian Qualifications Framework Council 2013), require intended learning outcomes to demonstrate certain levels of achievement in order for degree programmes to gain recognition and funding. As these units will form an important part of programme objectives, these requirements must also be considered.

Biggs & Tang (2007) provided a number of recommendations for the development of intended learning outcomes. They indicated that it was appropriate for a unit to have between four and six intended learning outcomes, expressed at suitably high cognitive levels. Adopting this approach enables the clear focus on what is important for the unit, and encourages depth over breadth, Principle 4 from Chapter 3. A small number of intended learning outcomes keep the focus clear for students.

As discussed in Chapter 2, the SOLO Taxonomy proposed by Biggs & Collis (1982) provides a framework for ensuring intended learning outcomes aim for suitably high cognitive levels. Each of the identified cognitive levels has an associated list of verbs likely to elicit that level of activity. Table 4.1 lists selected verbs associated with the various levels of the SOLO taxonomy. These verbs can be used when defining the unit's intended learning outcomes. Biggs suggests that intended learning outcomes for university level units should aim for at least the multistructural level, with many units aiming for understanding at the relational level. In a review of science curricula in Danish universities, Brabrand & Dahl (2009) argued that the SOLO taxonomy provided a good tool for specifying competency with a range of areas showing progressing depth in terms of SOLO verbs from undergraduate to graduate education.

We developed the following guidelines that can be used to inform the formation of the intended learning outcomes for portfolio assessed programming units.

LO-1: Express outcomes using verbs at an appropriate level of understanding with reference to the SOLO taxonomy.

LO-2: Cover both the required *conceptual knowledge*, and *programming competencies*.

LO-3: Use *simple terms* (where possible) to communicate outcomes, thereby helping to ensure they can be understood by all students undertaking the unit.

LO-4: The number of outcomes should be *minimal*, ideally between four and six. This is to help ensure that each outcome covers a meaningful body of knowledge to a sufficient depth.

LO-5: Outcomes need to be *general* to facilitate assessment of diverse portfolios, and *sufficient* to ensure that differing degrees of proficiency and understanding can be assessed.

LO-6: There needs to be *flexibility* to enable students to choose a range of means when addressing outcomes.

Developing intended learning outcomes in this way helps to address the following principles from Chapter 3:

- Stated outcomes become the goal students work toward throughout the teaching period. Ensuring these are expressed using appropriate verbs from the SOLO taxonomy ensures they are likely to engage appropriate cognitive levels. (P1 and P2)
- Keeping the list of objectives short help ensure they are focused on the concepts related to the unit. (P4 and P11)
- Using verbs from the relational level of the SOLO taxonomy helps communicate staff expectations. (P5)
- Ensuring flexibility and clarity helps support a wider range of student interests and capabilities. (P6)

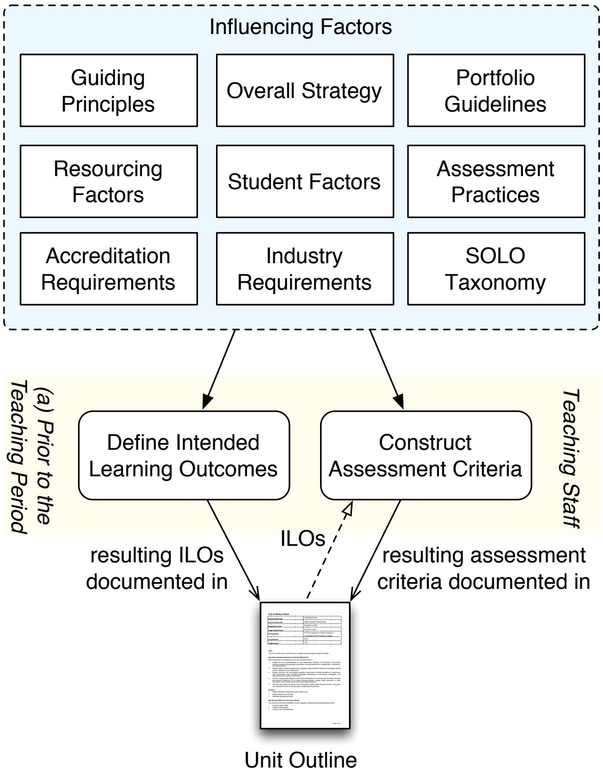

Influencing Factors
Guiding Principles
Overall Strategy
Portfolio Guidelines
Resourcing Factors
Student Factors
Assessment Practices
Accreditation Requirements
Industry Requirements
SOLO Taxonomy
(a) Prior to the Teaching Period
Teaching Staff
Define Intended Learning Outcomes
Construct Assessment Criteria
resulting ILOs documented in
ILOs
resulting assessment criteria documented in
Unit Outline

Figure 4.8: Factors that influence the defining of a unit's intended learning outcomes, and the construction of assessment criteria. These activities are undertaken by teaching staff prior to the start of the teaching period, see (a) from Figure 4.7.

Table 4.1: Selected list of verb related to the levels of the SOLO Taxonomy suitable for defining intended learning outcomes, adapted from Biggs & Tang (2007).

SOLO Level	Verbs likely to elicit indicated cognitive level
Unistructural	memorize, identify, recognise, count, define, draw, find, label, match, name, quote, recall, recite, order, tell, write, imitate
Multistructural	classify, describe, list, report, discuss, illustrate, select, narrate, compute, sequence, outline, separate
Relational	apply, integrate, analyse, explain, predict, conclude, summarise, review, argue, transfer, plan, characterise, compare, contrast, differentiate, organise, debate, make a case, construct, review and rewrite, examine, translate, paraphrase, explain causes
Extended Abstract	theorise, hypothesise, generalise, invent, originate, make original case

4.2.3 Constructing Assessment Criteria

Assessment criteria are developed alongside the definition of intended learning outcomes. The intended learning outcomes state what students need to demonstrate by the end of the unit, but these outcomes can be achieved to different standards. It is the role of the assessment criteria to state the required level of achievement students must demonstrate in order to be awarded various grade outcomes. This means that at the end of the teaching period students' portfolios can be assessed against the developed assessment criteria, but also that the assessment criteria can be used to guide the teaching and learning activities during delivery. Providing assessment criteria in the unit outline creates a simplified learning contract (Stephenson & Laycock 1993), in which students know "up front" what is required to achieve the different grades.

Principle 3 indicates that assessment should judge outcomes. To provide this holistic judgement the final summative assessment is *criterion-referenced*, as suggested by Biggs & Tang (1997). The criteria must, therefore, provide a means for teaching staff to assess submitted portfolios while also providing students with guidance they can use during the delivery and in the construction of their portfolios. Ensuring that the assessment criteria are stated clearly also helps ease students transition to this new form of assessment (Smith & Tillema 2001).

There is some contention regarding the specification of assessment criteria for assessing portfolios. For example, some consider that by overly specifying criteria students are limited in what can be included (see Driessen et al. (2005) and Tigelaar et al. (2007)). However, Smith & Tillema (2001) indicated that clearly communicating portfolio requirements helped ease students transition to this new, possibly unfamiliar, assessment approach. This is further supported by Allan (1996), who argued that clear communication of intended learning outcomes and assessment criteria enabled students to focus on developing knowledge required to succeed in a unit, and by Thorpe (2000) who noted that students found it easier to reflect on their learning if they were able to apply criteria defined by teaching staff.

The assessment criteria development process takes input from the intended learning outcomes, along with guidelines for portfolio assessment (Biggs & Tang 2007) and levels of achievement from the SOLO taxonomy (Biggs & Collis 1982) as shown in Figure 4.8. The resulting criteria are placed alongside the intended learning outcomes in the unit outline.

In order to facilitate assessment, and to guide student activity, the assessment criteria need to indicate clearly distinct requirements for each grade outcome. In the university where this work was carried out there are five grade categories: Fail, Pass, Credit, Distinction and High Distinction. The assessment criteria indicate what students need to demonstrate to achieve each grade.

The following list presents the assessment criteria developed in this work. The criteria are cumulative, with each level beyond pass requiring all previous requirements to be satisfied in addition to some deeper level of understanding being demonstrated. Pass requires that each intended learning outcome is met to a minimally acceptable standard, which will depend on the verb used in their description. Credit then requires an overall picture of the unit, with students starting to see how the various aspects of the unit come together as a whole. This is then required to achieve the Distinction grade, in which students must show they can apply unit concepts to the creation of a piece of work of their own invention. This does not need to be new or "ground breaking" work, just something the student created on their own that shows all of the intended learning outcomes in play. High Distinction then goes beyond this by asking students to engage is a small research project, encouraging them to work toward an extended abstract[1] level of understanding, but not requiring that they achieve this.

Fail is a result of anything less than Pass level.

Pass demonstrates *minimally acceptable* level of achievement. Students have been able to complete core tasks from the teaching and learning activities, and pass any hurdle[2] requirements.

Credit demonstrates all Pass requirements and shows a *good* depth of understanding across all intended learning outcomes, but does not go beyond presented work. Demonstrates at least a multistructural level of understanding of the unit overall.

Distinction demonstrates Credit level requirements and the ability to *apply* unit concepts to the creation of work of the students own invention. This demonstrates at least a relational level of understanding of the unit overall.

High Distinction demonstrates all Distinction level requirements and the ability to *research* a topic related to the unit. This still requires only a relational level of understanding, but provides opportunities and encouragement for students to explore beyond the current knowledge and work toward that extended abstract level of understanding.

[1] Extended abstract requires a level of understanding where new knowledge can be created.

[2] Hurdle requirements are anything that must be "passed" to pass the unit, but do not contribute marks toward the final grade.

The following guidelines were used to inform the definition and communication of the assessment criteria:

AC-1: Communicate assessment criteria using *simple terms* (similar to LO-3).

AC-2: Require sufficient progress to be demonstrated for all intended learning outcomes. At least a multistructural level of understanding should be obtained for passing students.

AC-3: Higher grades should require:

- evidence of *deeper learning*, while specifically avoiding an excessive volume of work.
- integrated understanding across related intended learning outcomes, as well as within each intended learning outcome.

AC-4: Aim to develop *clearly distinct* assessment criteria for each grade outcome, facilitating timely assessment and providing clear requirements for students.

AC-5: Clearly *map assessment criteria* to grade outcomes, ensuring students and staff have a shared understanding of how a portfolio relates to final grades.

Constructing the assessment criteria using these guidelines helps to address the following principles from Chapter 3:

- Requiring progressively higher levels of understanding for each grade classification help promote deep learning, and indicates staff expectations in terms of required levels of demonstration. (P1 and P5)
- Assessment criteria align to unit outcomes, with higher grades requiring students to demonstrate an understanding of the relationships between the intended learning outcomes and their associated concepts. (P2 and P11)
- Grades are awarded using criterion referenced assessment, assessing students' learning outcomes. (P3)
- Specific criteria help focus students on the most important aspects, with higher grades requiring demonstration of deeper learning. (P4)
- Simple terms help support a wide range of student language capabilities. (P6)
- Students can take responsibility for their learning, being able to aspire to achieve a given grade, and being able to apply their own imagination and interests in applying concepts to achieve higher grades. (P7)
- Evidence from student submissions provide a source of evidence for change, and can be reflected upon at the end of each teaching period to inform future changes. (P8 and P9)

4.2.4 Develop Teaching and Learning Activities and Resources

Having defined the intended learning outcome, and assessment criteria, teaching staff develop, or select, appropriate teaching and learning activities and resources. Figure 4.9 illustrates the role of this process in the overall unit delivery. The process uses inputs from the Unit Outline, and generates both teaching and learning activities and resources.

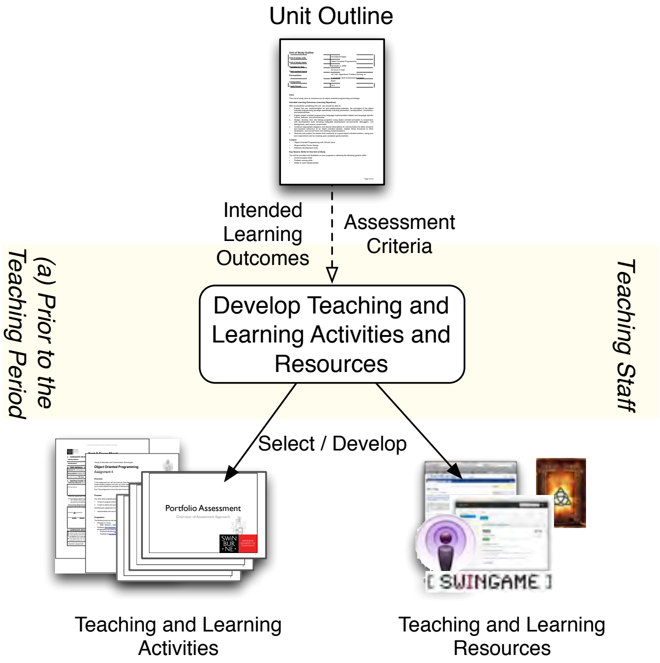

Unit Outline
Intended Learning Outcomes
Assessment Criteria
(a) Prior to the Teaching Period
Develop Teaching and Learning Activities and Resources
Teaching Staff
Select / Develop
Portfolio Assessment
SWINGAME
Teaching and Learning Activities
Teaching and Learning Resources

Figure 4.9: Development of teaching and learning activities and resources uses details from the unit outline to create/select appropriate resources and activities to ensure students engage appropriate activities during the teaching period.

The teaching and learning activities aim to elicit appropriate behaviour from students, ensuring that they engage in the cognitive processes representative of the desired level of achievement for each intended learning outcome. Output generated from this process includes lecture slides and tutorial/laboratory handouts. In relation to the overall strategy, these activities are likely to change frequently as teaching staff become better able to direct student efforts. In keeping with our agile principles (Principle 8) the effort spent on developing these teaching and learning activities should be minimised.

In contrast, teaching and learning resources provide students with detailed information they will require to successfully complete the teaching and learning activities. If designed appropriately, these resources should have a longer lasting value, and can change less frequently than the teaching and learning activities. Central to this approach is the idea of focusing (Principle 4) each aspect of the teaching and learning environment to best benefit the construction of student knowledge (Principle 1). Teaching and learning resources provide details, and extra attention and effort are required in their development, with the benefit of being able to be used in a range of contexts. Examples of resources include textual and visual illustrations, video podcasts showing example usage, online tools, and supportive software. Some of these resources can then be used in the creation of the lecture slides, but the lectures focus on providing cognitive guidance, directing students through the most important aspects with the details to be discovered later when students make use of the provided resources.

The following guidelines were used to inform this process:

- Teaching and learning **activities** should:
  - **TLA-1**: Actively engage the students – students must actively construction their own knowledge, as knowledge cannot be transferred by communication alone.
  - **TLA-2**: Align with the unit's intended learning outcomes – activities should relate to appropriately high cognitive levels, and students should be able to relate gained understandings to one or more of the unit's intended learning outcomes.
  - **TLA-3**: Focus on providing guidance – use communication to guide activity, but avoid attempting to provide all of the required details as these can be better provided in resources.
    - Lectures should inform, motivate and inspire students. Providing students with an overview of key concepts needed to get started with the tutorial/laboratory tasks.
    - Tutorial/Laboratory tasks should direct students to perform activities that engage appropriate cognitive levels, helping them create artefacts that can be included in their portfolios.
- Teaching and learning **resources** should:
  - **TLR-1**: Provide the details students require to perform the tasks from the teaching and learning activities.
  - **TLR-2**: Be created with a focus on re-usability.
  - **TLR-3**: Support a range of different learning styles.

Material developed using these guidelines address the following principles from Chapter 3:

- Aligning activities to intended learning outcomes ensures students develop appropriate knowledge related to the units concepts. (P1, P2, and P11)
- Focusing these activities on appropriate cognitive levels helps direct student attention, and communicate staff expectations. (P4 and P5)
- Separating activities from resources helps ensure activities can change to better meet student needs. (P8)

See Section 4.1.3 for some example activities developed using these guidelines.

4.2.5 Iteratively Deliver Unit and Provide Feedback

Constructive learning theories emphasise the active role of the learner in constructing knowledge. Our guiding principles are centred on this notion and adopt Biggs' pragmatic view of constructivism. The iterative, students centred, delivery process aims to embody Biggs' quote "It's what the student does that counts." (Biggs 1996), a statement that can be traced back to Tyler's quote, "It is what he does that he learns, not what the teacher does" (Tyler 1969).

Existing work on constructive approaches to teaching introductory programming provide some advice on designing and delivering student-centred teaching and learning activities. Ben-Ari (1998, 2001) discussed the need for students to construct appropriate models of the computer. Van Gorp & Grissom (2001) described collaborative and constructive environments with the use of code walk-throughs, writing code, debugging and other activities. Thramboulidis (2003*a*) presented a design-first approach to object oriented programming that focused on engaging students with object oriented design processes. Wulf (2005) reported strategies such as moving content from lectures to online video presentations. Similarly, the work of Thota & Whitfield (2010) also presented constructive approaches to teaching introductory programming that focused on group work, and the active role of the student.

Figure 4.10 illustrates the iterative delivery process at the heart of the model presented. The process centres on the students construction of knowledge, which draws up the teaching and learning activities and resources. This active process of learning generates pieces of work that the student submits for formative feedback. The teaching staff evaluate the evidence presented, trying to identify issues with the students current mental models, and provide formative feedback to the student. This feedback

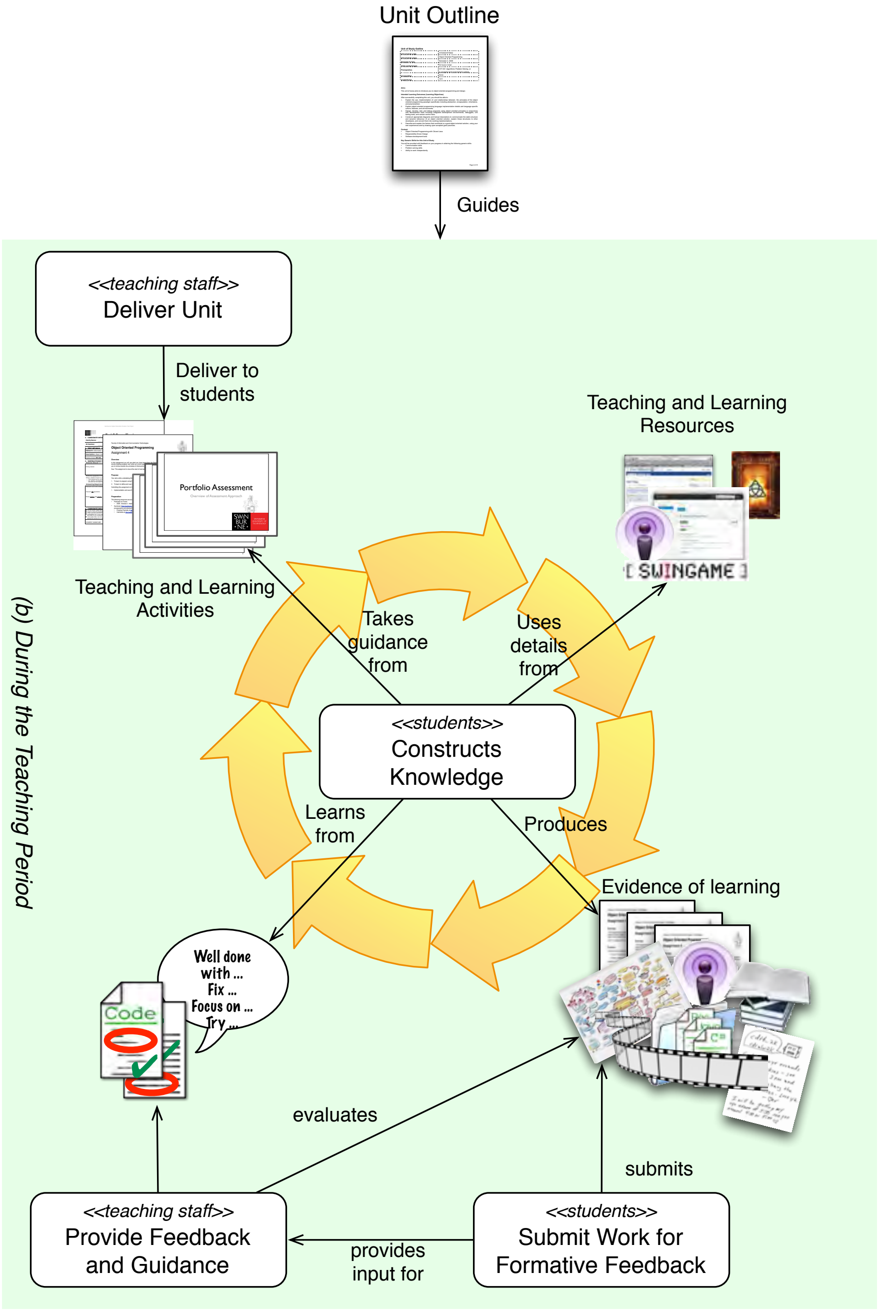


Figure 4.10: Iterative nature of the unit delivery process

helps inform the student of their progress, and provides them with tasks they can act up thereby ensuring we close the loop.

All of these processes occur on a weekly basis, with rapid iteration ensuring feedback is timely and gives students the best chance of addressing misconceptions before the end of the teaching period. Each week, students will:

1. Attend lectures which guide them in relation to key aspects and motivations.
2. Undertake set tasks from tutorial handouts, while drawing upon teaching and learning resources for required details.
3. Produce work and submit for feedback.
4. Receive feedback aimed to help them improve, and a list of changes required to meet the expected standard.

For any one topic, this iterative process may take a couple of iterations before the student is successful in getting the work signed off. Figure 4.11 is an illustration shown to students to indicate the iterative nature of the submission process. The highly connected nature of the topics in introductory programming means that it is critical students understand earlier topics before they move on. Work that is submitted is not considered complete by teaching staff until it demonstrates certain levels of understanding. While this is not the case, students need to correct and resubmit the work.

Requiring students to resubmit work ensures that each topic is completed in its entirety. This focus on quality, and depth of understanding, helps communicate the high standards expected of students, Principle 5. Students are expected to submit some work for assessment each week. To enable fast turn around, this work is required in hard copy at the start of each week's lecture. This work is then evaluated by the teaching staff *before* that week's tutorial classes (in practice 1 to 2 days). In the tutorial classes the work is returned to students, and the teaching staff briefly discuss progress with each student and the aspects of their work they can improve upon. This dialogue focuses on the student's individual understanding, and their demonstration thereof.

The following guidelines were developed and used to inform the planning and delivery of teaching and learning activities:

IDU-1: Provide opportunities through activity design to actively engage students – *it is what the student does that counts*.

IDU-2: Relate all activities to the objectives, providing students with opportunities to create evidence for their portfolio.

IDU-3: Use ungraded formative feedback to aid knowledge construction, with pref-

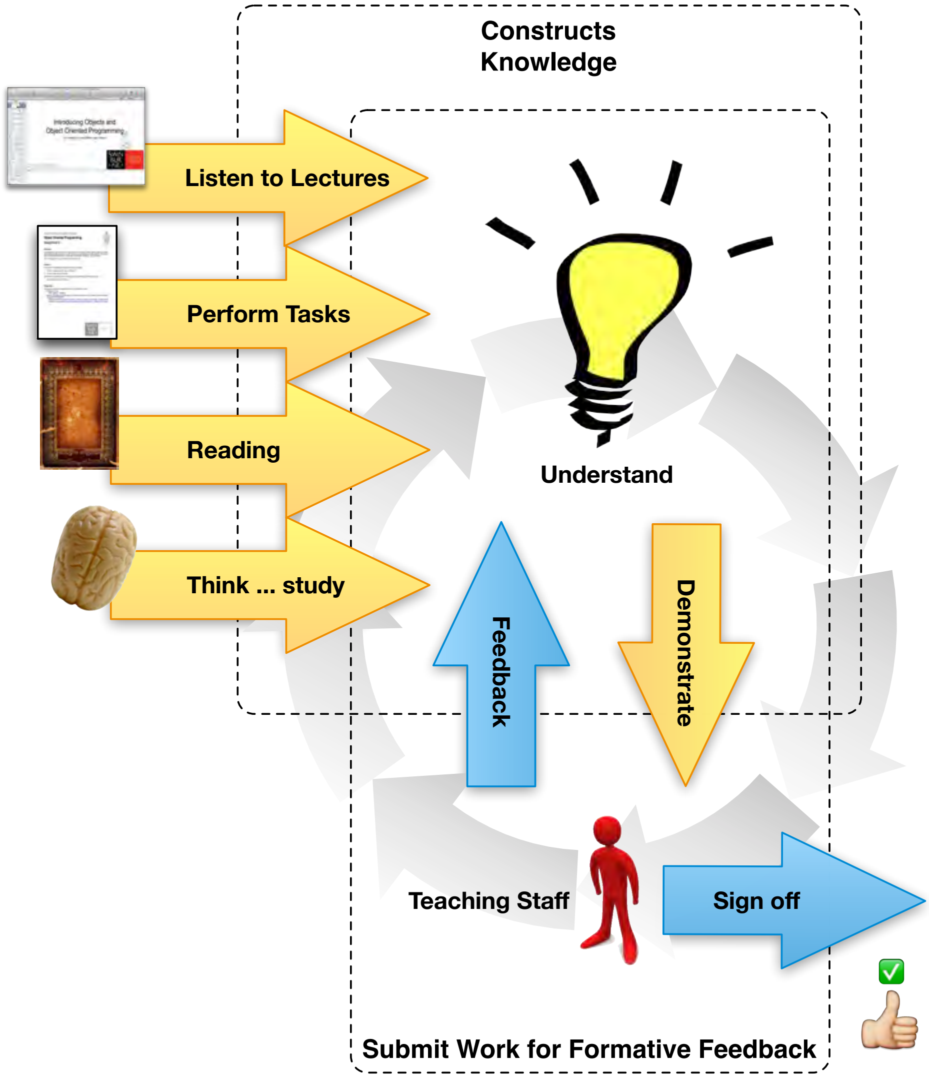


Figure 4.11: Iterative process students undertake to get work signed off.

erence for small, frequent guidance.

IDU-4: Actively support students both during and outside of scheduled class times.

Adopting these guidelines in the delivery of a unit will help address the following principles from Chapter 3:

- Frequent formative feedback aids students' construction of knowledge, and helps ensure learning aligns with unit outcomes and concepts. (P1, P2, P3 and P11)
- Feedback can focus on the most important aspects, ensuring it is relevant to each student and their current level of understanding. (P4 and P8)
- Requiring work to be completed to a good standard helps reinforce staff expectations, and supports students by encouraging reflection and deep approaches to learning. (P5, P6, and P9)
- Using formative feedback, without the marks associated with summative assessment, requires a Theory Y attitude to student motivation. (P7)
- Aiding students during and outside of scheduled class times directly supports their learning. (P6)

4.2.6 Construction, Submission, and Assessment of Portfolios

The final phase of the process is the development and submission of portfolios by students, and assessment by staff. This process uses the intended learning outcomes and assessment criteria from the Unit Outline to determine what needs to be demonstrated and assessed. Figure 4.12 illustrates the processes for portfolio construction by students, and assessment by staff. Students construct their portfolios from work completed during the teaching period. This work can incorporate feedback received, enabling students to showing off their best work and providing them with encouragement to act upon the feedback. Figure 4.13 shows an illustration used to explain the portfolio construction process to students.

In preparing the portfolio, students must demonstrate that they have met all of the unit's intended learning outcomes. This alignment is documented by students in a *Learning Summary Report*. The Learning Summary Report starts with a self assessment, in which the student indicates which grade they are *applying* for with this portfolio. In the following sections students provide justification for why they should be awarded this grade. Students are required to list the pieces of work they have includes, and then explain how these pieces demonstrate that the student has attained all intended learning outcomes. The report ends with a reflection in which the student is encouraged to reflect upon the significance of what they have learnt, as well as on the process of learning itself. This learning summary report is then combined with the other pieces of work, printed, bound and submitted for assessment.

The submission process differs based upon the grade students are aiming for in their portfolios. Students aiming for a Pass or Credit grade submit their portfolio by a set date in the examination period. These portfolios contain primarily work set out in the teaching and learning activities ("core" tasks), and will have been checked already by teaching staff throughout the teaching period as part of the formative feedback process. Students aiming for Distinction or High Distinction are required to present their portfolio at an interview. In this interview students outline their custom work, and discuss how this relates to the unit's intended learning outcomes. The interviews are conducted in an open, relaxed, and friendly manner, and students are encouraged to elaborate on what they have achieved.

Based on our experience we suggest the following guidelines be used to inform the creation and assessment of portfolios:

AP-1: Encourage unique, diverse, concise, and strongly aligned evidence.

AP-2: Motivate students to include evidence of learning from formative experience.

AP-3: Require students to reflect on their learning, and the evidence in their final portfolio, with respect to the intended learning outcomes of the unit and the assessment criteria.

AP-4: Use an interview, or hurdle test(s), to check for minimal pass criteria in an invigilated manner. Where tests are used they need only distinguish between Pass and Fail, and do not need to address higher grades.

AP-5: Accurately and consistently follow the terms of the assessment criteria, as this is the *contract* the students work towards.

Applying these guidelines to the creation and assessment of portfolios helps address the following principles from Chapter 3:

- Students are actively encouraged to include pieces that demonstrate their learning. (P1)
- Pieces included in the portfolio must align with the unit's intended learning outcomes. (P2)
- Feedback from the formative feedback process can be acted upon by students, with improved versions of earlier work being included as pieces in their final

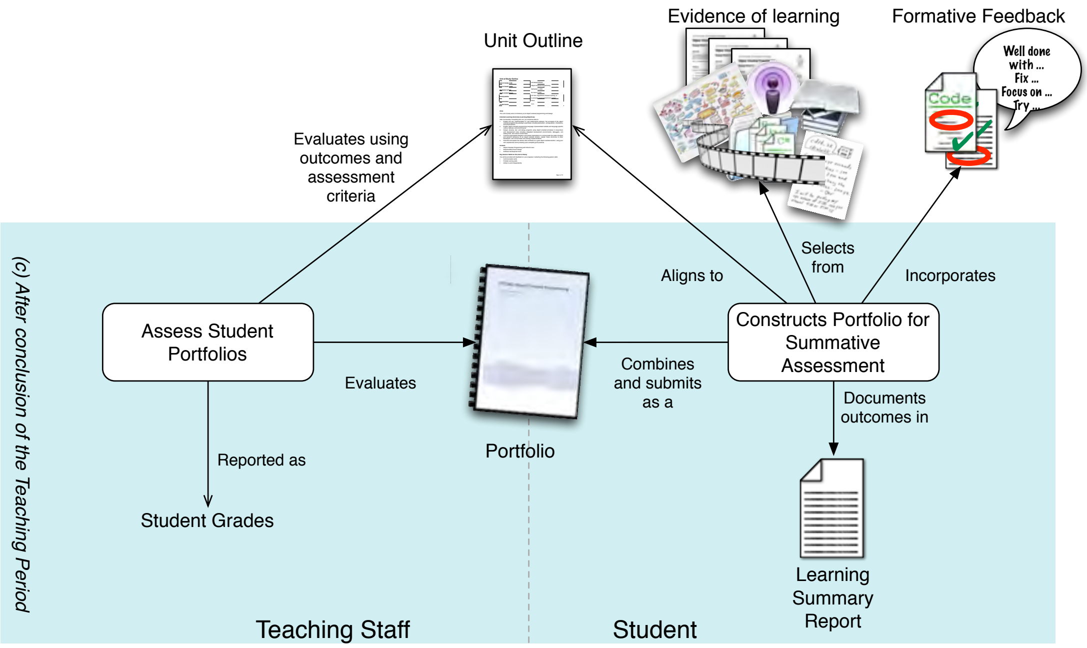


Figure 4.12: Processes of constructing, submitting, and assessment portfolios.

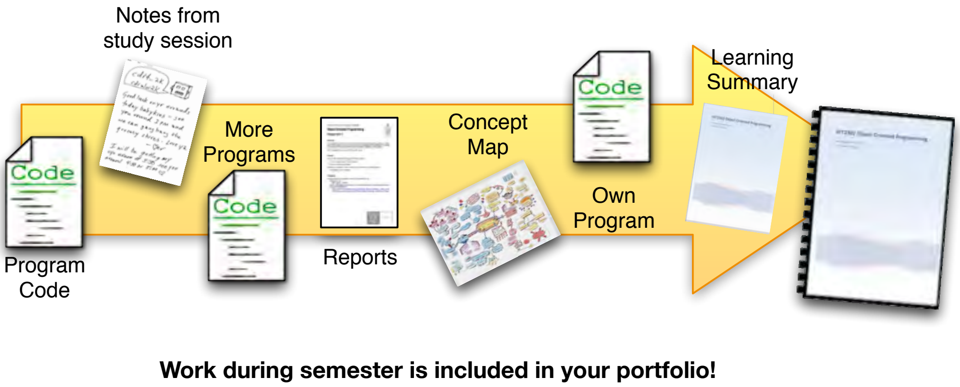


Figure 4.13: Illustration shown to students to highlight the process of constructing their portfolio during the teaching period

portfolios. (P3)
- In writing the Learning Summary Report, students need to address the assessment criteria that capture staff expectations for each grade outcome. (P5 and P7)
- Students are encouraged to reflect on their learning experience, and to document these reflections in the Learning Summary Report. (P9)

4.2.7 Addressing Plagiarism

Sheard et al. (2003) reported two studies on student attitudes and behaviours associated with cheating, and work practices of students in the Information Technology domain. This work identified widespread cheating within the Information Technology discipline, with the most common forms of cheating being associated with assignments and class tasks. This is particularly concerning, as the approach presented in this chapter focuses heavily on similar tasks at the core of its assessment strategy.

A subsequent follow up study (Sheard & Dick 2011) outlined a number of strategies that had been applied to address plagiarism, and indicated that frequency, and acceptability, of cheating had decreased over the ten years between the studies. The strategies implemented included adjustments to plagiarism policies to simplify its implementation, strategies to raise student awareness of plagiarism, as well as tools and practices to help identify and reduce cheating. These positive findings indicate the potential to address plagiarism issues through procedural changes.

The work of Sheard et al. (2003) also identified reasons students cheat, and reasons for not cheating. The top reasons for cheating included issues associated with time pressures, fear of failure, and difficulty of tasks. In contrast, reasons for not cheating related to personal factors associated with students taking responsibility for their learning.

While embodying a predominantly Theory Y atmosphere (Principle 7), the model presented in this chapter aims to minimise plagiarism by reducing reasons for cheating, while encouraging students to take greater ownership of their learning. This is achieved through the following mechanisms.

- Formative assessment does not punish students for misunderstandings. Rather, it actively encourages student to highlight their issues so that teaching staff can provide them with valuable feedback.
- Weekly interactions between students and teaching staff provide an opportunity

to verify understanding of the submitted work. For work to be signed off by the teaching staff, students need to be able to discuss the work with their tutors in class.

- A number of hurdle tests were included in the teaching and learning activities, the last of which had to be passed under examination conditions.
- Higher grades required an interview in which students elaborated on their work. This required the discussion of specific details that would be hard to fabricate.

All of these aspects are primarily included for other purposes, with the exception of the hurdle tests which are provided primarily as a means of validation for students having met minimum requirements. Unlike standard examinations, these tests aim only to assess *core* aspects of the unit that all students should be able to perform without issues. Figure 4.14 shows an illustration used to describe the tests to the students.

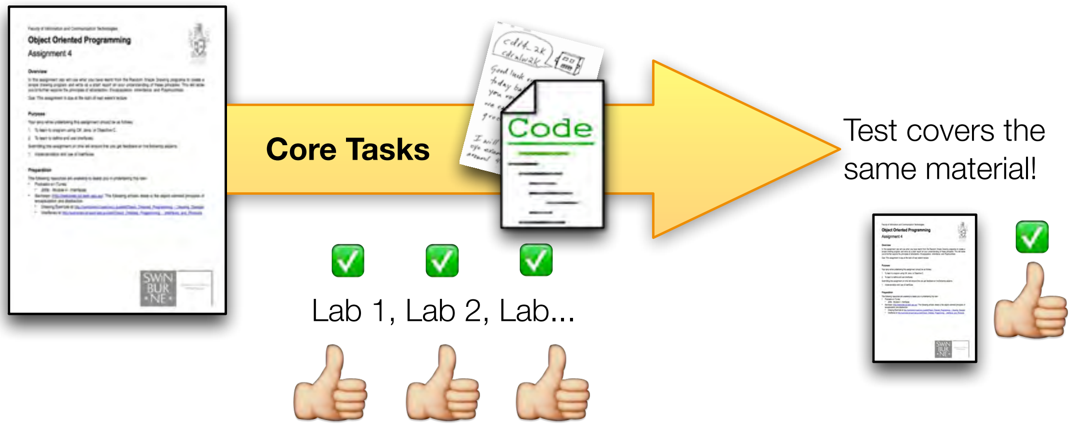


Figure 4.14: Tests cover aspects already presented in the tests, helping verify students completed the work themselves.

As the tests only assess core competencies, the tests are marked to an exacting standard. Students can be awarded one of three grades: pass, fix or redo.

- The *pass* grade requires the large majority of the test to be correct. Small issues like minor syntax errors, or other small mistakes can be overlooked, but as a majority the work must demonstrate good mastery of the topics covered.
- *Fix* grades indicate some larger issues are present, but nothing critical and the work still demonstrates a sufficient mastery of the content.
- Where the test indicates larger issues that represent critical misunderstandings for the student, their test is marked as *redo* and they must resit the test to be eligible to pass the unit. When no additional test opportunities are available this grade would indicate a *Fail* result for the unit.

With both pass and fix grades, students are expected to correct all issues in their test and include the corrected versions in their portfolios.

It is important to note that the *redo* grade is awarded where critical misunderstandings are demonstrated. This is not equivalent to getting less than 50%, or some other arbitrary percentage, of available marks. The work is assessed qualitatively, with teaching staff making expert judgements about the level of understanding being demonstrated. The response to even a single question could demonstrate critical misunderstandings, though typically this knowledge would be tested across a number of questions.

Students must include the tests in their portfolios, and the last test has to be passed in examination conditions. All tests also perform a formative role, with students needing to correct any issues themselves and resubmit the work, with the test only being signed off when students have been able to address it to the required standard.

These mechanisms help to address the main reasons that students cheat: time pressures, fear of failure, and difficulty of tasks (Sheard et al. 2003). The iterative nature of the formative feedback process, with the ability to resubmit work without penalty, helps to address all of these reasons. Time pressures are reduced with the summative assessment being delayed until the end of the teaching period. Frequent formative feedback, without penalties for misunderstandings, also helps to alleviate some of the issues associated with the fear of failure.

At the same time, the portfolio assessment puts the emphasis on the student to take responsibility for their own learning. This helps encourage aspects associated with reasons not to cheat as indicated by Sheard et al. (2003). Custom work required for higher grades requires students to invest something of their own interest in the work they submit, helping build student pride in their own work. At the same time, the formative feedback process provides students with opportunities to correct their own work without needing to resort to cheating to get a good mark.

Through a combination of mechanisms to reduce the reasons for cheating, and encouraging aspects associated with taking responsibility for their own learning, the model addresses issues of plagiarism without unduly emphasising a punitive approach. This is in keeping with Principle 7 from Chapter 3.

4.3 Summary

This chapter has outlined an overall strategy for teaching introductory programming with portfolio assessment based on an objects-later approach. Using the principles from Chapter 3, a model for the development of constructively aligned units was presented. Chapter 5 continues this work by demonstrating the application of this model in the creation and delivery of two introductory programming units.

5

Applying Constructive Alignment and Portfolio Assessment for Introductory Programming

Chapter 4 proposed a model for delivering introductory programming units based upon the principles from Chapter 3. The proposed model uses portfolio assessment, with a concept-based delivery that focuses on the students active construction of knowledge. This chapter provides example implementations of this model, demonstrating how the principles from Chapter 3 and the model from Chapter 4 can be realised in a programming curriculum.

Section 5.1 provides the final piece of the overall strategy for delivering introductory programming, describing the choice of programming paradigm for the first programming units. The following two sections, Section 5.2 and Section 5.3, then describe two programming units implemented using the model from Chapter 4. For each of these exemplar programming units the subsections are ordered to follow the processes from Chapter 4. First we outline the definition of the intended learning outcomes and the construction of the assessment criteria. This is followed by examples of various teaching and learning activities and resources developed and delivered as part of this curriculum. Finally each section concludes with an overview of how student portfolios were assessed.

5.1 Paradigm Choice

In describing Principle 8, Section 3.1.9 of Chapter 3 outlined a overall strategy that could be used to guide the development of the units. This strategy included the separation of material for teaching and learning activities from more detail focused resources, but also included defining an overall strategy to guide the development of this material. Within the overall strategy was the requirement to make decisions related to assessment and delivery approach that were subsequently discussed in Section 4.1 of Chapter 4, leaving only decisions related to choosing an approach to selecting content to be discussed.

Principle 10 from Chapter 3 indicated that, with programming, the choice of which content to include is heavily influenced by programming paradigm. As discussed in Section 3.2.1, a number of programming paradigms could be used to teach introductory programming. This section addresses the question of which programming paradigms were selected, and why, for the two introductory programming unit examples in this chapter.

Prior to conducting this research we have had experience with teaching introductory programming using both imperative-first and objects-first approaches. Our view mirrors those of Rist (1996) who reported on plans and cognitive schemas, the fundamental units of program design. In relating plans to objects, Rist (1996) indicated that objects were not different, they were more, as objects require additional overhead related to defining object[1] structures. Given this, units that take an objects-first approach will still need to have a significant focus on procedural aspects, as indicated by Robins et al. (2003), a reasoning that was also echoed in the "back to basics" approach of Reges (2006). Table 5.1 lists the main concepts programming concepts related to procedural and object oriented programming to illustrate this point. Conceptually objects represent a combination of structured data and associated functionality, so from this perspective objects build upon procedural programming concepts.

As a result, an objects-later approach was taken with the units reported in this work. However, we believe that the model discussed in Chapter 4 would also be appropriate for units developed using an objects-first approach – see Chapter 8 for further discussion on this point.

In covering structured procedural programming, the focus was on procedural programming concepts such as control flow, functions and procedures, parameter pass-

[1] Object structures are typically defined using classes or similar mechanisms in languages such as Java

Table 5.1: Illustration of the programming concepts related to procedural and object oriented programming

Concept	**Procedural Programming**	**Object Oriented Progamming**
Calling procedures to perform actions	✓	✓
Variables to store values	✓	✓
Parameters to pass values to procedural abstractions	✓	✓
Functions to calculate values	✓	✓
Code as a sequence of action statements	✓	✓
Selecting using if and case statements	✓	✓
Repetition using for, while, and repeat loops	✓	✓
Arrays to store multiple values	✓	✓
Iteration over array contents to process data	✓	✓
Structures to record multiple field values in a variable	✓	✓
Pointers to refer to other values	✓	✓
Classes to combine templates for object creation		✓
Methods called upon objects		✓
Inheritance of behaviour from parent classes		✓
Abstract class members		✓
Pure abstract interface definitions		✓
Subtype polymorphism		✓

ing, and data modelling using structures and records. Maintaining a clear focus on these concepts helps to address Principle 4 and Principle 11 from Chapter 3.

Reges (2006) back to basics approach also aimed to teach imperative programming concepts, which they did using the Java programming language. However, Java is an object oriented programming language and so, in effect, this approach taught students how *not* to use Java. While we have adopted the imperative programming focus, Principle 12 indicates that we must select a programming language that was designed for this purpose. The discussion of which language was used for the example units is presented in the following sections.

While objects did not appear in the first programming unit, their importance in students' education remained a focus. Rather than seeing programming as being delivered in a single stand-alone unit, we designed a sequence of two units that worked closely together. The first covered structured procedural programming, focusing on aspects such as control flow. The second focused on object oriented programming, which can then use a model driven approach similar to the one reported in Bennedsen & Caspersen (2004), but without having to cover procedural programming aspects.

Figure 5.1 shows the final overall strategy for the example units presented in this thesis. Structured procedural programming principles were used to inform the creation of the intended learning outcomes for the introductory programming unit. These outcomes then become prerequisite knowledge for the object oriented programming unit,

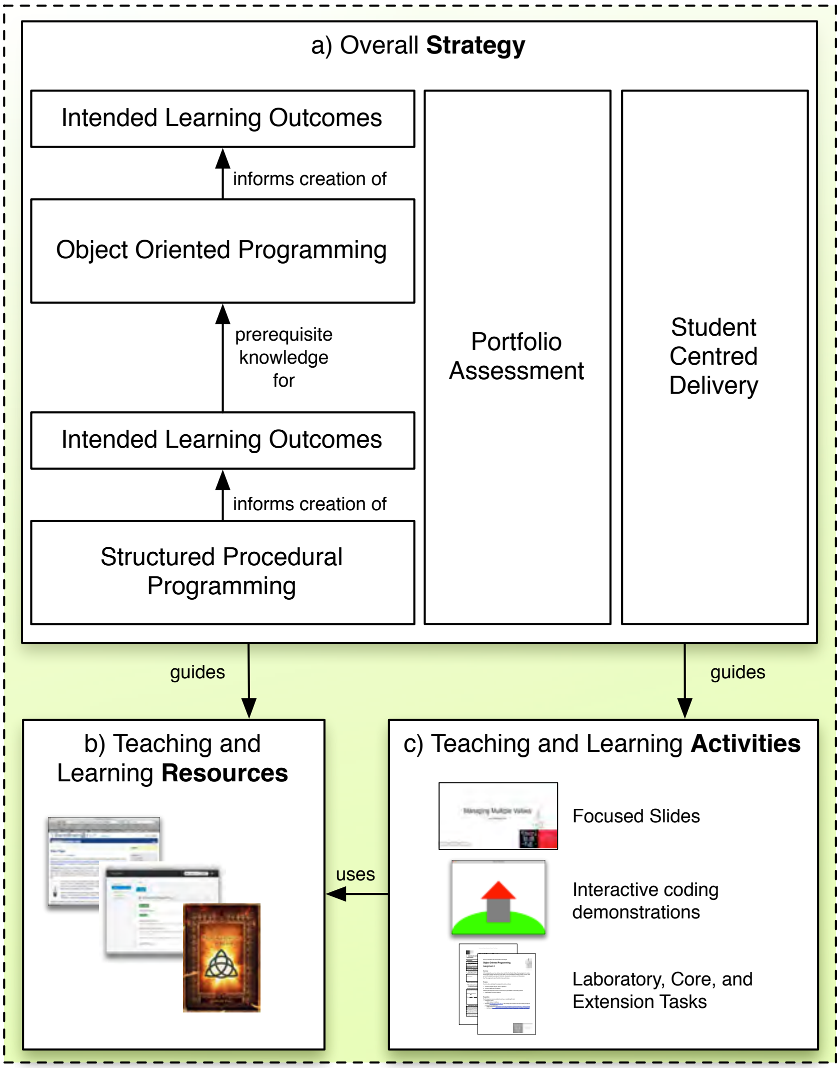

a) Overall **Strategy**
Intended Learning Outcomes
informs creation of
Object Oriented Programming
prerequisite knowledge for
Intended Learning Outcomes
informs creation of
Structured Procedural Programming
Portfolio Assessment
Student Centred Delivery
guides
guides
b) Teaching and Learning **Resources**
uses
c) Teaching and Learning **Activities**
Focused Slides
Interactive coding demonstrations
Laboratory, Core, and Extension Tasks

Figure 5.1: An updated version of Figure 3.9 showing the programming paradigms that will form the approach for selecting content

which focused on object oriented programming principles. Both units used portfolio assessment, and focused on active student centred approaches to introducing students to unit content.

The objects-later approach taken by these example units aligns well with the principles from Chapter 3, providing a clear focus that supported a concept-based approach to introductory programming. Concepts related to object oriented programming were then the focus of the second programming unit. The alignment of these two units to the *what* principles from Chapter 3 is outlined in the following list.

1. The first programming unit, introductory programming discussed in Section 5.2, is aligned with the principles in the following way:
    - Content selection was guided by the structured procedural programming paradigm.
    - Focus is on fundamental programming concepts, which include functions and procedures, variables, control flow, parameter passing, and related concepts.
    - The chosen programming language, or languages, must have been designed for procedural programming.
2. The second programming unit, object oriented programming discussed in Section 5.3, aligns with the principles in the following way:
    - Content selection was guided by the object oriented programming paradigm.
    - Focus is on object oriented programming concepts including abstraction, encapsulation, inheritance, and polymorphism.
    - The chosen programming language, or languages, must have been designed for object oriented programming.

5.2 Introductory Programming

5.2.1 Aims for Introductory Programming

The aim of the introductory programming unit was to introduce students to programming and software development fundamentals. While the focus was on developing depth in this area, the holistic nature of the portfolio assessment approach meant that programming was placed in the context of software development in general. As a result, this unit also touched on a number of areas not traditionally associated with introductory programming such as professional ethics and communication skills.

5.2.2 Defining Intended Learning Outcomes

The first process in creating the introductory programming unit was to define appropriate intended learning outcomes. This was influenced by a number of factors as described in Chapter 4 (see Section 4.2.2). These factors are discussed below, and are followed by a description of the unit's intended learning outcomes that resulted.

Influencing Factors

Figure 4.8 shows the specific factors that influenced the definition of the introductory programming unit. Three aspects will be discussed in the following sections: the overall strategy, accreditation requirements and industry requirements. The objects-later approach meant that this unit focused on procedural programming concepts. Accreditation requirements from the Australian Computer Society (ACS) focused the content on the wider role of software development in general. Whilst the model curriculum from the Association for Computing Machinery (ACM) and Institute for Electrical and Electronic Engineers (IEEE) provided guidance from an industry perspective.

Objects-Later The objects-later approach to this unit meant that it focused on structured and procedural programming concepts. The following list outlines the core concepts that were taught in this unit.

- Procedural programming abstractions:
  - Functional abstractions: functions and procedures
  - Data abstractions: variables, constants, arrays and types

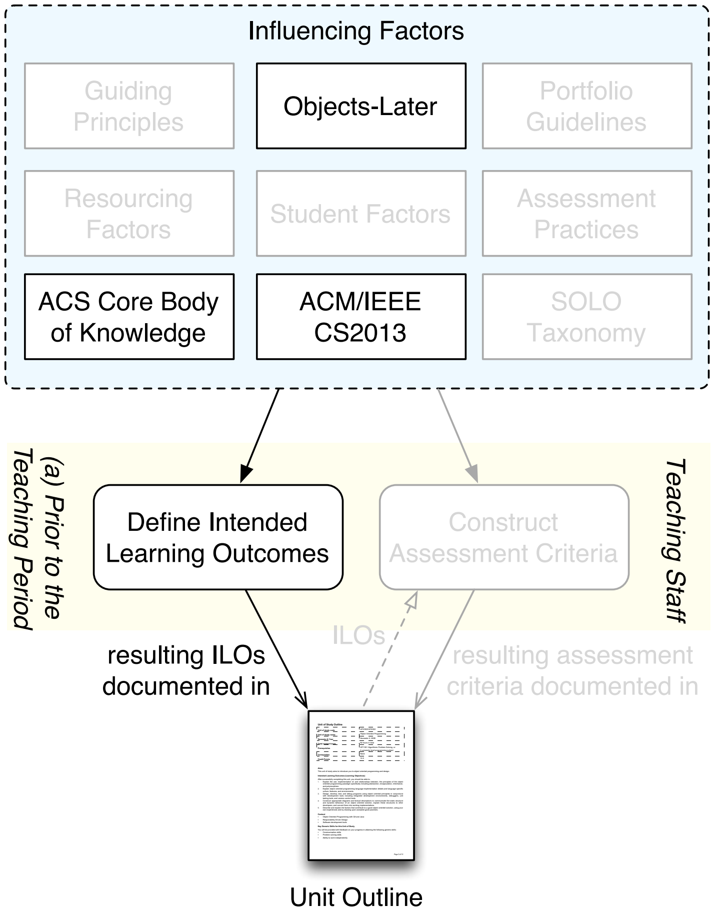

Influencing Factors
Guiding Principles
Objects-Later
Portfolio Guidelines
Resourcing Factors
Student Factors
Assessment Practices
ACS Core Body of Knowledge
ACM/IEEE CS2013
SOLO Taxonomy
(a) Prior to the Teaching Period
Define Intended Learning Outcomes
Construct Assessment Criteria
Teaching Staff
ILOs
resulting ILOs documented in
resulting assessment criteria documented in
Unit Outline

Figure 5.2: Factors that influenced the definition of the intended learning outcomes for introductory programming. Highlighting specific factors from the previous more general factors presented in Figure 4.8.

- Structured programming principles:
  - Sequence, selection and repetition
  - Control flow: pre-test and post-test loops, if and case
  - Iteration through an array of values
- Program comprehension:
  - Memory layout: stack, heap and static memory
  - Execution of control flow
  - Parameter passing: pass-by-value and pass-by-reference

Accreditation Requirements The Australian Computer Society (ACS) documented the ICT profession and associated body of knowledge (Gregor et al. 2008), which indicated graduates should develop both skills and knowledge as part of their undergraduate education. In the work, the skills component drew upon the Skills Framework for the Information Age (SFIA) while the knowledge area was divided into three aspects: a core body of knowledge, role specific knowledge and complementary knowledge. As a central role for a range of IT degrees, the introductory programming unit developed both student's skills and knowledge.

SFIA (Foundation 2011) documented a range of IT skills across six categories. In terms of the SFIA, the Introductory Programming unit aimed to contribute to the development of *programming and software development skill* from the *Solution development and implementation* category. SFIA ranked each skill across seven levels of responsibility, ranging from *follow* to *set strategy, inspire and mobilise*. Introductory programming aimed to provide significant progress towards students attaining a Level 2, *assist*, standard in this skill. To achieve this level of responsibility, students need to demonstrate the ability to design, code, test, correct, and document simple programs, as well as being able to assist with the development of larger software solutions.

The ACS divides the core body of knowledge into six areas: problem solving, professional knowledge, technology building, technology resources, service management and outcomes management. Introductory programming contributed toward the development of the problem solving, professional knowledge, technology building and technology resources as outlined in the following list:

- Problem solving:
  - Students used procedural programming abstractions, and were required to explain their various roles, properties and purpose.
  - Students followed methods and processes for designing and modelling procedural programming solutions.

- Professional knowledge:
    - Students developed general computer competencies, including the use of compilers, shell scripting and basic Bash commands.
    - Students read briefly about the history of computing, and the ICT discipline, providing a context for procedural programming and the structured programming principles.
    - Professionalism, and the role of reflection and life-long learning in professional behaviour was instilled in students.
    - Students performed self assessment of their competencies, and expertise, in applying procedural programming concepts.
    - The emphasis on demonstrating understanding enabled students to develop their written communication skills, including both technical and personal communications.
    - Frequent interaction with staff aimed to help students development their interpersonal skills.
- Technology building:
    - Students experienced many aspects of the software development lifecycle: undertaking simple analysis, design, implementation and testing processes.
    - Students worked with iterative software development processes, building larger solutions across a number of iterations.
    - The procedural programming topics listed developed practical technology building skills.
    - An understanding of the structured programming principles helped guide program construction and evaluation.
    - Students used simple white-box testing techniques to determine the success of their programs.
- Technology resources:
    - Students developed a basic understanding of software systems, including the basics of software processes, memory layout and file systems.

Industry Requirements The Association for Computing Machinery (ACM) and IEEE Computer Society 2013 Computer Science Curriculum documents (ACM/IEEE-CS Joint Task Force 2012) outlines a number of areas to be covered in a Computer Science curriculum. In terms of the ACM/IEEE model curriculum, the introductory programming unit primarily focused on *Software Development Fundamentals*, but also integrated a number of other areas, as shown in the following list.

- Algorithms and Complexity:

  - *Algorithmic Strategies*: Students were introduced to divide-and-conquer, and the idea of recursive backtracking.
  - *Fundamental Data Structures and Algorithms*: All students programmed simple numeric algorithms, sequential search, and basic sorting.
- Computational Science:
  - *Processing*: Fundamental programming concepts were covered in depth including algorithms, implementing algorithms in code, and processes in the software development lifecycle.
- Discrete Structures:
  - *Basic Logic*: Students used truth tables to learn to evaluate and construct boolean expressions.
- Graphics and Visualisation:
  - *Fundamental Concepts*: Applications of computer graphics, double buffering and animation were covered to make programming more interactive.
  - *Geometric Modelling*: Optional tasks allowed students to explore procedurally generated models (fractals).
- Human-Computer Interaction
  - *Programming Interactive Systems*: Students developed code to manage events and user interactions.
- Programming Languages:
  - *Basic Type Systems*: Students explored the use of a range of basic types, along with the definition of custom enumerated and record types.
  - *Language Translation and Execution*: Students were introduced to the topics of compilers and interpreters, as well as run-time layout of memory (call-stack, heap, static data), and manual memory management.
- Software Development Fundamentals
  - *Algorithms and Design*: Students were introduced to the concept of algorithms, problem solving using divide-and-conquer, abstraction and program decomposition.
  - *Fundamental Programming Concepts*: Students used programming language syntax, developed programs that contained statements, expressions, used variables, simple input and output operations, conditional control flow, included functions, various parameter passing techniques, and were introduced to the concept of recursion.
  - *Fundamental Data Structures*: Programs students implemented made use of arrays, record structures, strings and basic string processing, and students implemented a simple linked list.
  - *Development Methods*: Program comprehension was central to the unit, with basic details of program correctness being introduced. Students were also required to use basic refactoring techniques to restructure code, and pro-

gram tracing was covered as a debugging technique.

- Software Engineering
  - *Software Processes*: Students used an iterative software development process model, and were introduced to the phases of the software development lifecycle.
  - *Software Design*: Students were introduced to the principles of the structured design paradigm, and used these principles in the design and development of the programs they created.
  - *Software Construction*: Coding standards, and defensive coding practices were introduced to students.
- Social Issues and Professional Practice:
  - *Professional Ethics*: Students developed skills in professional practice including self assessment, reflective practice, computer fluency, and general approaches to life-long learning.
  - *Professional Communication*: To demonstrate their understanding students were required to read, understand and communicate technical material using clear language and visual mediums.

Intended Learning Outcomes

All of the factors listed above, and the factors and guidelines stated in Section 4.2.2 of Chapter 4, guided the definition of the intended learning outcomes for the introductory programming unit. The following list shows how the guidelines for defining intended learning outcomes were used in the development of the introductory programming unit.

- Verbs were selected from appropriate levels of the SOLO taxonomy. (LO-1)
- Outcomes aimed to cover both an understanding of procedural programming concepts and applied programming skills. (LO-2)
- Care was taken with the language used to help ensure students were able to understand each outcome. (LO-3)
- Only four outcomes were included. (LO-4)
- Outcomes could be met in a variety of ways, and assessed at a range of levels of understanding. (LO-5 and LO-6)

Given the wide range of skills and knowledge mentioned, the challenge was to ensure that this could be expressed in a small number of intended learning outcomes. This task was assisted by the use of the SOLO taxonomy, and recognising that the SOLO level of each outcome indicated that earlier levels must already have been achieved.

Each outcome aimed to engage students in activities likely to help them achieve a *relational* level of understanding. The verbs *analyse*, *apply*, *construct*, *implement*, *interpret* and *use* represent activities in which students need to use cognitive activities at the relational level of understanding. The multistructural *describe* verb and the unistructural *locate* verb are also used, but as a supporting activity for a higher level verb, such as with *describe* and *relate*.

By applying these guidelines, the resulting outcomes helped the introductory programming unit realise the following principles from Chapter 3, as previously discussed in Section 4.2.2.

- Use of verbs at the Relational level of the SOLO taxonomy indicated the activities students needed to be able to demonstrate to pass the unit. (P1, P2, and P5)
- The small number of outcomes provided a clear focus on the concepts related to structured programming. (P4 and P11)
- Flexibility in how outcomes were addressed helped to engage a wide range of student interests. (P6)

The final statement of the intended learning outcomes for the introductory programming unit are listed below, with the verbs from the SOLO taxonomy indicated in bold.

ILO-1: **Apply** code reading and debugging techniques to **analyse**, **interpret**, and **describe** the purpose of program code, and **locate** within this code errors in syntax, logic, and/or good practice.

ILO-2: **Describe** the principles of structured programming, **relate** these to the syntactical elements of the programming language used, and the way programs are developed using this language.

ILO-3: **Construct** small programs, using the programming languages covered that include the use of arrays, functions and procedures, parameter passing with pass-by-value and pass-by-reference, custom data types, and pointers.

ILO-4: **Use** modular and functional decomposition to break problems down functionally, **represent** the resulting structures diagrammatically, and **implement** the structure in code as functions and procedures.

Introductory programming used a procedures-first approach, and focused on the structured programming principles of organising code using *sequence*, *selection* and *repetition*. Students learnt to use functional and modular decomposition to break problems down, and implement solutions using functions and procedures. Data was managed using arrays and custom data types. Pointers and memory management were introduced. Various forms of parameter passing were covered, including pass-by-value and pass-by-reference. Weaved through this was an iterative development process, a focus on writing clear and legible code, and other good programming practices.

In addition to writing code, students learnt to read code for debugging purposes, and to demonstrate their ability to interpret other people's code. This outcome aimed to actively encourage stduents to develop effective models of computation. From a constructivist perspective this model is of critical importance, as it provides depth of understanding in relation to *how* programs work upon the machine, an imperative that was highlighted by Ben-Ari (1998, 2001) in their analysis of constructivism in computer science education. By requiring students to describe how programs work, the intended learning outcomes aimed to ensure students developed effective models of computation.

Other factors, not directly stated as intended learning outcomes, were incorporated in the unit as a means of addressing wider graduate attributes, or as beneficial outcomes not directly assessed in determining the final student grades. This included professional communication, software engineering methods and graphics and visualisation.

Professional Communication Traditionally, many programming units have focused on assessing code outcomes, assuming that if students could produce code, they understood it. Others, such as Lister et al. (2004), extended this to include small code reading and tracing tasks in order to expand on the assessment of student's understanding of code.

With the example introductory programming unit this is taken further, with the students needing to write about the associated principles and to describe code. This expanded assessment had the dual benefit of engaging higher levels of cognitive activity, while also helping students to develop their professional communication skills.

Each of the intended learning outcomes requires students to communicate their understanding. For example, in meeting the first outcome students needed to demonstrate the ability to interpret supplied code, and to communicate its purpose and any issues in logic or the application of recommended good practice. While not a direct focus of the assessment, communication skills play an enabling role in achieving this, and students were provided with support and encouragement in developing their communication skills alongside their technical skills.

Software Engineering Methods While typically the focus of later software engineering units, the software development lifecycle and iterative development methods were embedded within the unit. Students engaged in the development of software throughout the unit: analysing, designing, developing and testing software became a means for them to achieve these outcomes. Students experienced the software development lifecycle first hand, without needing it to be stated in the unit's intended learning outcomes.

Larger programs in the unit were broken down into a number of iterations, and students build these solutions by performing these iterations. Once again, this enabled them to experience software development methods without these methods being explicitly included in the assessed outcomes.

Together, both of these aspects were related to the more general concept of *problem decomposition*. The idea of approaching a solution through discrete iterations provided an example of decomposing a problem into smaller steps. Similarly, the idea of breaking each of these steps into more manageable processes was used to explain both the idea of decomposition and the software development lifecycle.

Graphics and Visualisation Identifying appropriate applications for student to develop is a common problem with teaching introductory programming. In teaching introductory programming it was decided to focus on having students program small computer games. The research literature related to the use of games indicates that this is popular with students (Bayliss & Strout 2006), and helps motivate them to spend time on the task (Feldgen & Clua 2004, Rajaravivarma 2005, Cliburn 2006), as well as supporting development of student understanding of programming concepts (Roberts 1995, Leutenegger & Edgington 2007). Similar approaches focusing on media manipulation (Guzdial & Soloway 2002, Guzdial 2003) have also shown positive results in motivating students to learn introductory programming.

The use of games as a context for software development requires students to gain some familiarity with fundamental concepts related to graphics and animation. The interactive nature of games also introduces students to the concept of programming in response to user input events and real-time signals.

5.2.3 Constructing Assessment Criteria

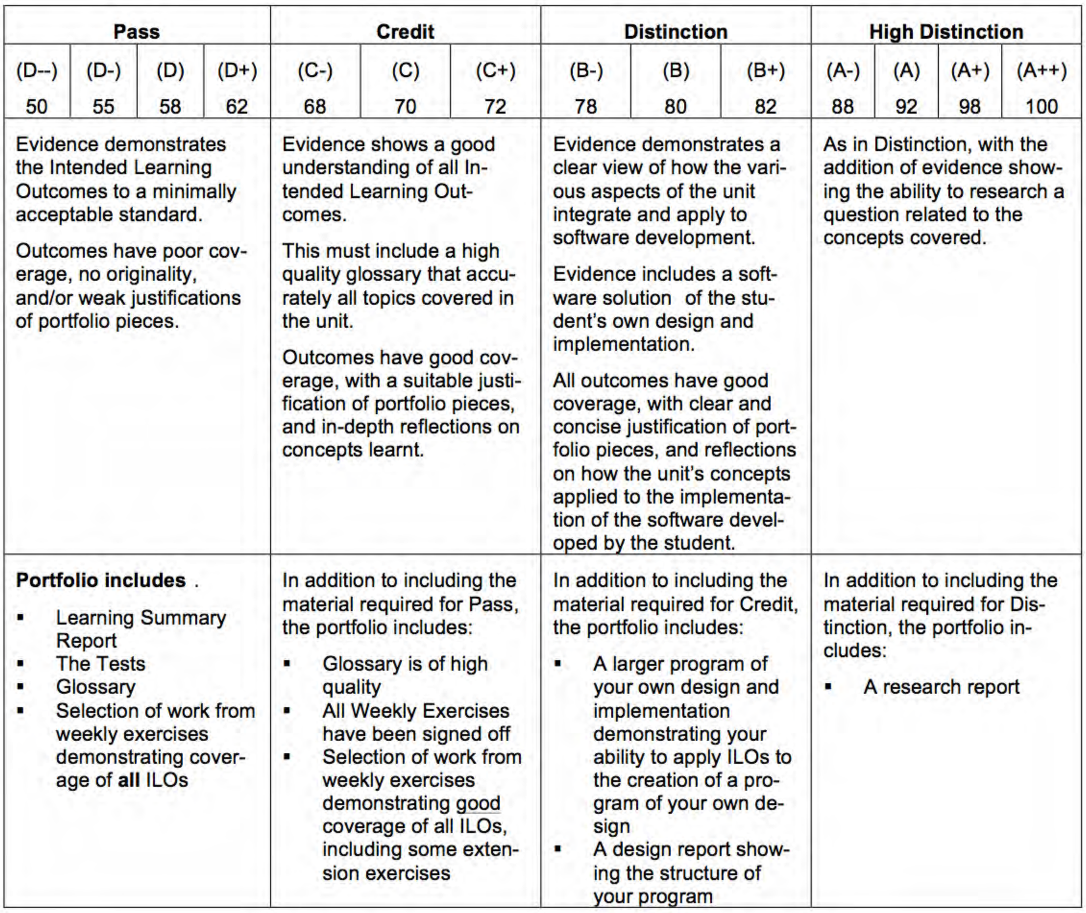
The construction of the assessment criteria happened alongside the definition of the intended learning outcomes. The goal, as stated in Section 4.2.3 of Chapter 4, was to create clearly distinct grades where each required students to demonstrate a deeper understanding of programming concepts and software development practices. These goals were met by applying the guidelines from Chapter 4 as outlined in the following list.

- Simple language was used to describe the various levels to which the outcomes could be met. (AC-1)
- Pass and higher grades required students to demonstrate how they had met each of the unit's intended learning outcomes. (AC-2)
- Critieria for grades higher than Pass required student to demonstrate deeper understanding of the concepts related to the intended learning outcomes. (AC-3)
- Each grade required distinct evidence. (AC-4)
- A clear mapping was provided from the assessment criteria to grade outcomes, and was provided in the unit outline. (AC-5)

Figure 5.3 shows the assessment criteria developed for the introductory programming unit. To receive a *Pass* or *Credit* grade, students completed set exercises, which were signed off through student interaction with staff. *Distinction* required the implementation of a program of the student's own creation, while *High Distinction* required a small research project.

To receive at least a *Pass* grade, students needed to satisfactorily complete three hurdle tests as well as a number of pieces of work that demonstrated they had met all of the intended learning outcomes. These pieces of work needed to come from the weekly tasks, but did not need to have been signed off by teaching staff.

The *Credit* grade required students to meet all Pass requirements, and to have succeeded in getting all tasks signed off. This ensured teaching staff were happy they had completed the work themselves and provided students with incentives to engage in the formative feedback process. Students' explanations of programming concepts and abstractions were the main items used to distinguish between Pass and Credit in terms of depth of understanding. In this regard, the student's work needed to demonstrate good coverage of all outcomes for the student to be eligible for a Credit grade.



<table>
<tr><th colspan="4">Pass</th><th colspan="3">Credit</th><th colspan="3">Distinction</th><th colspan="4">High Distinction</th></tr>
<tr><td>(D--)
50</td><td>(D-)
55</td><td>(D)
58</td><td>(D+)
62</td><td>(C-)
68</td><td>(C)
70</td><td>(C+)
72</td><td>(B-)
78</td><td>(B)
80</td><td>(B+)
82</td><td>(A-)
88</td><td>(A)
92</td><td>(A+)
98</td><td>(A++)
100</td></tr>
<tr><td colspan="4">Evidence demonstrates the Intended Learning Outcomes to a minimally acceptable standard.
Outcomes have poor coverage, no originality, and/or weak justifications of portfolio pieces.</td><td colspan="3">Evidence shows a good understanding of all Intended Learning Outcomes.
This must include a high quality glossary that accurately all topics covered in the unit.
Outcomes have good coverage, with a suitable justification of portfolio pieces, and in-depth reflections on concepts learnt.</td><td colspan="3">Evidence demonstrates a clear view of how the various aspects of the unit integrate and apply to software development.
Evidence includes a software solution of the student's own design and implementation.
All outcomes have good coverage, with clear and concise justification of portfolio pieces, and reflections on how the unit's concepts applied to the implementation of the software developed by the student.</td><td colspan="4">As in Distinction, with the addition of evidence showing the ability to research a question related to the concepts covered.</td></tr>
<tr><td colspan="4"><b>Portfolio includes</b> .
▪ Learning Summary Report
▪ The Tests
▪ Glossary
▪ Selection of work from weekly exercises demonstrating coverage of <b>all</b> ILOs</td><td colspan="3">In addition to including the material required for Pass, the portfolio includes:
▪ Glossary is of high quality
▪ All Weekly Exercises have been signed off
▪ Selection of work from weekly exercises demonstrating good coverage of all ILOs, including some extension exercises</td><td colspan="3">In addition to including the material required for Credit, the portfolio includes:
▪ A larger program of your own design and implementation demonstrating your ability to apply ILOs to the creation of a program of your own design
▪ A design report showing the structure of your program</td><td colspan="4">In addition to including the material required for Distinction, the portfolio includes:
▪ A research report</td></tr>
</table>

Figure 5.3: Assessment criteria from the Unit Outline of the introductory programming unit

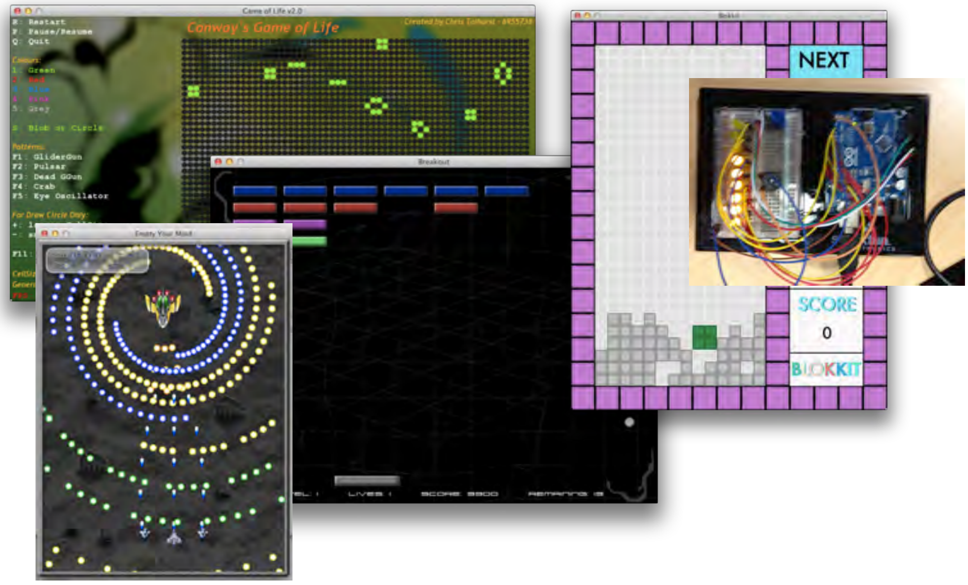


Figure 5.4: Example images of student work, including screenshots of games and a photograph of hardware components.

Distinction built on top of Credit requirements and required students to create a program of their own design. This could be any program the student was interested in creating, as long as it demonstrated good coverage of all of the unit's intended learning outcomes. In effect, this meant that students needed to create a program that contained a number of functions and procedures, used arrays and record types, and was of sufficient size and complexity. Most students who received this grade had implemented a game of some kind, many emulating classic arcade games such as asteroids, pong, frogger or space invaders. However others implements small databases, and in a couple of cases implemented programs for custom hardware. Figure 5.4 shows a number of images of student work, including a number of games and one piece of custom hardware.

High Distinction required students to engage in the creation of a short research report, in addition to having met the Distinction grade requirements. Each student aiming to achieve this grade worked together with staff to define a topic they could examine, and then the student carried out data collection, analysis and reporting tasks. As an introductory programming unit, this research was limited to examining simple tasks such as algorithm efficiency, different techniques to perform a task, or comparing performance aspects of different code. Students were encouraged to think deeply about their results, and to document their outcomes clearly.

As indicated in Section 4.2.3, the use of these assessment criteria in the introductory programming unit helped it to embody the following principles from Chapter 3:

- Criteria for Credit, Distinction, and High Distinction grades required progressively higher levels of understanding which promoted deep learning, and communicated high staff expectations. (P1 and P5)
- Students needed to demonstrate understanding across all outcomes. (P2)
- The criteria determined each student's final grade, delaying all summative assessment until the end of the unit and encouraging students integrate feedback they received. (P3 and P8)
- Different levels in the assessment criteria focused on addressing the most important aspects for students to focus on. (P4)
- As with the learning outcomes, the use of simple terms helped support a wide range of student language capabilities. (P6)
- Clearly distinct requirements helped students take responsibility for their learning, allowing them to aim to meet certain grade criteria. (P7)
- Students reflected on their learning in preparing their portfolios. (P9)
- Students needed to demonstrate appropriate use of procedural programming concepts, and programming languages. (P10, P11 and P12)

5.2.4 Developing Teaching and Learning Activities

Section 4.2.4 of Chapter 4 provided three guidelines to inform the development of teaching and learning activities. These indicated that teaching and learning activities should actively engage students (TLA-1), align with unit outcomes (TLA-2), and focus on providing guidance rather than aiming to transfer knowledge (TLA-3). All of these guidelines were adopted in the development of the teaching and learning activities for the introductory programming unit.

Allocated classes for this thirteen week introductory programming unit included a two hour lecture, and a two hour laboratory class each week. All classes were designed with the goal of actively engaging students, as discussed in Section 4.1.3. A typical lecture included a short presentation using “Beyond Bullet Points” style lecture slides (Atkinson 2007), an interactive programming demonstration and group activities. In the laboratory sessions, students were involved in code reading activities, guided programming tasks and practical hands-on exercises.

The teaching period consisted of twelve teaching weeks, and a single week semester break. Topics for the twelve lectures are shown in the following list. In weeks one to six students explored these concepts using a modern version of the Pascal programming language (Wirth 1971, Van Canneyt & Klämpfl 2011). To help reinforce the applicability of the programming concepts across programming languages, students were introduced to the C programming language (Ritchie et al. 1978) in week 7, and this language was used for the remainder of the semester.

1. Programs, Procedure, Compiling and Syntax
2. User Input and Working with Data
3. Control Flow: Branches and Loops
4. Procedural and Structured Programming
5. Arrays
6. Custom Data Types and Pointers
7. Learning a New Language
8. Programming in C
9. File Input and Output
10. Dynamic Memory Management
11. Recursion and Backtracking
12. Review and Future Studies

Each week’s laboratory class consisted of a number of activities. At the start of the class the teaching staff returned the feedback from the previous week’s core exercises, but delayed discussing feedback with students until after working through the exercises. Students where then guided through the week's laboratory exercises, and the core exercises were discussed. Students spent the remaining laboratory time on the week's core exercises. During this time the teaching staff visited each student individually to discuss their progress, and to mark their work as signed off. At the end of laboratory classes students were reminded of the tasks they needed to complete by the following week, and encouraged to attempt extension tasks.

To provide students with an opportunity to demonstrate their understanding, each week's core exercises also had students developing a detailed glossary. For the introductory programming unit, the glossary had students record details on the following topics:

- Core concepts:
  - Control flow
  - Structured programming principles
  - Functional and modular decomposition
  - Good programming practices
- Programming terminology:
  - Statements
  - Expressions
  - Identifiers
  - Parameters, local variables and global variables
- Programming abstractions:
  - Programs
  - Functions and Procedures
  - Constants and Variables
  - Arrays
  - Records and Enumerations
  - Pointers
- Statements:
  - Function and procedure calls
  - Assignment statements
  - If and case statements
  - While and repeat/do..while loops
  - For loops

Table 5.2 shows the planned alignment between the introductory programming unit's topics and its intended learning outcomes. This planned alignment was not shared with students to ensure that their portfolios reported how *they* believed the tasks aligned with the outcomes. In this way, the process of alignment was carried out by both staff *and* students – teaching staff planned activities they believed aligned with outcomes, while students reflected on their learning experience and reported how their work demonstrated they had met the outcomes.

By following the guidelines from Section 4.2.4, the teaching and learning activities in the introductory programming unit addressed a number of the principles stated in Chapter 3.

- Activities aimed to actively engaged students in tasks likely to engage the required cognitive levels. (P1)
- Staff planned activities they believed would enable the students to demonstrate they had met the unit's intended learning outcomes. (P2)
- Activities supported the use of weekly formative feedback, providing tasks over which staff and students could have meaningful dialogue. (P3 and P7)
- Activities focused on concepts, providing students with a range of opportunities to develop pieces that would demonstrate they had gained the required knowledge. (P4, P6 and P11)
- Core and extension tasks helped to communicate staff expectations. (P5)
- Leaving language details to be covered in teaching and learning resources helped to ensure activities could change in response to student needs. (P8)
- Activities ensured students implemented a range of procedural programs, making use of programming languages in ways in which they were intended. (P10 and P12)

Table 5.2: Alignment matrix showing staff-planned alignment of weekly topics to the introductory programming unit's intended learning outcomes. Student descriptions of the topic alignment differed based on their individual learning.

Topic	**ILO-1**	**ILO-2**	**ILO-3**	**ILO4**
Programs, Procedure, Compiling and Syntax	✓	✓	✓	✓
User Input and Working with Data	✓	✓	✓	✓
Control Flow: Branches and Loops	✓	✓	✓	
Procedural and Structured Programming	✓	✓	✓	✓
Arrays	✓	✓	✓	
Custom Data Types and Pointers	✓		✓	
Learning a New Language	✓	✓	✓	✓
Programming in C	✓	✓	✓	✓
File Input and Output			✓	
Dynamic Memory Management	✓		✓	
Recursion and Backtracking	✓		✓	
Review and Future Studies	✓	✓	✓	✓

Procedures First Topic Sequence

The order of topics was guided by Principle 11 from Chapter 3. The main objective being to enable each week's topic to build upon earlier topics, while providing a consistent set of abstractions for students to work with.

This was achieved using a procedures-first approach in which students program their own procedures from Week 1. The following list indicates the concepts, programming abstractions and statements introduced each week.

1. Programs, Procedure, Compiling and Syntax
    - **Sequence**: The focus of this week was on programs and procedures as a *sequence* of instructions that get the computer to perform a task.
    - **Syntax Rules**: Students learnt to use visual "railroad" diagrams (Braz 1990) to understand programming language syntax.
    - **Program**: Students created small programs that contained a sequence of procedure calls, with all values being hard coded.
    - **Procedure**: Students developed a small number of procedures, each with a defined task that contributed to the overall program.
2. User Input and Working with Data
    - **Data**: The central idea of this week was *data*, the idea that values can be stored and calculated.
    - **Variables**: Students created local variables, global variables and parameters.
    - **Constants**: Students declared constants.
    - **Functions**: Students used functions in the laboratory and core exercises, and developed their own in the extension tasks.
    - **Assignment Statements**: Students used assignment statements to store values in variables.
3. Control Flow: Branches and Loops
    - **Control Flow**: The ideas of sequence, selection and repetition were central to this week.
    - **Selection**: Students used **if statements** and **case statements** to implement branching in their control flow.
    - **Repetition**: Students repeated code using **while loops** and **repeat loops**.
4. Procedural and Structured Programming
    - **Functional and Modular Decomposition**: The idea of solving problems using divide-and-conquer was the main theme of this week.
    - **Software Development Lifecycle**: The basic steps of the software development lifecycle were discussed.

   - **Iterative Development**: The idea of iteratively working toward a solution was discussed.
   - **Structured Programming**: The structured nature of a functions/procedures code was discussed, illustrating how these steps can be broken down into blocks performing sequence, selection or repetition.
   - No new programming abstractions or statements were introduced in this week.
5. Arrays
   - **Arrays**: Students started to use arrays to store multiple values in their programs.
   - **For loops**: The for loop was introduced as a convenient means of looping through the contents of an array, allowing the code to easily apply a set of operations on each element in the array.
6. Custom Data Types and Pointers
   - **Types**: Custom types and the role of types in a programming languages were discussed.
   - **Records**: Students developed their own custom record types to model entities associated with their programs.
   - **Enumerations**: Students use of enumerations as a means of creating a type to represent a list of options.
   - **Pointers**: Pointers were used to create relationships between values, and to illuminate how pass-by-reference worked internally.
7. Learning a New Language
   - **Language Syntax**: Ways to approach a new programming language were discussed, and students used a new language to recreate previously developed programs.
8. Programming in C
   - No new programming concepts or abstractions were presented in this week. Instead the week was used to consolidate knowledge of the new language, and to develop a wider range of programs using the previously presented concepts.
9. File Input and Output
   - **File Input and Output**: The idea of persisting data was presented, and students learnt to save data to a file and read it back.
10. Dynamic Memory Management
    - **Stack and Heap**: Specifics related to memory layout were discussed, including the limitations of the stack.
    - **Dynamic Memory Allocation**: Students used memory allocation functions to allocate memory from the heap, and used pointers to work with the newly allocated space.

11. Recursion and Backtracking
    - **Recursion**: Students developed simple recursive solutions for problems like the Fibonacci sequence and the Towers of Hanoi.
    - **Backtracking**: The idea that a recursive solution can backtrack to search alternative paths was discussed, and extension tasks introduced backtracking to solve Sudoku and the Eight Queens puzzle.
12. Review and Future Studies
    - This week did not introduce any new programming concepts or abstractions, instead it was used to review everything that had been covered and to discuss future programming units.

Pattis (1990, 1993) indicated that the main challenge with the procedures first approach was that students did not have anything meaningful to program in their procedures prior to introducing control flow. Pattis (1993) reported that the majority of procedures-first texts had moved control flow prior to procedure declarations in response to this issue. We took an alternative approach, and introduced students to a game development framework in Week 1. This framework provided a range of useful procedures that students could call, and thereby addressed the issues that Pattis had raised. Details of this game development framework are presented in Chapter 6.

The following list outlines the focus of each week's tasks.

1. Programs, Procedure, Compiling and Syntax
    - **House Drawing**: Students created a procedure to draw a house at a fixed location on the screen, using a sequence of procedure calls.
    - **Knock Knock**: Students created a number of procedures to show images and play sound effects necessary to display a knock knock joke.
    - **Custom Splash**: Extensions encouraged students to develop a procedure to show their own splash screen.
2. User Input and Working with Data
    - **House Drawing**: Redeveloped to make use of parameters and local variables, allowing the house position to be changed.
    - **Bike Drawing**: Students developed a procedure to draw a fixed size bike, using parameters for the bike's location and colour.
3. Control Flow: Branches and Loops
    - **Lots of Bikes**: Students created a custom screen saver like program that drew thousands of bikes to the screen.
    - **Circle drawing**: Students developed a program in which they could move a circle around the screen, switching it between outlined and filled modes, and mouse clicking it to make the circle jump to a random position.

4. Procedural and Structured Programming
    - **User input functions**: Students created functions to read strings, integers, doubles, and ranges of values (between a minimum and maximum) from the user. These build on top of each other to demonstrate the power of functional decomposition.
    - **Arcade game**: Students were encouraged to develop a simple clicking arcade game using the concepts already covered.
5. Arrays
    - **Statistics**: Students developed a program that read in a number of values from the user and then calculated various statistic values. This included median, which required the values in the array to be sorted.
6. Custom Data Types and Pointers
    - **Address Book**: A simple address book, with links to friends was developed to demonstrate records and pointers.
    - **Pop Game**: Provided a larger, multiple iteration, project in which students developed a simple arcade style game. The game involved popping different kinds of shapes, with rounds of ten shapes each of random sizes and colours.
7. Learning a New Language
    - **User Input Functions**: Students redeveloped the user input functions using the new programming language.
8. Programming in C
    - **Statistics** was reimplemented in the new programming language.
9. File Input and Output
    - **Address Book** was reimplemented in C, and added the ability to save it to file, and load it from file.
10. Dynamic Memory Management
    - **Address Book**: Dynamic memory management was used to allow for a variable number of contacts in the address book.
    - **Maze Game**: Extension tasks had the students develop a network of rooms connected using pointers.
11. Recursion and Backtracking
    - **Recursive Programs**: Extension tasks had students develop recursive functions.
    - **Linked List**: Extension tasks demonstrated how to use pointers and records/structures to create a linked list.
12. Review and Future Studies
    - There was no lab exercises this week, and the lab was used to help students complete any outstanding tasks.

Programming Language Choice

Introductory programming aimed to teaching students to *program*, not the details of a programming language. We did not aim to develop students' expertise in one programming language, but to equip them with the knowledge and skills to become proficient in **any** imperative programming language.

Whenever discussing introductory programming it is always interesting to note how quickly people jump to the question of language. In the delivery of this unit the programming language was always a secondary concern – an enabling feature – not an aspect of great importance. The language choice was based on its ability to support the following requirements:

- **Explicit** *over* implicit
  - Require explicit variable declaration, with clear indication of the variable's type.
  - Strongly typed, avoiding implicit type conversions.
- **Procedural programming abstractions** *over* support for other paradigms.
  - Functions and procedures.
  - Pass-by-value and pass-by-reference.
  - Arrays, constants and variables.
  - Declaration of custom types including enumerations and record structures.
  - Pointers and dynamic memory management.

Table 5.3: Comparison of programming languages for the introductory programming unit.

	C	C++	C#	Java	Pascal	Python
Explicit Variables	✓	✓	✓	✓	✓	
Strongly Typed			✓	✓	✓	✓
Functions and procedures	✓	✓	Partial		✓	✓
Parameter passing options		✓	✓		✓	
Data abstractions	✓	✓	✓	✓	✓	✓
Records and enumerations	✓	✓	✓		✓	
Pointers	✓	✓	Partial		✓	
Explicit memory management	✓	✓			✓	
Used by staff	✓	✓	✓	✓	✓	✓

Table 5.3 lists the languages considered for the introductory programming unit. Pascal met all of the requirements, with C++ and C# satisfying most. C# was designed

for object oriented programming, and therefore using it for procedural programming was against Principle 12, which indicated we would only use languages as they were designed to be used.

The choice between C++ and Pascal was more challenging. C++ had the advantage of being widely used, and the popularity of the C-style syntax would mean that learning its syntax would more directly help students when with these other languages. However, the C++ language itself is cryptic and would be more challenging for students to master on their own. Pascal, on the other hand, was seen as having a more "beginner friendly" syntax. Pascal had also been used by Becker (2002), who stated that it had enabled them to focus less on the syntax, as compared to C++.

Rather than choosing *one* language it was decided to take advantage of both languages. Students would be able to focus more on the concepts using Pascal's more friendly syntax over the first few weeks of the semester. Once all the concepts were covered the language could be switched and students could explore the procedural aspects of C++, getting a start with a C-style syntax.

The focus on teaching students to "understanding syntax" from Week 1 supported, and was supported by, the change of language. Students had been exposed to programming language syntax using the visual "railroad" diagram syntax notation (Braz 1990), and had used this to learning Pascal over the first weeks of the unit. After the switch to C++, students could reinforce these same skills by applying them to learn a second programming language. Students again consulted railroad diagrams as they learnt the C syntax themselves. This approach aimed to encourage students to focus on the concepts, which would then enable them to more quickly learning the new language.

5.2.5 Delivering the Unit

Delivery of the unit followed the iterative delivery process outlined in Section 4.2.5 of Chapter 4, and used the activities outlined in the previous section. The process involved the delivery of lectures to guide student activity, used resources to provide students with details, provided weekly formative feedback on assessment tasks, which students then incorporated to improve their understandings. The following list outlines how the guidelines from Section 4.2.5 were adopted, with additional details provided in the following paragraphs.

- Lecture and laboratory classes aimed to actively engage students, as described in Section 5.2.4. (IDU-1 and IDU-2)

- Students submitted work weekly for formative feedback, with the portfolio being the only work that contributed to the students' grades. (IDU-3)
- An online discussion board and Programming Help Desk were used to actively support students throughout the teaching period. (IDU-4)

Each week the lecture presented the relevant concepts, with demonstrations introducing students to the syntax. Laboratory sessions helped students prepare for the core tasks, which they completed after each class. Completed core and extension tasks were submitted for formative feedback at the start of subsequent lectures and the process repeated across each week of the teaching period.

With lectures providing guidance, and laboratories getting students started with a topic, students often had questions outside of scheduled class times. These were supported via a Programming Help Desk and an online discussion board. The Programming Help Desk was available to students during the week, and was staffed by later year students and teaching staff from the unit. Students were encouraged to drop in to the help desk to seek clarification on any issues they had with the concepts or programming language syntax. The unit also made heavy use of the university's online learning management system to provide additional support for students outside of scheduled class times. The use of the discussion board was actively encouraged, and questions were promptly answered by teaching staff, and in some cases by fellow students.

Delivering the introductory programming unit in this way ensured it met all of the principles stated in Chapter 3, as outlined in the following list.

- Students were actively engaged in constructing their own knowledge as they worked through the teaching and learning activities. (P1)
- Activities provided opportunities for students to develop pieces of work that aligned to the unit's intended learning outcomes. (P2)
- Weekly tasks were used to provide frequent formative feedback, with students actively being encouraged to incorporate the feedback to improve their work and understanding. (P3)
- Activities had a clear focus: lectures providing guidance, laboratory classes a first exposure, and support structures providing assistance outside of scheduled class times. (P4)
- Weekly tasks, along with extensions, helped communicate high staff expectations. (P5)

- The discussion board and Programming Help Desk provided active support for students in addition to scheduled class times. (P6)
- Student had to take responsibility for their learning, no marks were attached to any of the weekly work so as to ensure it retained its formative focus. (P7)
- Execution of teaching and learning activities provided insights that could be feed back into the reflective agile process, ensuring changes helped address student needs. (P8 and P9)
- The concepts central to the unit focused on structured procedural programming. (P10 and P11)
- Students used the Pascal and C programming languages in ways in which they are intended to be used. (P12)

5.2.6 Assessing Student Portfolios

At the end of the teaching period student portfolios were assessed using the process shown in Figure 5.5. This illustration had been included in the unit outline, to help communicate the assessment approach to students. This process adheres to the guidelines presented in Section 4.2.6 of Chapter 4 as outlined in the following list.

- Students were encouraged to include pieces of work that were relevant to them personally, including work such as hand written notes or learning journals. Any work that was included had to be referred to in the student's Learning Summary Report, ensuring it aligned with at least one of the unit's intended learning (AP-1)
- All work included in the students' portfolios should have incorporated feedback they had received during the teaching period. (AP-2)
- The Learning Summary Report required each student to reflect on their learning, and how the pieces they had included demonstrated they had achieved the unit's intended learning outcomes. (AP-3)
- Hurdle tests were used, and included as a compulsory piece in student portfolios. These tests only checked core competencies that all students should have been able to easily demonstrate if they had met all of the unit's intended learning outcomes. Students who aimed for a Distinction or High Distinction grade were interviewed to allow them to demonstrate how they had met the assessment criteria to these standards. (AP-4)
- Portfolios were assessed using the assessment criteria expressed in the Unit Outline document. (AP-5)

Portfolio assessment by staff involved the following steps:

1. Determine a starting grade from the student's self assessment, cross referenced with data collected by staff during the teaching period.
2. Initially assume the work is "average" within its grade category, then examine the evidence and student's self assessment to determine if the result is of a higher or lower standard.
3. Portfolios sorted by grade outcomes, and quickly compared for consistency.
4. After assessing all of the portfolios, re-examine the portfolios of students who achieve the "top" High Distinction grade and determine if any should be awarded a perfect score.

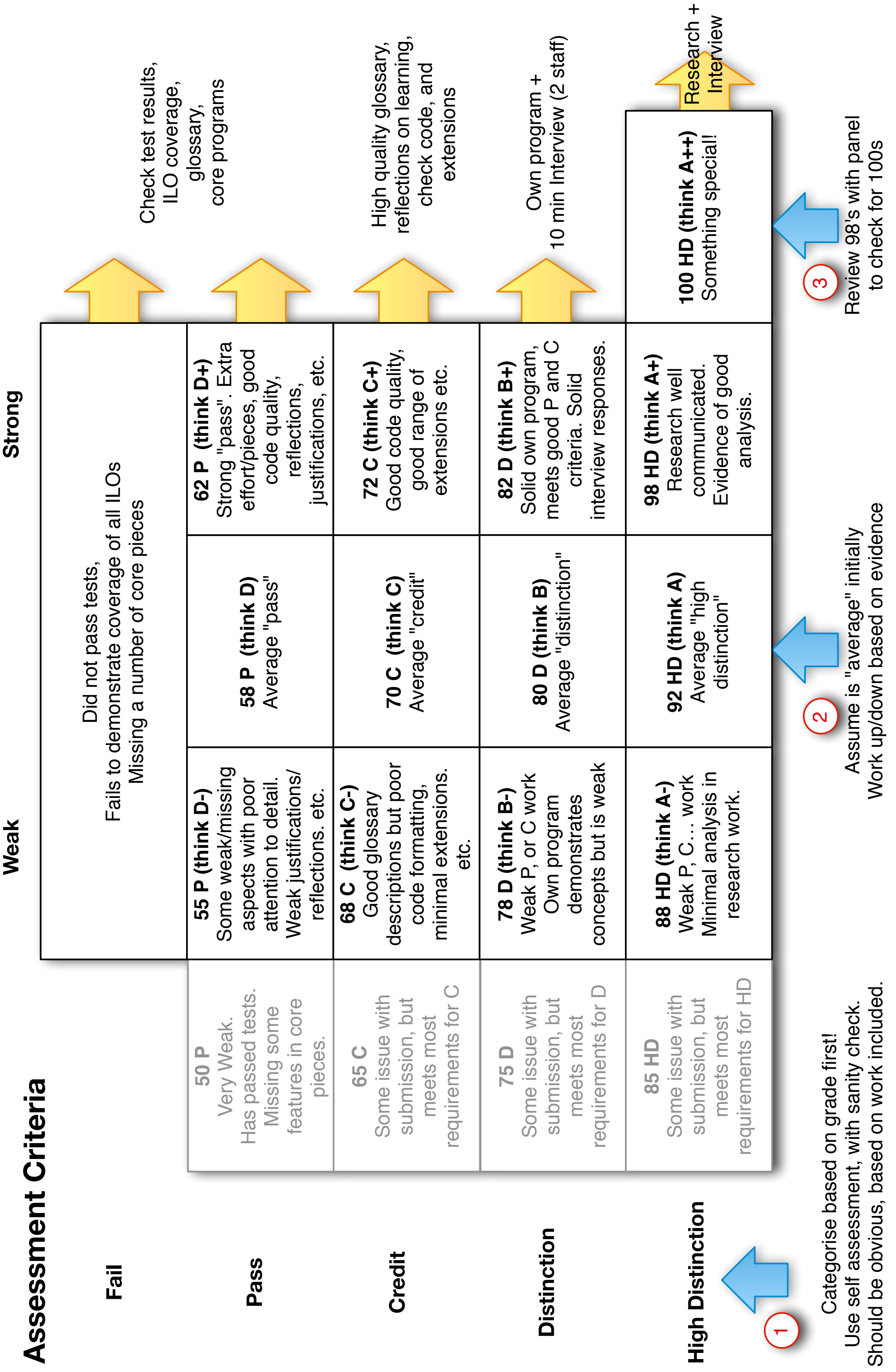
The clearly distinct criteria for each grade made determining portfolio grades a simple task, with consistent outcomes. Pass criteria required students to have satisfactorily completed the hurdle tests, Credit required a good quality glossary and all work to be signed off, Distinction the custom project, and High Distinction the research report.

For each of the grades the quality of the distinguishing artefacts needed to be checked against the expected standard. In the final portfolio assessment this task was greatly simplified due to the requirement for students to engage in the formative feedback process for Credit and higher grades. This meant that student work had already been checked by teaching staff, typically a number of times, before their portfolios were submitted. Any issues should have been identified and corrected before the final submission.

Students aiming for the Distinction and High Distinction grades were all interviewed. Each interview lasted around ten minutes, and was conducted by multiple teaching staff. In the interviews students were asked about their custom project and research work. The portfolio was used to guide the discussion, with screenshots and print outs of code often referred to. Overall the experience was very positive for students and staff, and provided staff with an opportunity to engage with the students who had achieved the most in the unit.

Assessing the students of the introductory programming unit in this way ensured all of the principles stated in Chapter 3 were embedded throughout the unit.

- Assessment of portfolios aimed to assess the structure of the observed learning outcomes, as demonstrated in student portfolios. (P1)
- Portfolios had to demonstrate how the students' work aligned with *all* of the unit's intended learning outcomes. (P2)
- Final grades were based entirely on student portfolios, enabling frequent formative feedback to support student efforts during the unit delivery. (P3, P6 and



Assessment Criteria

		Weak		Strong		
Fail		Did not pass tests, Fails to demonstrate coverage of all ILOs Missing a number of core pieces				
Pass	50 P Very Weak. Has passed tests. Missing some features in core pieces.	55 P (think D-) Some weak/missing aspects with poor attention to detail. Weak justifications/ reflections. etc.	58 P (think D) Average "pass"	62 P (think D+) Strong "pass". Extra effort/pieces, good code quality, reflections, justifications, etc.		Check test results, ILO coverage, glossary, core programs
Credit	65 C Some issue with submission, but meets most requirements for C	68 C (think C-) Good glossary descriptions but poor code formatting, minimal extensions. etc.	70 C (think C) Average "credit"	72 C (think C+) Good code quality, good range of extensions etc.		High quality glossary, reflections on learning, check code, and extensions
Distinction	75 D Some issue with submission, but meets most requirements for D	78 D (think B-) Weak P, or C work Own program demonstrates concepts but is weak	80 D (think B) Average "distinction"	82 D (think B+) Solid own program, meets good P and C criteria. Solid interview responses.		Own program + 10 min Interview (2 staff)
High Distinction	85 HD Some issue with submission, but meets most requirements for HD	88 HD (think A-) Weak P, C... work Minimal analysis in research work.	92 HD (think A) Average "high distinction"	98 HD (think A+) Research well communicated. Evidence of good analysis.	100 HD (think A++) Something special!	Research + Interview

1 Categorise based on grade first!
Use self assessment, with sanity check.
Should be obvious, based on work included.

2 Assume is "average" initially
Work up/down based on evidence

3 Review 98's with panel
to check for 100s

Figure 5.5: An overview of the assessment process used to explain the criteria to students.

P7)
- Assessment could focus on structured procedural programming concepts, as well as how they are realised in student programs. (P4, P10 and P11)
- Assessment criteria provided a clear message of high expectations, indicating what was required for students to achieve good grades. (P5)
- Portfolios provided valuable evidence on the effectiveness of the learning environment, activities, and resources for staff reflections. (P8 and P9)
- Portfolios provided opportunities for students to reflect on their learning. (P9)
- Programs created by students needed to demonstrate appropriate use of the programming languages used. (P12)

5.2.7 Introductory Programming in Summary

This section has presented an application of the model presented in Chapter 4, with the resulting unit encapsulating all of the principles from Chapter 3. The introductory programming unit was centred around its intended learning outcomes, and the central role of the student in constructing their own knowledge. The assessment criteria rewarded students for demonstrating a depth of knowledge, pushing students to strive for relational level understanding. Teaching and learning activities were developed to support the constructive nature of the unit, and to provide students with suitable challenges. The final summative assessment used assessment criteria and a specific process to quickly and efficiently determine student grade outcomes from the portfolios they submitted.

Chapter 7 and Chapter 8 provide further discussion of the results from delivering this unit.

In the next section another application of the model is presented in discussing the object oriented programming unit.

5.3 Object Oriented Programming

5.3.1 Aims for Object Oriented Programming

Object oriented programming was the second programming unit in the sequence of programming units described in Section 5.1. Having completed the introductory programming unit, many students went on enrol in the object oriented programming unit, which introduced them to the object oriented programming paradigm.

The design of the object oriented programming unit followed the model outlined in Section 4.2 in a similar way to the introductory programming unit presented in Section 5.2. To help illustrate the general applicability of the model, this section briefly describes the steps taken and discusses how these differ from the approach taken with the introductory programming unit. The following subsections outline the processes from the model described in Section 4.2, including the definition of the intended learning outcomes, construction of the assessment criteria, development of the teaching and learning activities, delivery of the unit, and portfolio assessment.

5.3.2 Defining Intended Learning Outcomes

Defining the object oriented programming unit's intended learning outcomes followed the same process as outlined for the introductory programming unit in Section 5.2.2. The process was influenced by similar factors, and used the guidelines stated in Section 4.2.2. The main difference between the object oriented programming unit and the introductory programming unit was the underlying programming paradigm.

Principle 10 indicates that the strategy for delivering the unit should be founded on the programming paradigm being taught. The introductory programming unit covered procedural programming concepts, with the plan that the object oriented programming unit shift students to the object oriented programming paradigm, a shift that can be challenging for student (Manns & Nelson 1993, Sheetz et al. 1997, White & Sivitanides 2005). In recognition of these challenges, the unit's intended learning outcomes focused on enabling students to make this shift in thinking.

To facilitate the paradigm shift the unit focused on the core principles underlying the object oriented programming paradigm: abstraction, encapsulation, inheritance, and polymorphism. Students learnt to use these principles to design and implement object oriented programs. In addition to the core principles, students learnt to use an integrated development environment (IDE), and unit testing tools such as JUnit (Junit n.d.). Object oriented thinking and design were also central to the unit, with designs being communicated using Unified Modelling Language (UML) (OMG 2011) class diagrams and sequence diagrams. While design patterns and heuristics provided students with a means of evaluating the quality of their designs.

The intended learning outcomes for the object oriented programming unit were:

ILO-1: **Explain** the principles of the object oriented programming paradigm specifically including abstraction, encapsulation, inheritance and polymorphism, and **explain** how these principles are used to create object oriented programs.

ILO-2: **Design**, **develop**, **test**, and **debug** object oriented programs, using an integrated development environment.

ILO-3: **Select** and **use** appropriate collection classes, from the language's class library, to manage collections of multiple objects.

ILO-4: **Construct** appropriate diagrams and textual descriptions to **communicate** the static structure and dynamic behaviour of an object oriented solution.

ILO-5: **Apply** accepted good practices related to the **construction** of object oriented programs.

The following list shows how the guidelines for defining intended learning outcomes, stated in Section 4.2.2, were used in the development of the object oriented programming unit.

- Verbs from the relational levels of the SOLO taxonomy were used to set appropriate levels of understanding. (LO-1)
- Outcomes aimed to cover both an understanding of object oriented programming concepts and applied programming skills. (LO-2)
- Simple terms were used to help ensure students were able to understand each outcome. (LO-3)
- Five outcomes were included, keeping the focus on a few key aspects. (LO-4)
- Outcomes were expressed in a general sense, enabling them to be met in a variety of ways and assessed at a range of levels of achievement. (LO-5 and LO-6)

These guidelines helped with the formation of the intended learning outcomes for the object oriented programming unit, as they had introductory programming unit. The same general approach was taken, with differences in outcomes relating to the various unit focuses.

5.3.3 Constructing Assessment Criteria

The general nature of the assessment criteria from the introductory programming unit enabled these same criteria to be used for the object oriented programming unit. These did not need adjustment as each grade was described in a topic neutral manner. For example, for a Distinction students needed to "demonstrate the *application* of the unit's concepts to the creation of a program of their own design." This statement worked equally well for the object oriented programming unit as it did for the introductory programming unit.

In essence, the assessment criteria described in Section 5.2.3 aimed to ensure students had reached a relational level of understanding in order to achieve a Distinction or higher grade, and as a result the assessment criteria are likely to work for other technical software engineering units. Removing all topic specific details from these criteria results in criteria that could form the basis for a wide range of university units, as outlined in the following list.

Pass is awarded to students who have done no more than meet the unit's minimum standards, as checked in weekly work or hurdle tests.

Credit is awarded to students who have demonstrated an integrated understanding of unit concepts, but have not demonstrated they can *apply* the concepts to the creation of a piece of work of their own creation.

Distinction is awarded to students who have demonstrated an integrated understanding of unit concepts, **and** have demonstrated they can *apply* the concepts to the creation of a piece of work of their own creation.

High Distinction is awarded to students who have met the Distinction criteria and go beyond the scope of the unit in some way, such as through conducting a small research project.

The following list outlines how these general assessment criteria relate to the principles stated in Chapter 3.

- Students are encouraged to meet all intended learning outcomes, and to develop a relational level of understanding to achieve higher grades. (P1 and P2)
- If portfolios determine all of a student's final grade then all summative assessment is delayed until the end of the unit. (P3)
- Student attention is focused on achievable steps at each level of the assessment criteria. (P4)
- Criteria for Distinction and High Distinction grades help communicate high staff expectations. (P5)

- Simple terms help support a wide range of student language capabilities, helping to ensure students will understand what is required of them. (P6)
- Use of clearly distinct requirements allow students to aim to meet certain grade criteria based on their personal interests and motivation. (P7)
- Frequent formative feedback encourages students to reflect on their learning and integrate feedback they receive. (P8 and P9)
- Students needed to demonstrate appropriate application of concepts associated with the unit. (P10, P11 and P12 in the case of introductory programming units)

5.3.4 Developing Teaching and Learning Activities, and Delivering the Unit

Teaching and learning activities in the object oriented programming unit did differ from those in the introductory programming unit due to the altered focus of the unit. Section 5.1 outlined the programming concepts shared between procedural and object oriented programming, as illustrated in Table 5.1. This shows that object oriented programs and procedural programs share many features, and therefore many low level details do not need to be elaborated upon as students should be familiar with these concepts from the introductory programming unit.

Object oriented programming involves constructing programs that consist of a number of interrelated object, with each object interacting with other objects to help achieve system goals. The thinking involved in designing object oriented programs, therefore, differs significantly from structured procedural programming. While the focus on the introductory programming unit had been on the internal implementation details of functions and procedure, the object oriented programming unit focused on modelling objects and designing object interactions. This required a different approach to unit teaching and learning activities.

As with the introductory programming unit, the guidelines stated in Section 4.2.4 helped to shape the teaching and learning activities used. The activities aimed to actively engage students (TLA-1), align with unit outcomes (TLA-2), and provide guidance (TLA-3). These guidelines resulted in the use of the Beyond Bullet Points approach to presentations and object role-plays in lecture sessions, and the development of a number of case studies in laboratory classes.

Lectures were developed using the Beyond Bullet Points approach to ensure the focus was on concepts (TLA-3 supporting P4 and P11). The short lecture presentations provided opportunities to engage students in more interactive activities in scheduled lecture times (TLA-1 supporting P1). Where the introductory programming had made use of interactive code demonstrations, the object oriented programming used role-plays to focus students on object interactions, rather than language syntax (P11). In these role-plays students take on the role of objects in a program and send each other messages, simulating the objects students would later create in their code. Similar use of object role-plays was reported by Börstler & Schulte (2005), who found that they were beneficial in helping students understand object oriented concepts and thinking.

The nature of object oriented paradigm means that it is best suited to developing larger solutions than those used in the introductory programming unit. The use of the object oriented programming principles, specifically inheritance and polymorphism, require a certain amount of infrastructure to see how these dramatically influence the way programs are designed. To help students engage with these principles, and to encourage students to develop greater learning independence, laboratory classes were adjusted to focus on a smaller number of programming problems that were larger than problems addressed in the introductory programming unit. These decisions were guided primarily by the chosen paradigm (P10) and the need to ensure programs students create are an appropriate use of the chosen languages (P12).

A total of four programming "case studies" were used in the object oriented programming unit, two in the first half of the unit and a further two in the second half of the unit. One case study in each half of the teaching period was designed with significant guidance for students, and replaced the laboratory tasks that had been used in the introductory programming unit. The second case study required the application of the same principles, but with less guidance helping students to apply the concepts being learnt (P1). UML diagrams were provided to explain the design of each case study, with iterations building relevant parts of the overall program. As the iterations progressed the details of the design were reduced, with students being required to design and implement the later iterations.

Case studies supported students active role in the development of their knowledge and ensured students made appropriate use of the associated concepts (TLA-1 and TLA-3 supporting P1, P10, P11, and P12). As with the introductory programming unit, the alignment was performed by both staff and students. Staff planned for the case studies to align with the unit's intended learning outcomes, and students related their individual learning to the outcomes in the preparation of their portfolios (TLA-2 supporting P1, P2 and P11).

In terms of alignment (P2) teaching staff planned for each case study to align with all of the intended learning outcomes, as shown in Table 5.4. The following list indicates how the staff planned for each of the intended learning outcomes to relate to the case studies. Students were also required to indicate how they felt the pieces they had included in their portfolios aligned to the unit's intended learning outcomes, as was done in the introductory programming unit.

- Students understanding of object oriented principles (ILO-1) was central to each case study, with students encouraged to reflect on how these principles were realised in the designs they implemented.
- Each case study involved aspects of design, development, testing, and debugging (ILO-2) with students building functional programs.
- The programs developed required students to use a range of classes from the languages' class libraries, including a variety of collection classes.
- Students needed to read as well as develop UML class and sequence diagrams in various parts of the case studies.
- Designs for the case studies demonstrated accepted good practice, and the iterative formative feedback process ensured students followed these, and associated, practices in their design and implementation tasks. Later case studies also introduced students to design patterns including the Composite and Command patterns (Gamma et al. 2001).

Core, and laboratory, tasks in the weekly work had students develop central object roles in the various case studies. These tasks were extended with optional extension tasks which were required in order to fully complete the programs described in the case studies. This helped communicate high expectations (P5) with all students being exposed to the full set of requirements and being encouraged to complete more than just the core tasks. As an example, Figure 5.6 shows a UML class diagram from the core tasks for the fourth iteration of the Monopoly case study, which focused on inher-

Table 5.4: Alignment matrix showing staff-planned alignment of case studies to the object oriented programming unit's intended learning outcomes. As with the introductory programming unit, student descriptions of the case study alignment differed based on their individual learning.

Topic	**ILO-1**	**ILO-2**	**ILO-3**	**ILO4**	**ILO5**
First Half					
Case Study 1 - Drawing Program	✓	✓	✓	✓	✓
Case Study 2 - Monopoly	✓	✓	✓	✓	✓
Second Half					
Case Study 3 - RPN Calculator	✓	✓	✓	✓	✓
Case Study 4 - Text-based Adventure	✓	✓	✓	✓	✓

itance and polymorphism. Extension tasks associated with this iteration had students implement a "Jail Tile" and a "Go To Jail Action."

Using such large case studies as the basis for the weekly tasks also helped to encourage students to think in terms of objects. Each case study started with an overall description of the final program, but built up the number of abstractions used each iteration to encourage this focus on object roles, responsibilities, and collaborations (Wirfs-Brock & McKean 2003).

Unit delivery followed the same iterative process as described for the introductory programming unit. Lectures were used to guide student activity, case studies had students apply concepts, weekly formative feedback assisted with student progress, and students incorporated feedback to improve their work and understanding. The following list outlines the similarities between the teaching and learning activities in the two programming units.

- Lectures focused on concepts, with role-plays taking the place of interactive coding demonstrations.
- Case studies formed the basis of the weekly laboratory, core, and optional tasks.
- Weekly tasks were submitted for formative feedback, and students were encouraged to incorporate the feedback they received.
- Staff discussed progress with each students weekly, and their work was signed off as complete after necessary feedback had been incorporated.
- The Programming Help Desk was available to support students outside of scheduled class times.
- Students were encouraged to use online discussion boards, with questions being monitored and answered by teaching staff and fellow students.
- A glossary was used alongside the case studies as a means for students to explain the concepts learnt.

The following list outlines how the object oriented programming unit's teaching and learning activities and delivery met all of the principles stated in Chapter 3.

- Case studies and lectures were designed to actively engage students in the construction of their own knowledge, providing guidance rather than aiming to transfer knowledge to students. (P1)
- Staff planned for lecture topics and case studies to align with the unit's intended learning outcomes, and students reflected on how these activities had aligned from their perspective. (P2)
- Weekly tasks did not carry any weight in the final assessment, with learning

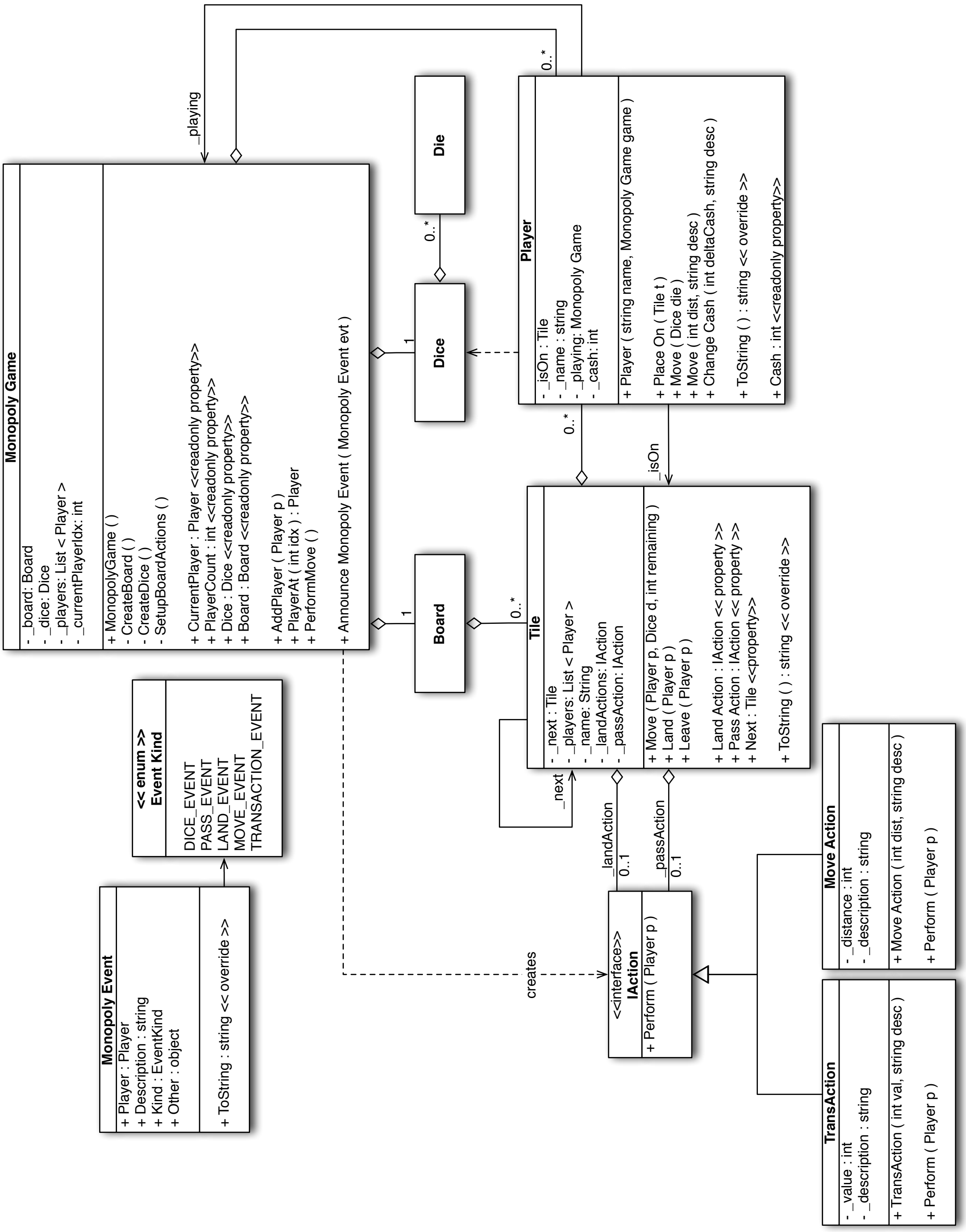


Figure 5.6: An example UML class diagram from the Monopoly case study in the object oriented programming unit.

outcomes being assessed from evidence included in student portfolios. (P3)

- Lectures focused on providing guidance, case studies on applying concepts, and support structures on providing assistance outside of scheduled class times. (P4)
- Case studies, with the associated core and extension tasks, helped communicate high staff expectations. (P5)
- Students were actively supported during and outside of scheduled class times. (P6)
- Students were trusted to manage their own learning, with no marks being associated with weekly work to ensure it remained focused on developing student understanding. (P7)
- Students and staff were encouraged to reflect on teaching and learning activities; students in terms of what they had learnt, and staff in terms of how well the activities had helped students understand the concepts. Staff reflections were used to alter subsequent delivery of the unit, while student were encouraged to incorporate reflections in the learning process. (P8 and P9)
- Concepts related to the object oriented programming paradigm were central to all aspects of the unit. (P10 and P11)
- Students developed object oriented programs using programming languages in ways in which they are intended to be used. (P12)

Choosing an Object Oriented Programming Language

Another interesting aspect of the object oriented programming unit was the decision to allow students to choose which programming language they would use in the unit. By allowing students to choose their language, teaching staff were forced to discuss details at a conceptual level, making the unit clearly about object oriented programming and not the details of one specific language.

While a wide variety of languages could be used in this unit, students were only provided with a choice between four programming languages: C#, C++, Java and Objective-C. Each of these languages had a C-style syntax, helping ease students into the new syntax. C#, Java and Objective-C had similar support for object oriented programming principles, with C++ being sufficient but requiring some extra attention to syntax to achieve the same outcomes. C#, C++ and Java are statically typed, while Objective-C is dynamically typed. All had support from modern integrated development environments, and unit testing tools also required by the unit's intended learning outcomes.

Using multiple languages forced lectures, and laboratory class discussions, to focus on concepts (P11), while empowering students to choose a language they saw as relevant to their future careers (P7). At the same time, the use of multiple languages enabled deeper exploration of concepts through the examination of how different languages implement object oriented features. For example, the difference between static and dynamic typing were discussed in lectures, and demonstrated to show different views on these concepts impact on program design. Similarly, different approaches to memory management were also discussed including manual memory management, reference counting, and garbage collection. In each case, these discussions focused on the general concepts and highlighted the various strengths and weaknesses of the various language implementations, providing a much richer learning experience than would have been possible with a single language.

Once again, the different languages helped strengthen the overall approach. This unit focused on object oriented programming, not on the specifics of a single language. The aim was not for students to develop a depth in *one* programming language, but to learn the principles that underlie *all* object oriented programming languages. Using multiple languages concurrently helped draw attention to this, and beneficially forced teaching staff to communicate at the concept level.

5.3.5 Assessing Student Portfolios

As the assessment criteria were the same, the process of assessing student portfolios was the same as outlined for introductory programming. To pass, students had to satisfactorily complete two hurdle tests and demonstrate sufficient progress with the four case studies. Credit required a glossary that clearly documented the object oriented programming principles, and how they applied to programs the student had created. Distinction required a custom project that demonstrated the application of the object oriented programming principles, while High Distinction involved a research report. As with the introductory programming unit, students aiming for distinction and high distinction were interviewed as part of the assessment process.

Given the process was the same for assessing the object oriented programming unit, as it had been for the introductory programming unit, it too was able to embody all of the principles stated in Chapter 3.

- Portfolios were assessed to determine how well students understood object oriented programming concepts. (P1)
- Students needed to demonstrate how their work aligned with *all* of the unit's intended learning outcomes. (P2)

- Delivery involved an iterative process centred around formative feedback that supported student learning. (P3, P6 and P7)
- Object oriented programming principles and concepts were central to the entire process, including both delivery and assessment. (P4, P10 and P11)
- Case studies and assessment criteria aimed to clearly communicate high expectations. (P5)
- Staff and students were involved in reflective practice, reflecting on past learning experiences to improve outcomes. (P8 and P9)
- Students developed a range of object oriented programs, demonstrating appropriate use of the languages they were expressed in. (P12)

5.3.6 Comparison with the Introductory Programming Unit

Both programming units applied *all* of the principles stated in Chapter 3, and the guidelines from Chapter 4, in delivery and assessment. The following list outlines common features in both units.

- The central role of the learner in constructing their own knowledge was incorporated through activities designed to actively engage students, with portfolios capturing learning outcomes understandings for assessment. (P1)
- Activities and assessment aligned to each unit's intended learning outcomes with *both* staff and students involved in performing this alignment. (P2)
- Assessment aimed to evaluate learning outcomes, with formative feedback being used throughout the delivery of the unit. (P3)
- Activities were designed to focus on actively engaging students as appropriate for the various class types used. (P4)
- High staff expectations were communicated to encourage students to aim for excellence. (P5)
- Students were actively supported both in class activities, and between scheduled classes. (P6)
- Students were trusted and empowered to manage their own learning, with flexibility in how outcomes were met. (P7)
- Activities and resources were organised to encourage change to better meet student needs. (P8)
- Staff and students reflected upon learning outcomes; staff to incorporate change between semesters, and students in addressing weekly feedback and in preparation of their portfolios. (P9)
- Each unit focused on a given programming paradigm, introducing students to structured procedural programming and object oriented programming paradigms.

(P10)

- Concepts were central to each unit, with each week's activities building up understanding developed in previous week's topics. (P11)
- Both units used programming languages in the ways they were intended, ensuring the chosen languages were appropriate for each unit. (P12)

Where the units differed was in the requirements expressed in the unit's intended learning outcomes. The central nature of these in constrictive alignment meant that altered intended learning outcomes impacted on the kinds of activities used, and expectations of what needed to be included in student portfolios. These example units help to demonstrate the general applicability of the student centred approach to constructive alignment described in Chapter 4, with each unit following the processes and guidelines presented. They also help to demonstrate the versatility of the portfolio assessment approach, and applicability of the SOLO taxonomy, with both units able to use the same assessment process and criteria.

5.4 Summary

This chapter provided two example implementations, in Section 5.2 and Section 5.3, of the model described in Chapter 4. In each case the unit was developed and delivered using the processes discussed in Chapter 4, and was guided by the principles from Chapter 3. The described units have been delivery successfully across a number of teaching periods, and Chapter 7 reports on the evaluation of these units while Chapter 8 discusses the relative importance of the various aspects of the approach overall.

In the following Chapter 6 we describe the teaching and learning resources that supported the delivery of these units.

6

Supporting the Curriculum with Tools and Technologies

In addressing the need to be agile and willing to change, Principle 8, the model from Chapter 4 proposed the separation of teaching and learning activities from resources used to support these activities. This separation aimed to enable greater flexibility in the activities, which were then supported by more generally targeted resources.

Chapter 5 outlined the teaching and learning activities for two example implementations of the proposed model from Chapter 4. The teaching and learning activities discussed in Chapter 5 were supported by a number of tools and resources, and these resources are the topic of this chapter.

Resources described in this chapter include tools to support both *how*, and *what*, we teach. Section 6.1 describes an online tool used to support the teaching and learning environment, providing support for *how* we teach the units. Section 6.3, Section 6.4 and Section 6.2 describe three different resources used to support *what* we teach in the units. Section 6.3 describes a programming text that was used to support the concept-based approach, providing students with details in order to support the concept focused lectures. These details were also provided as a series of video podcasts on language syntax, as outlined in Section 6.4. Providing general support for what we teach, Section 6.2 describes a game development framework used to enable students to create more engaging programs.

6.1 Visualising Task Progress to Support Formative Feedback

Formative feedback (P3) plays an important role in shaping the teaching and learning environment for the units implemented using the model presented in Chapter 4. The emphasis on iterative feedback helps support the focus on students active construction of their own knowledge (P1) but provides an additional challenge as students no longer have marks to influence their behaviour.

One risk of this greater flexibility is the potential for students to underestimate the amount of work required to satisfactorily complete all of the unit's core tasks within the scheduled time frame. As described in Section 4.1.3, these core tasks help the students develop pieces for their portfolio and, ideally, students should complete all of these tasks during the teaching period.

Student engagement in the iterative delivery process, discussed in Section 4.2.5 and Section 5.2.5, is integral to the underlying principles (P1, P3, P6, and P7). To help encourage students to remain actively engaged with the unit, a task tracking tool was developed. This tool, named "Doubtfire", is outlined in this section, which expands on previous publications related to the tool (Cain et al. 2013, Woodward et al. 2013). Section 6.1.1 outlines the requirements for Doubtfire, with Section 6.1.2 describing the approach taken to address these requirements. A brief analysis of the ability of the Doubtfire tool to meet its requirements is then provided in Section 6.1.3, with further analysis being presented in Chapter 7. The section concludes with a short summary in Section 6.1.4.

6.1.1 Requirements

Doubtfire's primary requirement was to provide students with a means of tracking their progress on a unit's tasks. To achieve this it was decided to adapt burn down charts from the Scrum agile software development process (Schwaber & Beedle 2002). Using this approach, each chart shows the cumulative amount of work remaining for week-by-week, which decreases as work is complete, or "burns down" over time.

Agile software development methods (Beck et al. 2001) embrace change (Beck 2000) by specifically allowing for adaptive and periodic adjustment of activities. The basis for such adaptation in Scrum is *empirical* information; a consistent measure of the work remaining ("backlog"), and a measure of the rate work is being completed by the team ("velocity").

The purpose of a burn down chart is to allow stakeholders to consider the velocity of the team with respect to the current backlog. This chart acts as a "information radiator" (Cockburn 2002) for the team, providing details on either release or "Sprint" iteration schedules. Since quality of work should not be compromised, the requirements (backlog) of work can be adjusted in order to meet the required schedule and cost (Sutherland & Schwaber 2007).

In adapting burn down charts as a tool for supporting students engagement with formative assessment tasks, the assessment tasks become the "backlog." Students are then able to see the number of tasks remaining, and use their current "velocity" to determine if their progress is sufficient to complete the unit on time. The core tasks represent the minimal set of activities that students should complete by the end of the unit, but additional tasks – such as the extension tasks, custom project, and research report – can be *acquired* if velocity permits.

Burn down charts provide students with a visual representation of the tasks they need to complete, the number of tasks, the scheduled task due dates, and estimated effort. Students should be able to use the tool to assess their progress, and to determine whether they need to increase their rate of progress (velocity). If so, they can commit more time or take greater advantage of the support resources available.

To assist in providing students with support in their learning (P6), it was decided to extend the scrum-style Boolean marking of tasks as completed, to allow students to indicate if they were *working on*, or *having trouble* with, particular tasks. This requirement aimed to increase student engagement with the tool, and improve likelihood that students would make active use of the tool throughout the teaching period.

Gamification (Deterding et al. 2011) provided a second inspiration for encouraging students to further engage with the tool. Badges could be awarded to students for appropriately engaging with the tool, providing them with further motivation to engage appropriately with unit content. Badges could be awarded for completing tasks on time, and for indicating tasks students are working on or having issues with. Encouragement could also be provided for persisting with formative tasks, and getting work signed off after revisions are incorporated.

Task heterogeneity required staff to be able to set a specific *weight* for each task. This weight represents the estimated effort students needed to expend to satisfactorily complete the task, based on staff opinion and historical data. Rather than specifying task weight in terms of hours, this was done in a more abstract unit. One popular approach within agile software development is to assign tasks "*t-shirt size*" weights

(Peixoto et al. 2010). Using this approach, task weight is set to a common t-shirt size: *extra small*, *small*, *medium*, *large*, *extra large* etcetera. The t-shirt sizes are then allocated weights, with each increment in size doubling the associated weight: *extra small* had a weight of one, *small* a weight of two, *medium* four, etcetera.

Task weights needed to be incorporated into the burn down chart, with each chart showing the cumulative number of *task-points* remaining. Using task-points in the burn down chart enables it to visually show weeks where more, or less, effort is likely to be required.

Progress also needed to be projected to indicate an expected completion date. This projection can be calculated from the average number of task-points the student completes each week, their *velocity*. For example, if six task-points were completed in one week, based on the velocity, a thirty-six task-point project is expected to be completed in six weeks. Each student's projected completion needed to be recalculated as time progresses.

To aid with assessment, Doubtfire's requirement also included the ability for teaching staff to sign tasks off as *complete*. This could then be used by teaching staff to indicate that the individual student had demonstrated satisfactory knowledge in the associated concepts. Using the tool in this way also supports the rapid assessment of student portfolios, as work marked as *complete* demonstrates Pass/Credit understanding, and would not need to be checked thoroughly in final portfolio submissions.

As an interactive system, Doubtfire had a number of requirements to ensure that it could be best utilised by all targeted users. Our aim was to create a tool that was simple and appealing for students to use and was easily accessible from a range of devices and locations. Students not should feel that interaction with the tool is difficult or additional "work." The following requirements were identified:

- **Online**: Students needed to be able to easily access the tool both in and out of scheduled class times. It was decided to make Doubtfire an online tool, thereby making it accessible from virtually anywhere. It also simplified the development progress with only web platforms needing to be supported, and avoided the need for students to install client software.
- **Easy to use**: The tool needed to be simple to use; with good usability to reduce barriers for student adoption.
- **Mobile friendly**: Staff and students needed to be able to quickly check, and update, their progress from a range of devices. It was thought that by ensuring Doubtfire could be easily accessed via a mobile device it would encourage

students to access the tool even when they were away from desktop computers.

- **Aesthetically pleasing user interface**: To encourage adoption of the tool among students, a visually appealing user interface was desirable.

Doubtfire also needed to addressed the requirements of teaching staff. Tutors are responsible for managing classes, and therefore need to be able to respond to student actions. Convenors have overall responsibility for the unit, and need to be able to observe the performance of the student cohort and perform simple administrative actions. All teaching staff benefited from the requirements listed, with the mobile nature allowing tutors to easily check and update student progress during scheduled classes.

In terms of the development and deployment of the tool, a number of software qualities were desirable, including:

- **Supporting the teaching environment**: The tool should play a supportive role, and should therefore fit inside the teaching environment; it should not be necessary to fit the teaching environment around the tool.
- **Quick to develop and extend**: It should be easy to add new features to the tool, and adapt existing features, to ensure it remains relevant.
- **Controllable**: The schema that defines the way tutors and students interact over tasks must be easy to alter to allow for adaptation if assessment criteria change.

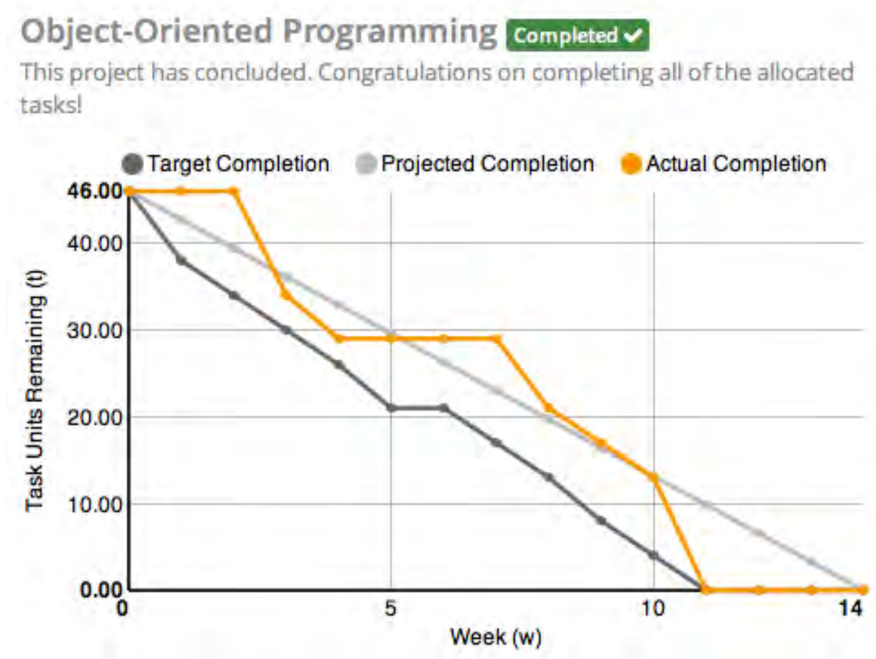
6.1.2 Doubtfire Solution

Features

Doubtfire allows each student to track their progress against a unit's core tasks using a "burn down chart" as shown in Figure 6.1. The burn down chart consists of three lines, shown in Figure 6.1. The lines indicate:

- **Actual Completion**: Shows the number of task-points signed off for the student by week.
- **Target Completion**: Shows the recommended schedule from the task due dates set by the convenor.
- **Projected Completion**: Indicates the current velocity, which is then projected to indicate an expected end date if current velocity is maintained.

The following list provides an overview of Doubtfire's current features. Each of these features is more fully described in the following section, which describes the features



Object-Oriented Programming Completed ✓

This project has concluded. Congratulations on completing all of the allocated tasks!

Figure 6.1: An example burn down chart showing progress against weekly tasks

in relation to the associated user roles.

- Individual student progress can be monitored using a Burn Down chart
- Collectively, student progress can be viewed in terms of six statuses: *ahead*, *on track*, *behind*, *in danger*, *doomed*,[1] and *haven't started*.
- Units, their tasks and students, can be administered allowing tasks and student to be added and removed.
- Task status can be updated in response to student work, and staff feedback.
- The tool can be accessed using modern web browser, and provides an adaptable user interface, which caters for both mobile and desktop access.
- Role based access, providing appropriate interfaces and actions for teaching staff and students.

User Roles

Doubtfire provides functionality for three distinct user roles: Convenor, Tutor and Student. Each of these roles has access to a different set of features, as described in Table 6.1. Convenor and Tutor roles support teaching staff, with the students having a separate role.

[1] The "doomed" status was a wordplay related to the classic "Doom" 3D shooter game, and the notion of incorporating gamification ideas. This status was only visible to staff.

Table 6.1: Available features for each user group in Doubtfire

Role	Features
Convenor	**Unit Administration**: Includes the ability to enrol students and create tasks. **Monitor Student Progress**: View distribution of students by progress indicators. See Figure 6.2. **Monitor Task Progress**: View progress distribution for each unit's tasks. See Figure 6.3. **Examine Student Progress**: View student list showing status for each task. See Figure 6.4. **Update Task Status**: Mark student work as complete.
Tutor	**Examine Student Progress**: View student list showing status for each task. See Figure 6.4. **Update Task Status**: Mark student work as complete.
Student	**Update Task Status**: Mark student work as complete. See Figure 6.6. **Monitor Progress**: View progress on task completion using a burn down chart showing Target, Actual and Projected completion.

Unit convenors are responsible for the overall delivery and management of the unit. To support this role, Doubtfire provides convenors with tools to set up tasks and enrol students. A dashboard provides an overview of student progress, and enables quick access to views of student progress by task, and to individual students via scheduled classes. Figure 6.2 shows an example convenor dashboard overview of students progress within a unit. Figure 6.3 shows an example chart that visualises the distribution of student status for each task. In addition to these tasks, Convenors also have the ability to perform the same actions as Tutors.

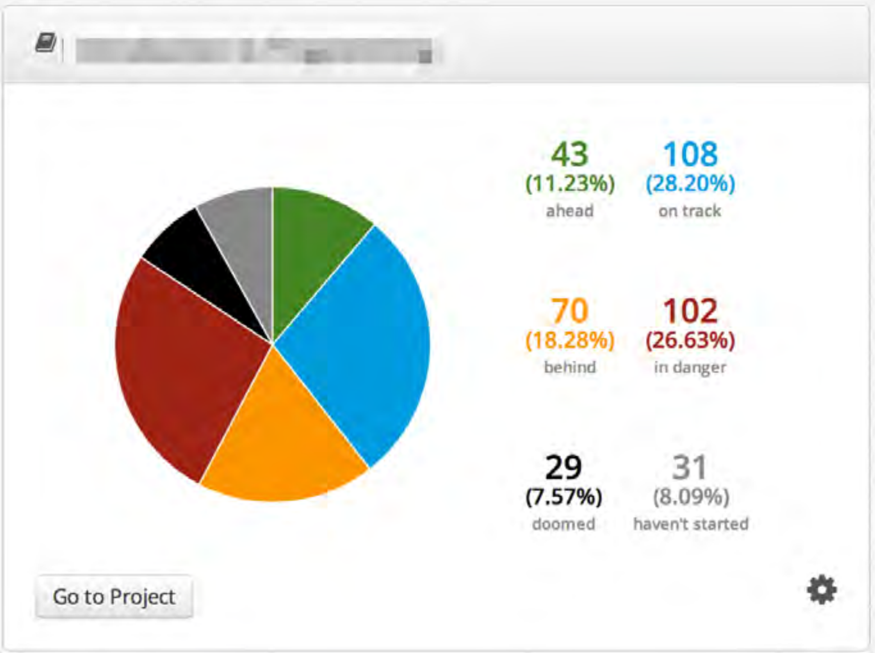


Figure 6.2: Overview of progress by unit from the Convenor Dashboard showing indicators of student progress

Tutors are responsible for conducting the tutorial classes, and providing formative feedback to the students. To support this role, Doubtfire provides tutors with a class-list view showing student progress. From the class list, tutors can drill down to view individual students and their burn down charts. It also provides a convenient means for tutors to update the status of student tasks.

Figure 6.4 shows an example of the class list used by Tutors to view student details and update task status. Each task is represented by a coloured rectangle that indicates the task's current status for that student. Tutors are able to update the status of a student's tasks directly from the class list by selecting the task, and updating it from a list of possible options, also shown in Figure 6.4.

Figure 6.5 shows an example of the dashboard provided to students, which shows their progress for the units they are enrolled in. For each unit, students can view their

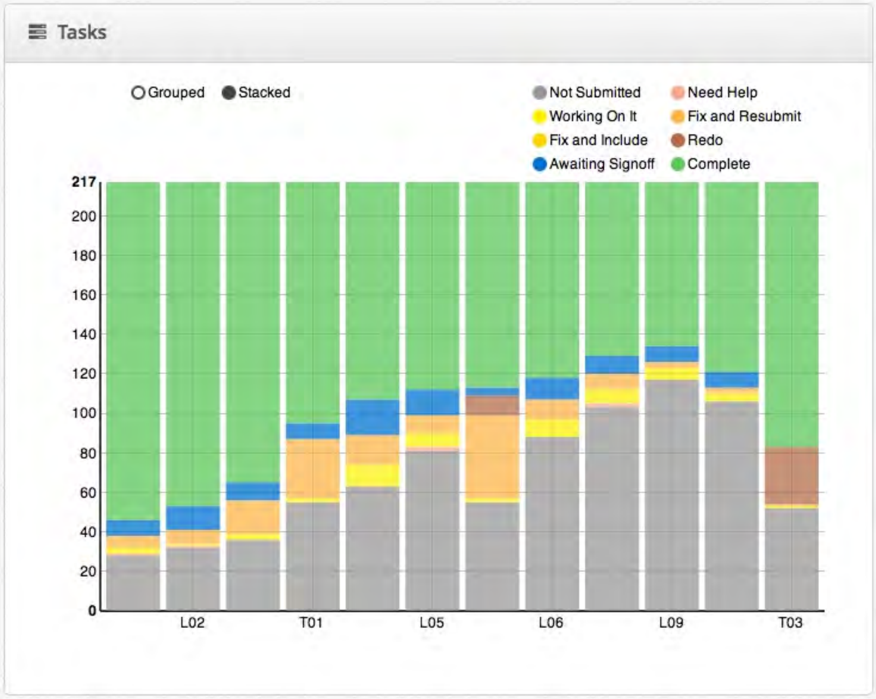
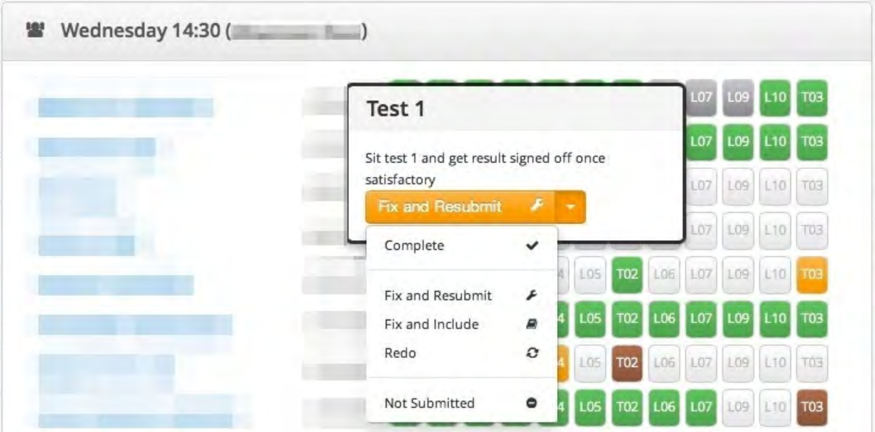


Figure 6.3: Convenor view showing distribution of student status by task, bars can be stacked as shown or grouped by task status.

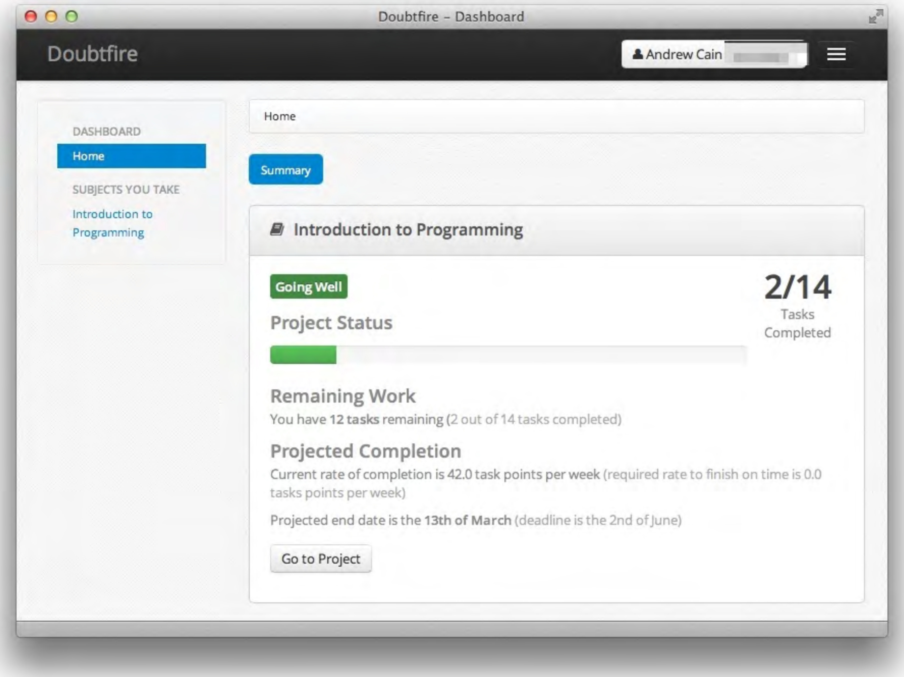


Test 1

Sit test 1 and get result signed off once satisfactory

Figure 6.4: Tutor view of class group, and adjustment of task status. Student names and id numbers have been obscured.

burn down chart and each task's status. Students can update the status of a task by selecting it in the task last, and choosing a new status as shown in Figure 6.6.



Remaining Work

You have **12 tasks** remaining (2 out of 14 tasks completed)

Projected Completion

Current rate of completion is **42.0 task points per week** (required rate to finish on time is 0.0 tasks points per week)

Projected end date is the **13th of March** (deadline is the 2nd of June)

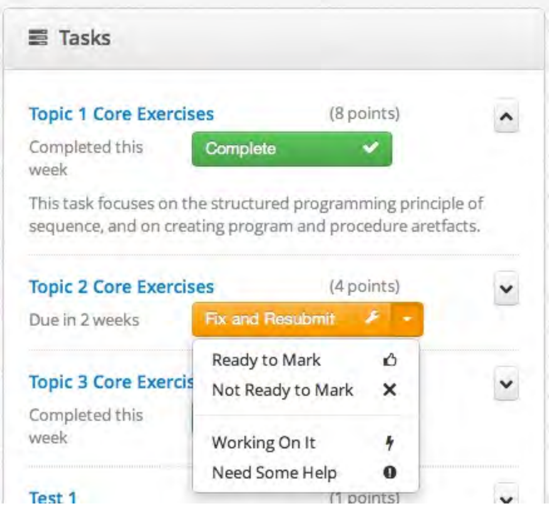
Figure 6.5: Student dashboard in Doubtfire showing personal progress for each enrolled unit using the tool

Task Processes

Tasks in the Doubtfire system have one of a number of states, with different users being responsible for updating task status at various stages during unit delivery. The main states and the transitions between these is shown in Figure 6.7 as a UML State Chart (OMG 2011), with additional annotations to indicate user roles reponsible for performing the highlighted transitions.

Initially all tasks are set to the *Not Started* status. When students begin work on the task they are encouraged to update its status to *Progressing*, and when it is ready for assessment to the *Ready to Mark* status. Once tutors receive the work, they examine the work and provide the student with formative feedback. After having discussed the task with the student, the tutor updates the status by either returning it to a *Progressing* state if the task needs to be fixed, or updating the task as *Complete*.

The *Progressing* status was divided into a number of sub-states for the purpose of



Tasks

Topic 1 Core Exercises (8 points)

Completed this week

Complete ✓

This task focuses on the structured programming principle of sequence, and on creating program and procedure aretfacts.

Topic 2 Core Exercises (4 points)

Due in 2 weeks

Fix and Resubmit

Ready to Mark

Not Ready to Mark ✕

Working On It

Need Some Help

Topic 3 Core Exercis

Completed this week

Test 1

(1 points)

Figure 6.6: The "Tasks" list enables students to view and change task status

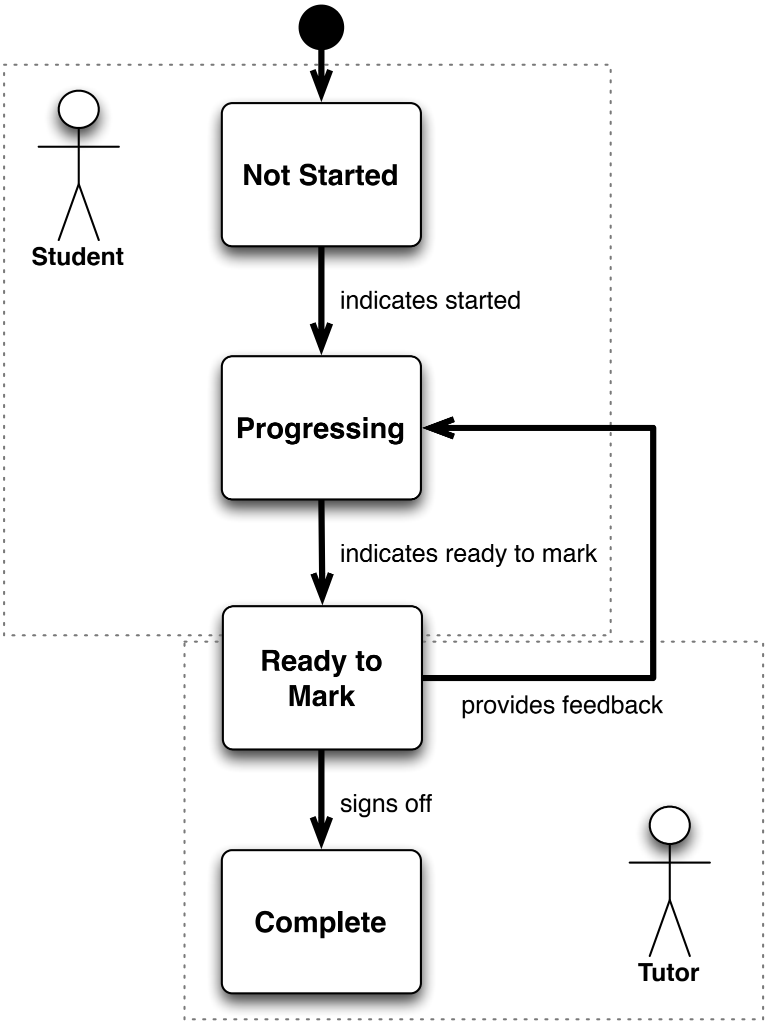


Figure 6.7: UML state chart showing task states and transitions, and Tutor or Student roles associated with performing these transitions.

providing students with finer-grained feedback.

Students were able to set the status of a task to *Working on It* or *Needs Help* to indicate their current progress on the task to their tutor and to the unit convenor. *Fix* and *Redo* statuses could be used by students to indicate that tasks needed some aspects adjusted (the *Fix* status) or that it should be redone (the *Redo* status). These status, shown in Figure 6.8, were designed to help provide more accurate details of progress for both staff and students. Students indicated how they were progressing with the tasks, and staff could provide feedback on the outcomes students had achieved.

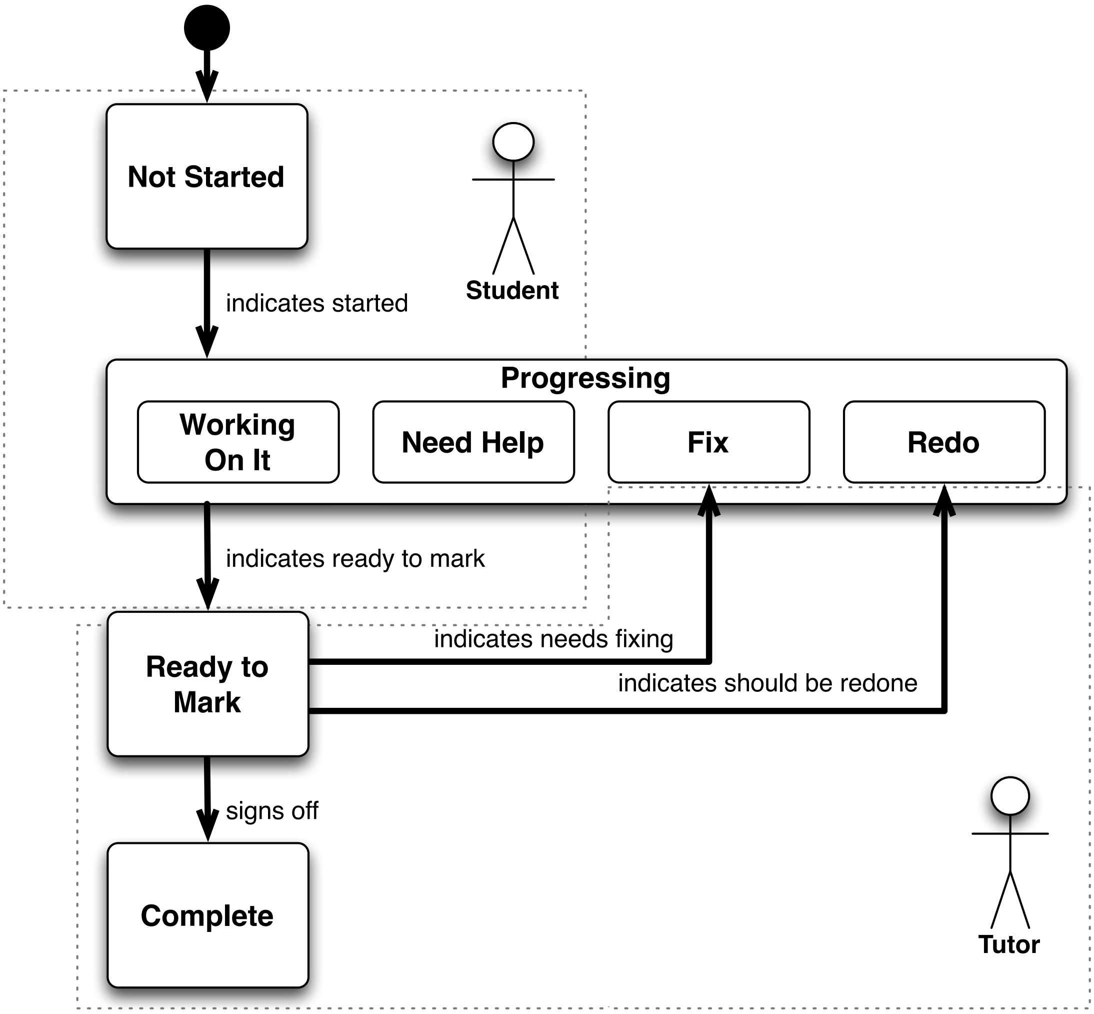


Figure 6.8: UML state chart showing the detailed states within the Progressing state

Architecture and Implementation

Figure 6.9 provides an overview of the main components of the Doubtfire system. The core of Doubtfire was implemented using Ruby on Rails (Ruby et al. 2013), with a RESTful API (Richardson & Ruby 2007). This encapsulated the core entities – units, users, and tasks – and their associated processing. Data is persisted to a MySQL (MySQL 2013) database, while an LDAP (Sermersheim 2006) compliant directory server is accessed to provide authentication against the university wide data store.

On the front end, the dynamic nature of the site is achieved though the use of a number of libraries, backed primarily by the jQuery (jQuery 2013) JavaScript library. D3.js (D3.js 2013), and NVD3 (NVD3 2013), are used to provide all of the charts, including the burn down charts. The visual style, and layout, of the website is achieved through use of Twitter Bootstrap (Twitter Bootstrap 2013).

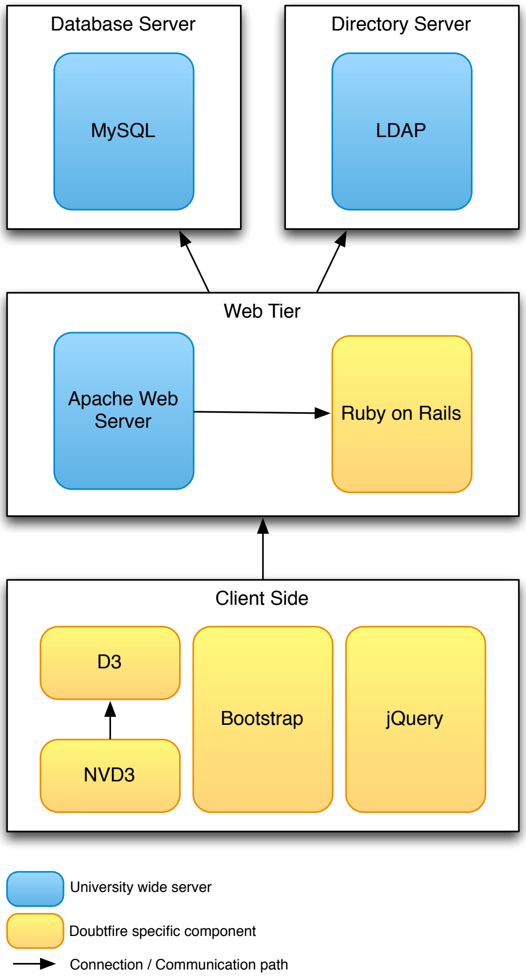


Figure 6.9: Overview of main software components in Doubtfire's implementation

6.1.3 Use and Evaluation of Doubtfire

While Chapter 7 provides a more in-depth discussion of the use of Doubtfire, this section briefly comments on how it was used in the delivery of the introductory programming and object oriented programming units, and discusses how well its implementation met its requirements.

Doubtfire has been successfully used in multiple iterations of the programming units described in Chapter 5. In each case, the core tasks from the teaching and learning activities were used as the tasks to be completed, and teaching staff assigned each a t-shirt style weighting to represent its relative size. During unit delivery, students and staff tracked progress against these tasks, with work being signed off by the tutors once complete.

Analysis of student reflections indicated that the effectiveness of the tool, for students, depended on their level of engagement with the unit. Engaged students made active use of Doubtfire, and responded quickly to addressing issues and closing gaps in their knowledge. Students who struggled to complete the weekly tasks generally made poor use of the tool at the start of the teaching period, but engaged actively later in the semester. While some, characteristically disengaged, students avoided use of the system and attempted to establish progress in their own way.

For teaching staff, the Doubtfire tool provided useful information on how students were engaging with the formative process. It was easy to see which students were doing well, to identify those who were falling behind, and those who were not engaging with the unit at all. This information was used to prompt students, encourage those who were doing well and suggest appropriate resources for those who were struggling or falling behind.

In terms of its requirements, Doubtfire was felt to meet its core requirement of providing students with a means of viewing their task progress. The following list outlines the requirements that Doubtfire has met, and those that are currently on the backlog to be implemented in future iterations.

- Requirements met:
  - Staff and students are able to track progress against unit tasks.
  - Heterogeneous tasks are supported with task weights.
  - Tasks support a range of states, encouraging students to engage with the system beyond marking work as complete.
  - Students and staff are able to update task progress based upon their roles.

  - Student burn down charts show target, projected, and actual completion lines to help indicate likely end dates if current velocity is maintained.
  - Doubtfire is visually appealing, easy to use, and mobile friendly.
- Requirements not implemented:
  - Optional tasks can be entered into the system, but cannot be *acquired* by students.
  - Gamification ideas were not implement.
  - Task overviews are provided to staff, but finer-grain detail requires data to be manually exported from the database.

In relation to the principles from Chapter 3, approach from Chapter 4, and example units from Chapter 5, Doubtfire provided the following contributions:

- Doubtfire supported the principles by providing:
  - Staff and students with progress data on formative tasks. (P3)
  - Staff will details to help support student learning. (P6)
  - Staff with a means of verifying students had completed tasks, without having to revert to marks for motivation. (P7)
  - Staff with evidence of student progress that can be used to inform future adjustments to unit delivery. (P8 and P9)
  - Students with progress details they can reflect upon in their Learning Summary Reports. (P9)
- Doubtfire supported the approach, and example units by:
  - Encouraging students to engage in the formative feedback process.
  - Providing evidence that students had completed core tasks.
  - Enabling staff to identify students who needed additional help and encouragement.

6.1.4 Summary

The strong use of formative feedback in the model provides a challenge as students cannot be motivated to work by the fear of losing marks during the teaching period. This can lead to students allocating insufficient time to complete learning activities, resulting in them falling behind in the unit.

Doubtfire was designed to address these concerns through the use of agile development burn down charts that visually represented the amount of work students had remaining in the unit. By using this tool staff and students were able to monitor progress throughout the teaching period.

6.2 A Game Library to Support Procedures First

Chapter 5 described two programming units implemented using the approach described in Chapter 4, and principles stated in Chapter 3. Following Principle 10, these units were each centred around concepts associated with a single programming paradigm. As described in Section 5.1, an objects-later approach was adopted for the introductory programming unit, which focused on introducing students to the procedural programming paradigm, with the subsequent object oriented programming unit introducing the object oriented paradigm.

Principle 11 indicates the desire to structure the programming curriculum around programming concepts in such a way that each topic builds upon prior knowledge. This lead to the procedures first approach for the introductory programming unit described in Section 5.2.4.

With the procedures first approach to teaching introductory programming, students are introduced to calling and creating procedures before being introduced to other programming concepts. This approach promotes a focus on **sequence** in these early tasks, with students creating procedures to group together a sequence of procedure calls that perform a certain task.

The challenge with this approach, as identified by Pattis (1993), is finding meaningful tasks for students to perform. Standard programming language features do not provide sufficient functionality for student to perform meaningful, and engaging, actions without having to use a wider range of programming constructs.

This section describes SwinGame (SwinGame 2013) a game development library designed to facilitate a procedure first introduction to programming, and to help students create more interesting programs. Section 6.2.1 outlines the requirements for the SwinGame library, which is then described at a high level in Section 6.2.2. Section 6.2.3 describes how SwinGame was used to support the teaching of introductory programming, and relates this to the principles, approach, and units from chapters 3 to 5. The discussion of SwinGame concluded with a short summar in Section 6.2.4

6.2.1 Requirements

SwinGame was created to help teaching the introductory programming unit described in Chapter 5. The main goal for SwinGame was to provide students with resources to enables them to create more engaging programs, while also providing support for a procedures-first approach to the programming curriculum.

The requirements for SwinGame were to:

- Provide functions, procedures, and custom types to enable the creation of small two dimensional games. Including:
  - Resource management for images, sounds, animations, and maps.
  - Drawing operations to draw primitive shapes, images, and text.
  - Sprite management, enabling the creation of movable, animated, game entities.
  - Ability to play sound effects and music.
  - Collision detection operations including collisions of geometric shapes, and pixel level image collisions (image-image, and image-shape collisions).
  - Support for two-dimensional game physics.
  - Input handling routines to support keyboard, mouse, touch, accelerometer, and gyroscope input.
  - Time tracking, and management.
  - Camera support, enabling game space coordinates to be mapped to screen coordinates for drawing.
  - Network support to enable peer-to-peer interactions, as well as enabling http requests to get/post high score details to web servers, for example.
- Give the programmer full control over the game's actions, requiring explicit requests to perform any task.
- Enable developers to develop their programs on a number of platforms: Linux, MacOS, and Windows.
- Enable programs to be run on a number of devices: Desktop computers, tablets, and smart phones.
- Be implemented in such a way that it can be developed and extended by students.
- Provide access to the functionality across a range of programming languages, ensuring that each language provides appropriate programming abstractions to ensure that the library can be used appropriately. (P12)

6.2.2 SwinGame Solution

Figure 6.10 provides an overview of the components involved in the use of development of the SwinGame library. The library itself, provides students with an interface that, to procedural programming languages, exposes a range of functions and procedures that can be called to perform required actions. The logic of SwinGame itself is divided into *Core Logic*, and *Back-end Logic*. The components in the Core Logic implement the higher level game engine mechanics, while the components in the Back-end Logic provide a consistent interface to lower level third party components.

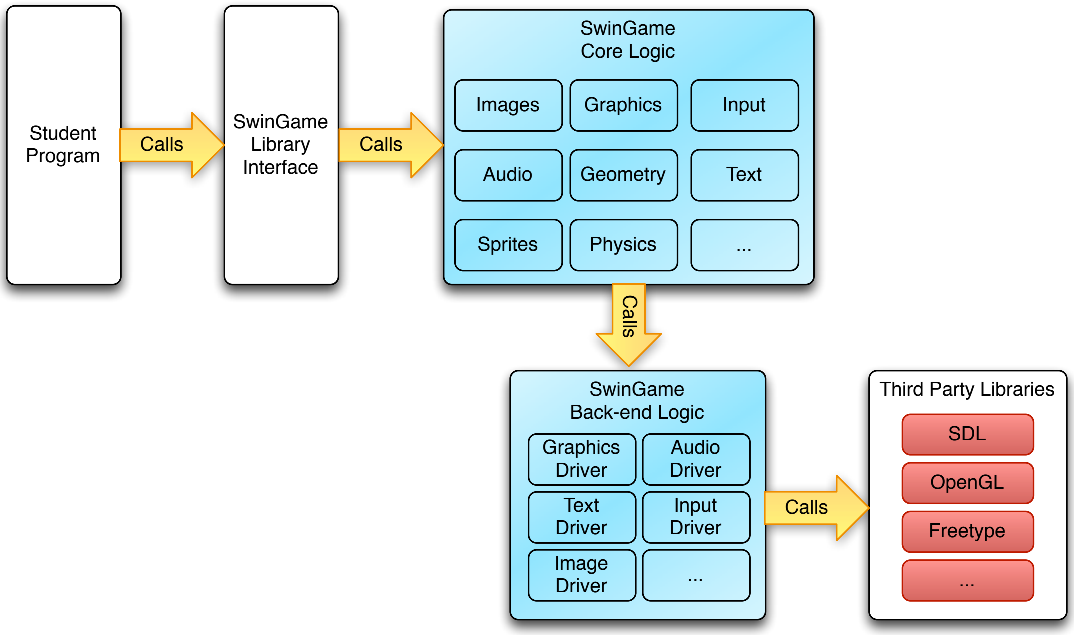


Figure 6.10: Overview of the components in the SwinGame library, their connections and organisation.

The division of SwinGame, into Core Logic and Back-end Logic, decouples the core game engine logic from the lower level libraries used. This enables the library to update Back-end components with minimal impact on the core logic. Currently SwinGame supports a number of different back-end components used to support different platforms and underlying third party libraries.

SwinGame provides users with a number of components and features. The main components are briefly outlined in the following list.

- **Animations**: provides the ability to load and play cell based animations.
- **Audio**: supports loading and playing of sound effects and music.
- **Camera**: enables a virtual camera to be moved around the game world – adjust-

ing what is shown by mapping game to screen coordinates.
- **Geometry**: provides mathematical operations to manipulate geometric shapes.
- **Graphics**: enabling a window to be opened and providing functions to draw geometric shapes.
- **Images**: supports loading and drawing bitmap images.
- **Input**: supports keyboard, touch, and accelerometer input.
- **Networking**: provides the ability to create and use network connections.
- **Physics**: provides functions to perform collisions between entities.
- **Resources**: supports the management of image and sound resources, mapping names to resources.
- **Sprites**: enables the creation of image based sprites.
- **Text**: supports font loading, and text rendering.
- **Timers**: provides access to components to track and manage time based actions.
- **User Interface**: enables creation of user interfaces including labels, text boxes, lists, buttons, and other components.
- **Utilities**: provides other miscellaneous operations useful for game development.

6.2.3 Use and Evaluation of SwinGame

Supporting Early Exercises and Lecture Demonstrations

SwinGame was used to support the early exercises in the teaching and learning activities for the introductory programming unit described in Chapter 5. For example, procedures were introduced in Week 1 where students developed a small program that drew a house, the resulting code from this exercise is shown in Listing 6.11. This task aims to focus students attention on concepts related to procedure declarations, procedure calls and instruction sequence. Sequence was explored by adjusting the order of the instructions in the program, and examining the results on the images drawn. Core exercises in Week 1 then built on this, having students complete a program that used images and sound effects to deliver a joke, the starting code for which is shown in Listing 6.12. Screen-shots of the resulting programs are shown in Figure 6.13.

With the students in full control of their programs, any form of user interaction required students to implement their own event handling loop. This provided a convenient motivation for control flow mechanisms in Week 3 of introductory programming. By this point, students could create parametrised procedures to draw shapes, but program duration was always set by the length of the delay coded into the se-

Pascal

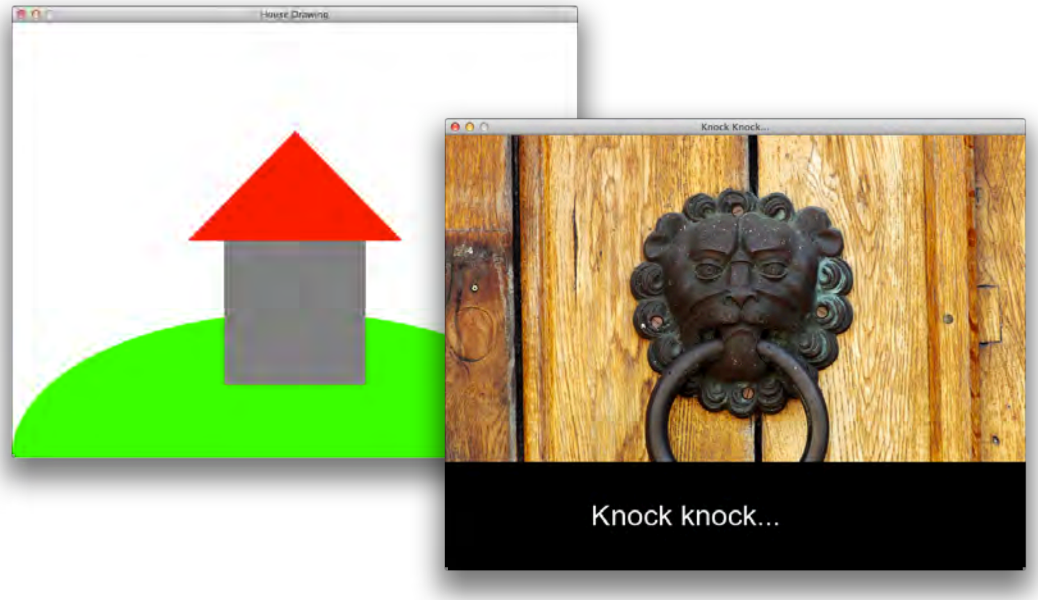
```
program HouseDrawing;
uses SwinGame;

procedure DrawBackground();
begin
  ClearScreen(ColorWhite);
  FillEllipse(ColorGreen, 0, 400, 800, 400);
end;

procedure DrawHouse();
begin
  FillRectangle(ColorGrey, 300, 300, 200, 200);
  FillTriangle(ColorRed, 250, 300, 400, 150, 550, 300);
end;

procedure Main();
begin
    OpenGraphicsWindow('House Drawing', 800, 600);

    DrawBackground();
    DrawHouse();

    RefreshScreen();
    Delay(5000);
end;

begin
  Main();
end.
```

Figure 6.11: The Pascal code for the House Drawing laboratory exercise from the introductory programming unit. In this program students explored concepts related to procedures and sequence.

Pascal

```pascal
program KnockKnock;
uses SwinGame;

procedure LoadResources();
begin
  LoadBitmapNamed('door', 'KnockKnock.jpg');
  LoadSoundEffectNamed('knock', 'door-knock-3.wav');
  LoadFontNamed('joke font', 'Action Man.ttf', 48)
end;

procedure DrawDoor();
begin
  DrawBitmap('door', 0, 0);
  RefreshScreen();
  Delay(2000);
end;

procedure ClearAreaForText(); ...

procedure ShowKnockKnock();
begin
  ClearAreaForText();
  PlaySoundEffect('knock');
  DrawText('Knock knock...', ColorWhite, 'joke font', 200,500);
  RefreshScreen();
  Delay(2000)
end;

procedure Main(); ...

begin
  Main();
end.
```

Figure 6.12: The core exercise in Week 1 of the introductory programming unit had students complete a program that told a joke. This included code to draw images, play sound effects and draw text.



Knock knock...

Figure 6.13: The programs created in Week 1 of the introductory programming unit

quence. By introducing **repetition** it became possible to keep a window open *until* the user asked for it to close. With **selection** user actions could then be responded to, updating values in variables that changed how things were drawn on the screen.

The visual nature of the games developed with SwinGame help create a more engaging atmosphere in lectures. By creating visual programs, the lecture demonstration code can be designed and implemented *with* the students. For example, in Week 3 of the introductory programming unit the lecture demonstration created a small program where the user could move a light around the screen and turn it on and off. The code in Listing 6.14 shows a part of the code developed in this lecture. The program was created iteratively, involving the students at each stage, as outlined in the following list.

1. Initially the program was developed using concepts from previous week, with the code implementing the `DrawLight` procedure. At this stage a call to `Delay` was used to keep the program open for a short period. In this way the example helps build upon student's prior knowledge. (P11)
2. The first problem was highlighted by explaining how the program worked, while it was running. Half way through the explanation the program ended, leading to the question "How can we keep the program open until we want it to close?". Via various prompts the event loop was described, and coded as the repeat loop in main. Counting out a period longer than the previous delay highlighted that the new code had solved the old problem, and the class discussed what was ac-

tually happening behind the scenes to enable this. Closing the window demonstrated the ending of the repeat loop.

3. The next problem was to make the program more interactive: "Wouldn't it be good to be able to turn the light on and off?". This lead to a discussion of what needed to *vary* in the program, and the addition of the `lightIsOn` variable in `Main`. This was set to false when the program started, and its state was flipped (on to off, and visa versa) in the repeat loop.
4. Running the program had an *interesting* effect, as the light flickered between its two states. Questions started with "Do we always want to change the state of the light?", and eventually lead to "So, you only want to change the state of the light **if** the user has typed the space bar?" and follow on to "How can we achieve this in our code?". After reviewing selection, and the syntax for the if statement, the assignment statement that changed the lights state was put within an if statement. The program was executed and the results discussed.

Other features were added in a similar style, each focusing on the application of the concepts to solve a problem or to introduce a new feature. The exercise demonstrates the application of constructive learning theories (P1) , in that it aims to help guild students in the construction of their knowledge. The example starts at a point they should be familiar with, and identifies ways in which the new knowledge, control flow in this case, can enhance the functionality of the program they are creating (P11). In this way, the examples demonstrate appropriate applications of the new concepts, helping students work toward the goal of thinking and acting as experts.

Each week's lecture demonstrations followed a similar sequence. SwinGame enabled the focus on programming concepts (P11) due to its requirement for explicit control, while its support for multimedia resources helped make the programs more "fun." Later week's lecture demonstrations continued to expand on concepts learnt, and culminated in the development of a small game. The functionality and theme of these games were proposed by students, helping them take ownership of the learning activities (P7). Students supplied images and sound effects, suggested features, discussed implementation strategies and were engaged in the iterative implementation of these games. Figure 6.15 shows screen-shots of two games developed in the lectures, the code of which was then shared with the students.

Supporting Custom Projects

As well as supporting the delivery of lecture material, SwinGame provided students with a wide range of capabilities they could use in the creation of their *custom projects*.

Pascal

```pascal
program GameMain;
uses SwinGame, sgTypes;

procedure LoadResources();
begin
  LoadBitmapNamed('on', 'on.png');
  LoadBitmapNamed('off', 'off.png');
end;

procedure DrawLight(isOn: Boolean; x, y: Integer);
begin
  if isOn then
    DrawBitmap('on', x, y)
  else
    DrawBitmap('off', x, y);
end;

procedure Main();
var
  lightX, lightY: Integer;
  lightIsOn: Boolean;
begin
  lightX := 10;
  lightY := 10;
  lightIsOn := False;

  OpenGraphicsWindow('Light Switch', 800, 600);
  LoadDefaultColors();
  LoadResources();

  repeat
    ProcessEvents();

    if KeyTyped(vk_Space) or MouseClicked(LeftButton) then
      lightIsOn := not lightIsOn;

    if KeyDown(vk_Left) then lightX := lightX - 1;
    if KeyDown(vk_Right) then lightX := lightX + 1;

    if lightX < 0 then lightX := 0;

    ClearScreen(ColorWhite);
    DrawLight(lightIsOn, lightX, lightY);
    RefreshScreen();
  until WindowCloseRequested() or KeyTyped(vk_Escape);
end;

begin
  Main();
end.
```

Figure 6.14: The final code from the lecture example developed with students in the Week 3 lecture of the introductory programming unit. The program shows a light bulb image that can be turned on and off with the mouse and space bar, and moved around the screen using arrow keys.

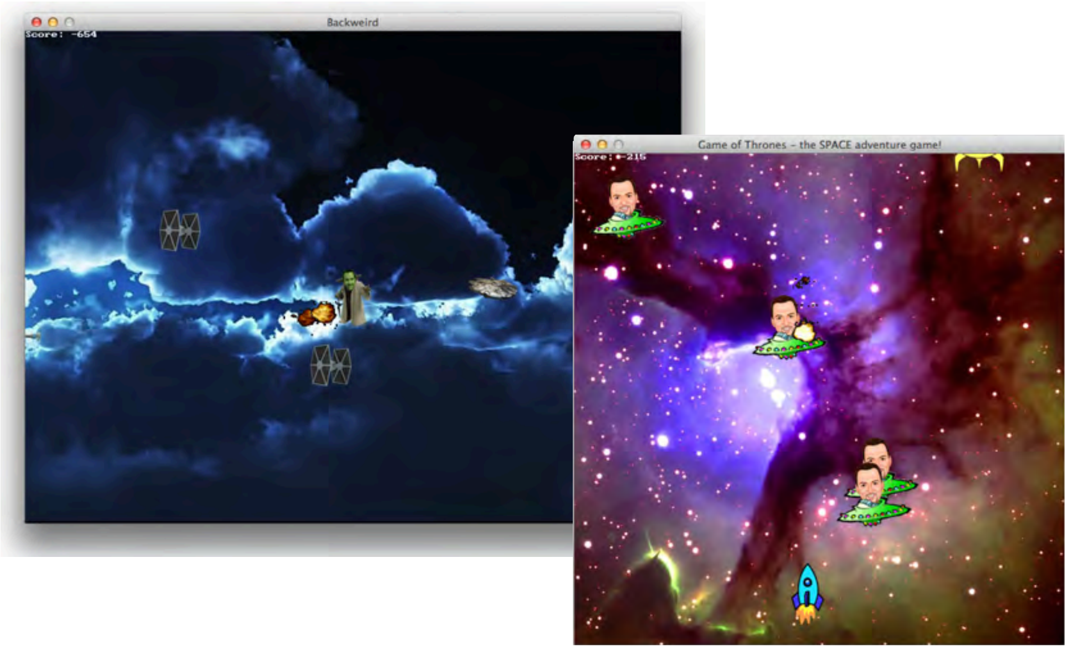


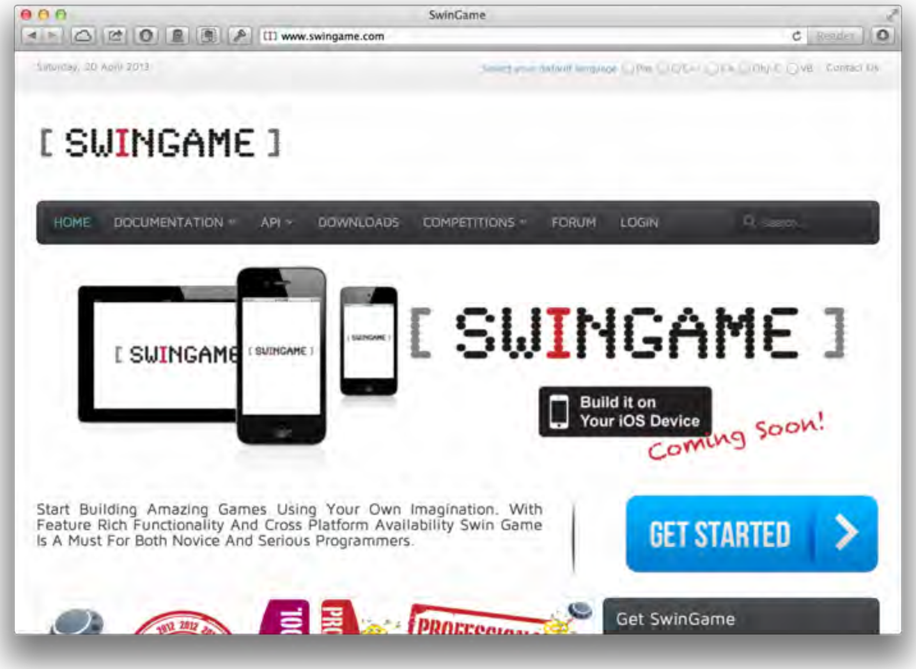
Figure 6.15: Games developed with students across a number of weeks in introductory programming

The assessment criteria developed for the introductory programming unit, described in Section 5.2.3, required students to demonstrate they could apply the concepts from the unit to develop a program of their own design. While there was no requirement for students to use the library, most chose to create a game using SwinGame.

To help support students with their use of SwinGame was documented on a website (SwinGame 2013), as shown in Figure 6.16. The website was created to list the SwinGame functions and procedures, and additional documentation was added for some of the more common tasks, with the aim of supporting students as they started to develop their own programs (P6). The site also provided a means of distributing the SwinGame library to a wider audience.

Developing SwinGame and its Documentation

SwinGame was developed through the collaboration of both staff and students over a number of years. At the end of each year, students were invited to work on enhancing SwinGame's implementation and documentation. This provided an opportunity for staff to work closely with students, and for students to further develop their software development skills outside of the standard teaching periods.



Start Building Amazing Games Using Your Own Imagination. With Feature Rich Functionality And Cross Platform Availability Swin Game Is A Must For Both Novice And Serious Programmers.

Figure 6.16: The SwinGame website provides a means of distributing SwinGame and its documentation

Promoting opportunities to work on SwinGame also provided an opportunity to indicate the depth of knowledge students had developed in the past. This helped communicate the high expectations of staff (P5) and provided encouragement for students to do their best.

Supporting Multiple Languages

With the introductory programming unit using two programming languages, and object oriented programming using four, SwinGame was required to be accessible from a range of programming languages. This included both procedural and object oriented programming languages, which each needed to be supported with appropriate abstractions (P12). To achieve this goal a number of tools were created to simplify the process of creating programming language specific versions of SwinGame.

SwinGame's core logic can be accessed via language specific wrappers as shown in Figure 6.17. Each wrapper mirrors the SwinGame functionality, and acts as an adapter that performs any required transformation of data between the program's runtime environment and the native SwinGame library, which is accessed via the SwinGame Library Interface. This interface consolidates all of the SwinGame functionality and exposes it as a native library, it also acts as an adapter that converts SwinGame types to, and from, appropriate native representations that can be exchanged across the native interface.

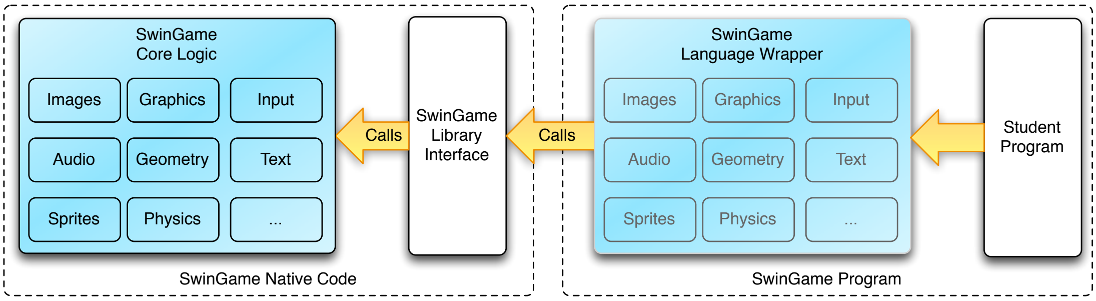


Figure 6.17: SwinGame core logic is implemented in a number of modules that are accessible via language specific wrappers

To ease the creation of the language specific wrappers a translator was created that reads the source code of the SwinGame core logic and outputs the SwinGame Library Interface, a number of matching language specific wrappers, and the programmer documentation for the SwinGame website, as shown in Figure 6.18. This ensures consistency between the wrapper, SwinGame Library Interface and the documentation, while allowing the development of SwinGame to focus on enhancing the core logic.

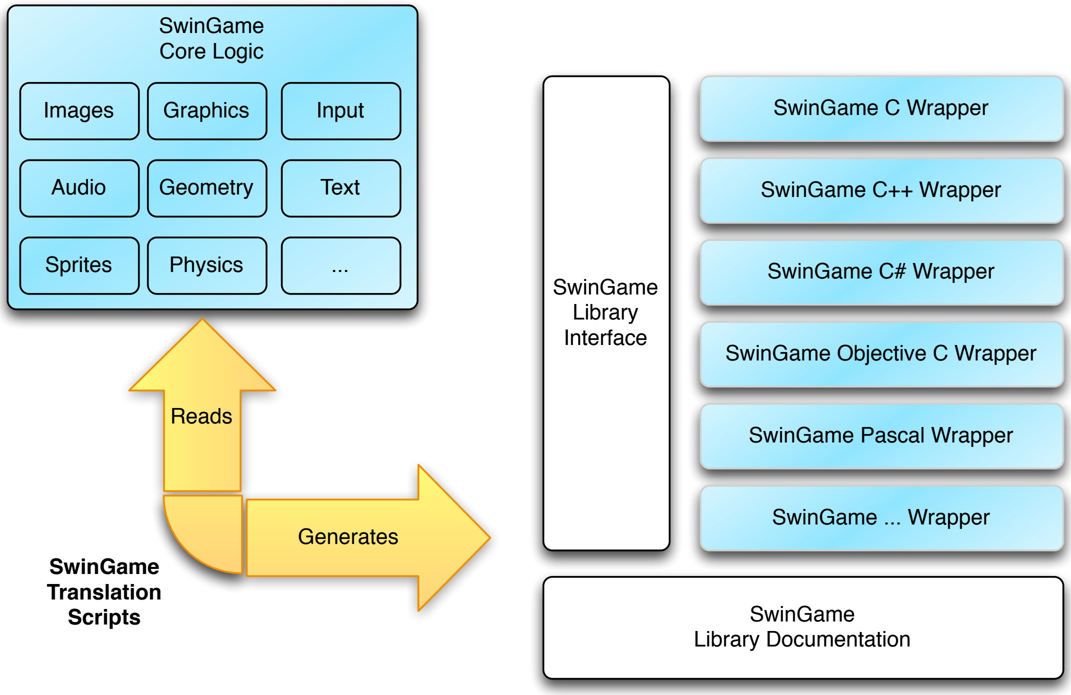


Figure 6.18: SwinGame's language specific wrappers, library interface and programmer documentation are all generated from its source code. The translator reads the source code, and outputs the SwinGame library, and language specific wrappers for a range of programming languages.

SwinGame's language translation scripts requires additional information to enable the construction of the various artefacts. It was decided to store these additional details using attributes in comments in the code of the core logic. These special comments started with `///` and contained documentation as well as other information needed to assist with the translation. These attributes were marked using the `@` symbol followed by a attribute identifier and a number of values. Two example functions from the SwinGame core logic are shown in Figure 6.19, and Table 6.2 provides details of the attributes shown.

Pascal

```
/// Loads the `SoundEffect` from the supplied filename. The
/// sound will be loaded from the Resources/sounds folder
/// unless a full path to the file is passed in...
///
/// @param filename name of the sound effect file to load.
///
/// @lib
///
/// @class SoundEffect
/// @constructor
/// @csn initFromFile:%s
function LoadSoundEffect(filename: String): SoundEffect;

/// This version of PlaySoundEffect allows you to indicate
/// the number of times the sound effect is repeated.
/// ...
///
/// @lib PlaySoundEffectA(effect, loops, 1.0)
/// @uname PlaySoundEffectWithLoop
/// @sn playSoundEffect:%s looped:%s
///
/// @class SoundEffect
/// @overload Play PlayWithLoops
/// @csn playLooped:%s
procedure PlaySoundEffect(effect: SoundEffect; loops: Longint);
```

Figure 6.19: Example of markup language used to annotate SwinGame core logic to enable generation of language specific wrappers

The `@lib` parameter determines if the function or procedure is added to the SwinGame Library Interface. When the function or procedure is not to be added, an alternative call is provided. The generated wrapper code is then adjusted to call the appropriate function. The code in Figure 6.19 indicates that `LoadSoundEffect` should be included directly in the interface, whereas this version of the `PlaySoundEffect` procedure calls `PlaySoundEffectA` and passes in default values for some parameters.

All SwinGame resources are accessible via pointers, enabling the language translation scripts to create object oriented abstractions when this is appropriate for the wrapper's

Table 6.2: The main language translation attributes, their format and purpose.

Attribute	Format	Purpose
`@param`	`@param name docs`	Provides additional documentation for a parameter.
`@lib`	`@lib`	Indicates the function/procedure should appear in the SwinGame Library Interface.
	`@lib call`	Indicates that the wrapper should call another function/procedure from the SwinGame Library Interface.
`@uname`	`@uname name`	Function and procedure names in the SwinGame Library Interface, and some wrappers, need to be unique. This attribute provided a unique name for this function or procedure.
`@sn`	`@sn format`	Provides a format string for the creation of the Objective C signature for this function/procedure. This allows parameters to be mixed with the name of the method.
`@class`	`@class name`	The name of the class to add the method to.
`@method`	`@method`	The name of the method to add to the class.
`@overload`	`@overload name uname`	Overloads the method name, uname is used in langauges that do not support overloading.
`@csn`	`@csn format`	Similar to @sn, providing the format for the Objective C signature for the method added to the class.

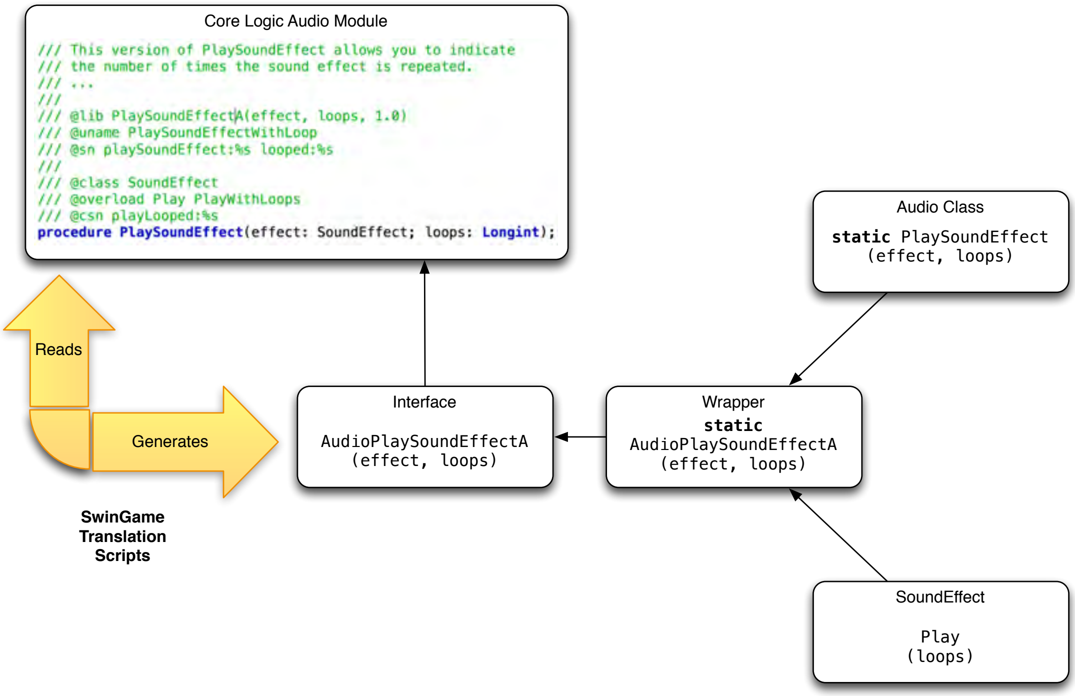
programming language. Figure 6.20 illustrates the main components created by the translator.

Each function and procedure in the core logic is capable of creating two methods in the object oriented language wrappers. The first method is created as a static[2] member of a class that mirrors the module from the core logic. The second method can then be associated with a matching resource class, as an instance member. The specific class is indicated by the `@class` attribute. These classes contain a field to track the resource's pointer, and so the method will have one fewer parameters. When these methods are called, they calls the matching method static method and pass in the value from the pointer field along with any other parameter values passed to the method.

The Objective C syntax required special attention, as method signatures do not separated name from parameters. For example, the procedure PlaySoundEffect shown in Figure 6.19 could be called using `[Audio playSoundEffect:effect looped:3]`. The signature to match this required the parameters to be embedded within the method's name. This was achieved by adding `@sn` and `@csn` attributes that allow the developer to specify a format string into which the parameter signatures are injected.

By supporting both procedural and object oriented programming languages, SwinGame was able to be used in both the introductory programming and object oriented programming units. In introductory programming students worked with SwinGame from Week 1, and explored its use from both the Pascal and C programming lan-

[2] Static in this context is meant to indicate that the method is associated with the class, rather than instances of the class.



```
/// This version of PlaySoundEffect allows you to indicate
/// the number of times the sound effect is repeated.
/// ...
///
/// @lib PlaySoundEffectA(effect, loops, 1.0)
/// @uname PlaySoundEffectWithLoop
/// @sn playSoundEffect:%s looped:%s
///
/// @class SoundEffect
/// @overload Play PlayWithLoops
/// @csn playLooped:%s
procedure PlaySoundEffect(effect: SoundEffect; loops: Longint);
```

Core Logic Audio Module
Reads
Generates
SwinGame Translation Scripts
Interface
AudioPlaySoundEffectA (effect, loops)
Wrapper
static AudioPlaySoundEffectA (effect, loops)
Audio Class
static PlaySoundEffect (effect, loops)
SoundEffect
Play (loops)

Figure 6.20: Attributes in the core logic code define the generation of a module level wrapper, and the creation of classes for object oriented access to SwinGame resources.

guages. In the object oriented programming unit, SwinGame provided a familiar framework for students to work with at the start of the teaching period. As the teaching period progressed, students were transitioned away from using SwinGame to encourage them to explore other commercially available libraries by the end of the unit.

Supporting What We Teach

SwinGame performed a central role in supporting *what* was taught in the two example programming units described in Chapter 5. It also provided backing for other resources developed to support the teaching of these units, as illustrated in Figure 6.21. SwinGame helped enable interactive lectures, and provided the tools necessary to create engaging examples in the Programming Arcana text, outlined in Section 6.3, and the video podcasts, described in Section 6.4.

It also provided a consistent library for students to use as they moved between programming languages, and programming units (P6 and P11). While the SwinGame interface differed slightly, adapting to language conventions and abstractions, the interface was still a familiar quantity as students moved between languages and paradigms. This meant that students could more rapidly create programs in the new language, as the SwinGame library remained relatively consistent between the different environments.

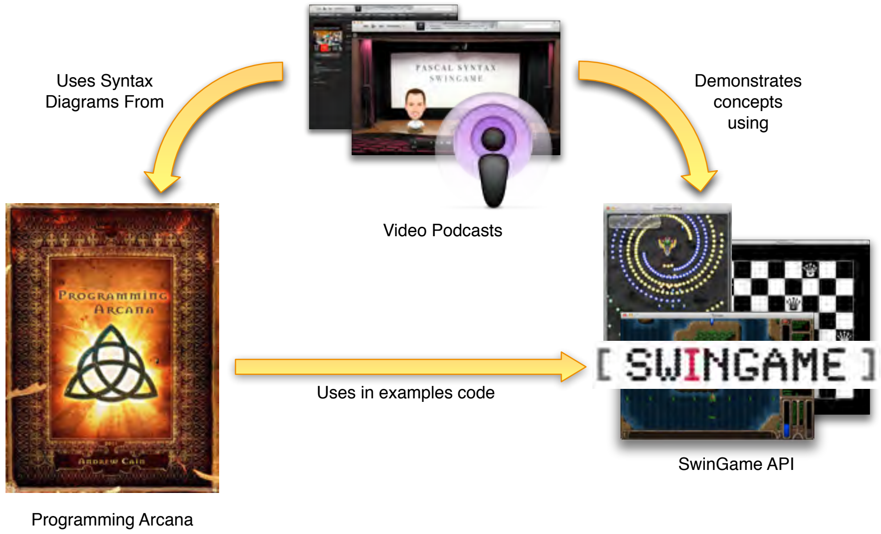


Figure 6.21: SwinGame supported unit delivery, the Programming Arcana, and the video podcasts.

Evaluation and Support of Principles

Staff and student reflections indicate that SwinGame has been a successful part of the teaching strategy for the introductory and object oriented programming units. In the introductory programming unit SwinGame enabled staff to use interactive lecture demonstrations, and provided students with a library they could use to create small games for their custom projects. It supported the shift between programming languages, and between programming paradigms as student transition from the introductory programming unit to object oriented programming.

SwinGame has met most of the requirements listed in Section 6.2.1, the following list outlines current strengths and weaknesses.

- Strengths:
  - Provides required procedural, and object oriented, access to functionality to enable the creation of 2D games.
  - Works cross platform, and across multiple languages.
  - Device support includes desktop computers, and iOS devices (with touch and accelerometer support).
  - Does provide support for drawing (shapes and images), audio, sprites, resource management, camera management, input, timers, networking, and

physics.

- Does **not** take control away from the programmer – less productive for better understanding.

- Weaknesses (wish list):
  - Documentation is very weak, developed by students early in their degree programmes.
  - SwinGame is not highly optimised, being developed primarily by students.
  - Needs support for a wider range of mobile devices, and support for additional input mechanisms.
  - Does not support:
    - Image rotation and scaling, an often requested feature.
    - Particle effects.
    - Special effects, such as blur, fog, etcetera.
  - Lacks developer support beyond immediate teaching staff.

SwinGame directly supported the programming units outlined in Chapter 5. Many of the lectures and weekly tasks in the introductory programming unit made use of the game library. Similarly, many of the early tasks in the object oriented programming unit used SwinGame as a means of learning the new language, and paradigm, without also needing to learn a new library.

In terms of the principles stated in Chapter 3, and the model outlined in Chapter 4, SwinGame provided the following support:

- SwinGame helped realise the principles through:
  - Interactive lecture demonstrations aimed to help guide students in the construction of their knowledge. (P1)
  - Requiring explicit programmer control helped align SwinGame programming tasks with the programming concepts, and in turn the unit's intended learning outcomes. (P2 and P11)
  - The consistent framework helped students overcome language differences, to better focus on programming concepts. (P4)
  - Student involvement in the development of SwinGame provided a means of communicating high staff expectations. (P5)
  - SwinGame documentation, and simple interface, helped support student exploration of the library. (P6)
  - SwinGame helped staff engage with students, enabling them to guide lecture content. (P7)

  - SwinGame provides procedural and object oriented programming abstractions, enabling it to be use in both procedural and object oriented programming languages. (P10 and P12)

- SwinGame helped implement the model in the example units by:
  - Providing a valuable resource used in the teaching and learning activities of both programming units.
  - Enabling students to develop a range of games for their custom projects.
  - Providing a consistent library to help simplify the transitioning between languages.
  - Supporting visual programming, making it easier for students to see when their programs were not working successfully.

6.2.4 Summary

SwinGame provided a valuable resource in supporting students learning in the introductory programming units. The consistent library supported students transition between languages, interactive lecture demonstrations, and provided a wealth of features students could exploit in their custom programs. The strengths of SwinGame help maintain the focus on programming concepts. SwinGame remains an actively used library, and development continues to help address some of its current weaknesses.

6.3 Programming Text to Support Concept-Based Approach

Concepts are central to *what* we aim to teach. Principle 11 indicates that we should focus on concepts over language syntax. In addressing this principle the programming units from Chapter 5 had little, if any, coverage of syntax in lectures, leaving these details instead to teaching and learning resources for students to use. These resources provide the syntax details students need to turn these concepts into working code.

One of the central ideas of "Beyond Bullet Points" is to fully document a presentation using the notes attached to a presentation's slides (Atkinson 2007). In effect, details are moved from a slide itself to the slides' notes area, which can then be printed as an informative handout. While documenting slides in this manner can provide students with the required details, it does mix the purpose of the presentation's slides as a means of guiding student thoughts and providing detailed information.

From our experience using the "Beyond Bullet Points" approach, this has a number of drawbacks in relation to the principles from Chapter 3. The dual purpose of the presentation works against maintaining a clear focus (P4) as details may be better presented in a different order to the presentation slides, and visa versa. Similarly, the need for detailed notes for each slide works against Principle 8, being agile and willing to change. The creation of the detailed notes results in significant effort being expended on the creation of each week's presentation, and thereby adds resistance to change if the presentation is found to be ineffective.

Instead of documenting these notes in the presentations themselves, they were written up in a separate resource which became the "*Programming Arcana*" (Cain 2013*b*). The title, cover image (see Figure 6.22), and layout were designed around a magic theme in the aim of engaging students, they are becoming wizards of the modern era capable to making the computer do amazing things.

Documenting language details in a separate text from the presentations also helped to address another issue raised as a result of choosing Pascal as one of the programming languages. Pascal is not currently a popular language with institutions or text book writers, and while the Free Pascal Language Reference Guide (Van Canneyt 2013) provides details of the language it is not designed for beginners. By providing our own text it was possible to maintain the concept-based ideas throughout all teaching and learning resources and activities, while also providing details on the Pascal and C languages, along with suitable programming examples.

This section outlines the requirements that the Programming Arcana aimed to met in

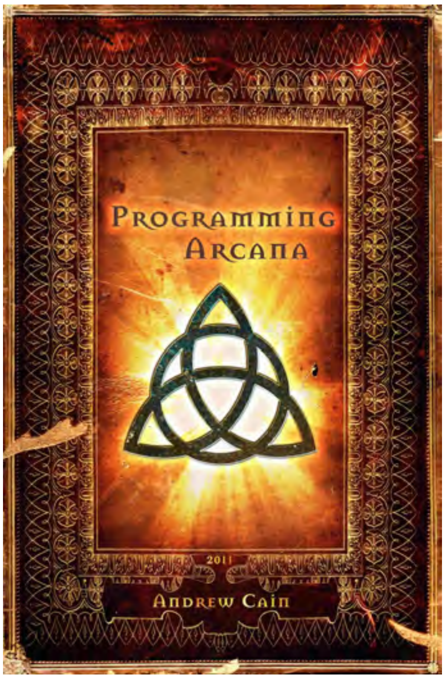


Figure 6.22: Front cover of the Programming Arcana, which used a magic theme to engage students as they worked towards becoming wizards with the computer.

Section 6.3.1. Section 6.3.2 describes the structure of the Programming Arcana, and the way in which it met its requirements. A short evaluation of the use of the Programming Arcana follows in Section 6.3.3. This section then concludes with a brief summary in Section 6.3.4.

6.3.1 Requirements

The requirements for the Programming Arcana were to:

- Focus on programming concepts, presenting them in a language neutral manner.
- Order concepts to align with the introductory programming unit.
- Demonstrate application of the concepts to create a sample program, illustrating the thought process experts use.
- Provide syntax details to map concepts to code for the Pascal and C programming languages.
- Explain how the concepts operate on a notional machine.
- Provide sample code demonstrating applications of concepts, large and small.
- Be concise, focusing only aspects necessary to understand core ideas – rather than presenting details on a range of possible options.
- Communicate ideas using a range of media.

6.3.2 Arcana Solution

Chapter Sequence

Chapters in the Programming Arcana align with the concept topics from the introductory programming unit. This means that the text embodies the concept-based approach (P11) with each chapter providing a coherent set of concepts that build upon concepts presented earlier in the text. The following list outlines the main focus for each chapter. Additional details are provided in Appendix A2 which lists the programming concepts that are presented in each of these chapters.

1. **Building Programs**: Introduces students to the tools they require, and shows them a basic, "Hello World", program they can compile to check that their tools are working.
2. **Program Creation**: describes how code can be written to create a *Program*.
3. **Procedure Declaration**: Introduces the idea that you can create your own procedures to encapsulate the steps of a task.
4. **Storing and Using Data**: Makes programs more dynamic using variables and constants to store data, and functions to calculate values.
5. **Control Flow**: Introduces structured programming principles, along with selection and repetition.
6. **Managing Multiple Values**: Presents the use of arrays to make it easier to work with a large amount of data.
7. **Custom Data Types**: Describes how developers can create types to help them organise the data in their programs, much as functions and procedures helped to organise functionality.
8. **Dynamic Memory Allocation**: Extends programs beyond the confines of the stack, allowing the allocation of data on the heap.
9. **Input and Output**: Describes how to save and load data from file.

In proposing Principle 11, with its focus on programming concepts, Chapter 3 outlined the requirement "Introduce programming concepts incrementally." The Programming Arcana provides an example of how the details of a programming language can be presented in such a way as to ensure most topics are presented with each topic building upon the previously presented topics.

There were two cases where concepts could not be suitably explained within the overall context presented in a chapter. There were:

1. In Chapter 1 the code for a working program was given to enable students to compile something before they understood what it represented. However, the main focus of the chapter was the tools being presented and not the specific details of the program's code, and so this does not directly contradict the underlying principle.
2. Chapters 2 and 3 makes use of values passed to procedures before topics related to *how* data can be stored in a program. The idea that data can be passed to a procedure is covered, but not how that data was received, as the concept of a variable was not introduced until Chapter 4.

Other than these two cases, all other chapters were able to explain all concepts in terms of the presented, or previously presented, concepts.

When comparing the suitability of the C and Pascal languages for supporting the concept-based approach it was noted that, in general, mapping the concepts to syntax was simpler for the Pascal programming language, with the C[3] language providing a number of challenges. C's standard input and output functions, `printf` and `scanf`, provided a range of challenges associated with the use of format strings and pointers. The format string provides an additional syntax to learn, and results in a range of runtime errors where the types indicated in the format string do not match the types of the associated variables or expressions. The need to pass explicit pointers to scanf also required a brief description of pointers in early material. Other challenges related to the need to understand arrays before working with strings. Early topics avoided strings, or used only string literals. In this way one example can be mapped to both C and Pascal languages.

In relation to the use of the text in supporting the introductory programming unit, many of the issues with C were avoided due to the use of Pascal in the first part of the unit. This enabled teaching and learning activities to take advantage of Pascal's more convenient support for strings and terminal input and output. For example, consider a program that asked the user to enter their name and then echoes back a welcome message. In C this requires an understanding of variables, format string syntax, arrays, pointers, and how arrays are automatically passed by references. In Pascal the same program only requires an understanding of variables and pass-by-reference, as the language abstracts away many underlying details of how strings work.

[3] The C code was compiled with a C++ compiler to add support for function and procedure overloading, and pass-by-reference.

Chapter Layout

Each chapter of the Programming Arcana has a similar sequence to its sections, with the intention of reducing cognitive overhead and promoting a consistent approach to studying each of the topic. In keeping with Principle 11, the concepts were presented as the focus of each chapter.

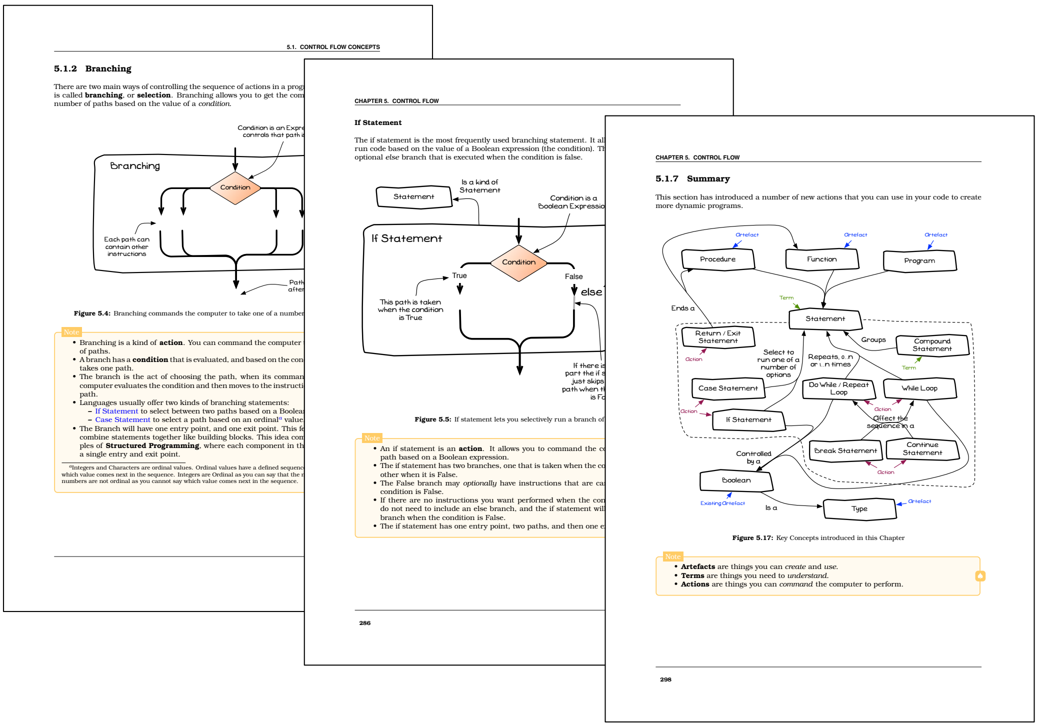
1. **Concepts**: Each chapter starts with a list of related concepts, each of which is described at a relevant level of detail for that chapter.
2. **Applying the Concepts**: An example of how to apply the chapter's concepts is then discussed, using pseudocode and flowcharts to illustrate how the concepts can be applied.
3. **Syntax in C and Pascal**: Details related to the syntax needed to realise these concepts in code are first presented for the C programming language, and then for the Pascal language.
4. **Understanding the Concepts**: Traces the execution of the pseudocode on a conceptual machine, with the aim of showing students how the concepts are realised at run time.
5. **Examples**: A number of examples are given to further demonstrate the application of the chapter's concepts, each is presented in pseudocode and then in C and Pascal code.
6. **Exercises**: Provides a sequence of exercises students can use to develop their understanding of the topic.

Details of the sections related to presenting the concepts follow. This outlines how the Programming Arcana implemented the concept-based approach, and reinforce the focus on concepts over syntax throughout the material presented.

Concepts Each chapter starts with a section that provides details of the concepts being presented. This starts with a brief overview describing how all of the concepts are related, with following subsections presenting details for each concept. Concepts are presented using a textual description, visual concept map, and a series of notes with important details related to the topic. At the end of the concept section an overall concept map is included to visually summarise the relationships between the concepts covered.

Figure 6.23 shows an example of the concept of branching from Chapter 5 of the Programming Arcana. The diagrams were deliberately drawn using irregular, rough looking, shapes to indicate these were a conceptualisation, rather than an exact representation of the associated concepts.

One of the design goals was to fit each concept on a single page. This goal aimed to help support a student's active construction of knowledge, Principle 1. Aiming to keep each topic to a single page ensured a focus (Principle 4) on the most important details, and where topics expanded over multiple pages the details were examined to ensure they did not included any unnecessary details.



5.1. CONTROL FLOW CONCEPTS

5.1.2 Branching

There are two main ways of controlling the sequence of actions in a prog
is called **branching**, or **selection**. Branching allows you to get the com
number of paths based on the value of a *condition*.

Figure 5.4: Branching commands the computer to take one of a number

Note

- Branching is a kind of **action**. You can command the computer
of paths.
- A branch has a **condition** that is evaluated, and based on the con
takes one path.
- The branch is the act of choosing the path, when its comman
computer evaluates the condition and then moves to the instructi
path.
- Languages usually offer two kinds of branching statements:
  - If Statement to select between two paths based on a Boolea
  - Case Statement to select a path based on an ordinal[a] value
- The Branch will have one entry point, and one exit point. This fe
combine statements together like building blocks. This idea com
ples of **Structured Programming**, where each component in th
a single entry and exit point.

[a] Integers and Characters are ordinal values. Ordinal values have a defined sequenc
which value comes next in the sequence. Integers are Ordinal as you can say that the n
numbers are not ordinal as you cannot say which value comes next in the sequence.

CHAPTER 5. CONTROL FLOW

If Statement

The if statement is the most frequently used branching statement. It all
run code based on the value of a Boolean expression (the condition). Th
optional *else* branch that is executed when the condition is false.

Figure 5.5: If statement lets you selectively run a branch of

Note

- An if statement is an **action**. It allows you to command the c
path based on a Boolean expression.
- The if statement has two branches, one that is taken when the co
other when it is False.
- The False branch may *optionally* have instructions that are ca
condition is False.
- If there are no instructions you want performed when the con
do not need to include an else branch, and the if statement wil
branch when the condition is False.
- The if statement has one entry point, two paths, and then one e

286

CHAPTER 5. CONTROL FLOW

5.1.7 Summary

This section has introduced a number of new actions that you can use in your code to create more dynamic programs.

Figure 5.17: Key Concepts introduced in this Chapter

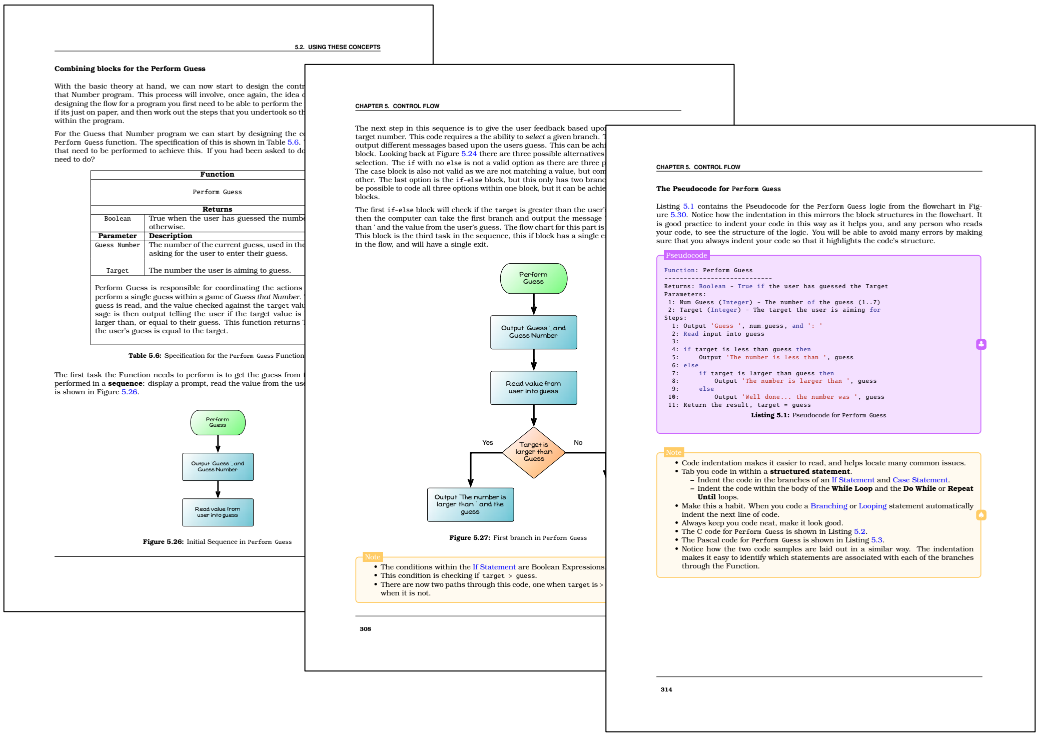
Note

- **Artefacts** are things you can *create* and *use*.
- **Terms** are things you need to *understand*.
- **Actions** are things you can *command* the computer to perform.

298

Figure 6.23: Example concept pages from Chapter 5 of the Programming Arcana, showing the use of visual concept maps to help explain concepts.

Applying the Concepts After the concepts are presented, the next section outlines how these concepts work together to create an example program. This helps demonstrate the thought process of an expert programmer in applying these concepts to a program design, with the aim of helping students construct similar thought patterns (P1). This section starts with a specification of a program to be created. This is then followed by a discussion of how a program can be designed using the concepts covered to that point in the text. The description of the design includes pseudocode, flow charts, sequence diagrams and structure charts, and the section concludes with a complete design for the specified program. Figure 6.24 shows an example of designing a "Guess that Number" game that demonstrates the application of control flow concepts.



5.2. USING THESE CONCEPTS

Combining blocks for the Perform Guess

With the basic theory at hand, we can now start to design the contr that Number program. This process will involve, once again, the idea c designing the flow for a program you first need to be able to perform the if its just on paper, and then work out the steps that you undertook so th within the program.

For the Guess that Number program we can start by designing the co `Perform Guess` function. The specification of this is shown in Table 5.6. that need to be performed to achieve this. If you had been asked to do need to do?

Function	
`Perform Guess`	
Returns	
`Boolean`	True when the user has guessed the numbe otherwise.
Parameter	**Description**
`Guess Number`	The number of the current guess, used in the asking for the user to enter their guess.
`Target`	The number the user is aiming to guess.
Perform Guess is responsible for coordinating the actions perform a single guess within a game of *Guess that Number*. guess is read, and the value checked against the `target` valu sage is then output telling the user if the target value is larger than, or equal to their guess. This function returns the user's guess is equal to the target.	

Table 5.6: Specification for the `Perform Guess` Function

The first task the Function needs to perform is to get the guess from performed in a **sequence**: display a prompt, read the value from the use is shown in Figure 5.26.

Perform Guess
Output 'Guess ', and Guess Number
Read value from user into guess

Figure 5.26: Initial Sequence in `Perform Guess`

CHAPTER 5. CONTROL FLOW

The next step in this sequence is to give the user feedback based upo target number. This code requires a the ability to *select* a given branch. T output different messages based upon the users guess. This can be achi block. Looking back at Figure 5.24 there are three possible alternatives selection. The `if` with no `else` is not a valid option as there are three p The `case` block is also not valid as we are not matching a value, but com other. The last option is the `if-else` block, but this only has two branc be possible to code all three options within one block, but it can be achie blocks.

The first `if-else` block will check if the `target` is greater than the user' then the computer can take the first branch and output the message than ' and the value from the user's guess. The flow chart for this part is This block is the third task in the sequence, this if block has a single e in the flow, and will have a single exit.

Perform Guess
Output 'Guess ', and Guess Number
Read value from user into guess
Target is larger than Guess
Yes
No
Output 'The number is larger than ' and the guess

Figure 5.27: First branch in `Perform Guess`

Note
- The conditions within the If Statement are Boolean Expressions
- This condition is checking if `target > guess`.
- There are now two paths through this code, one when `target` is > when it is not.

308

CHAPTER 5. CONTROL FLOW

The Pseudocode for `Perform Guess`

Listing 5.1 contains the Pseudocode for the `Perform Guess` logic from the flowchart in Figure 5.30. Notice how the indentation in this mirrors the block structures in the flowchart. It is good practice to indent your code in this way as it helps you, and any person who reads your code, to see the structure of the logic. You will be able to avoid many errors by making sure that you always indent your code so that it highlights the code's structure.

Pseudocode

```
Function: Perform Guess
----------------------------
Returns: Boolean - True if the user has guessed the Target
Parameters:
 1: Num Guess (Integer) - The number of the guess (1..7)
 2: Target (Integer) - The target the user is aiming for
Steps:
  1: Output 'Guess ', num_guess, and ': '
  2: Read input into guess
  3:
  4: if target is less than guess then
  5:     Output 'The number is less than ', guess
  6: else
  7:     if target is larger than guess then
  8:         Output 'The number is larger than ', guess
  9:     else
 10:         Output 'Well done... the number was ', guess
 11: Return the result, target = guess
```

Listing 5.1: Pseudocode for `Perform Guess`

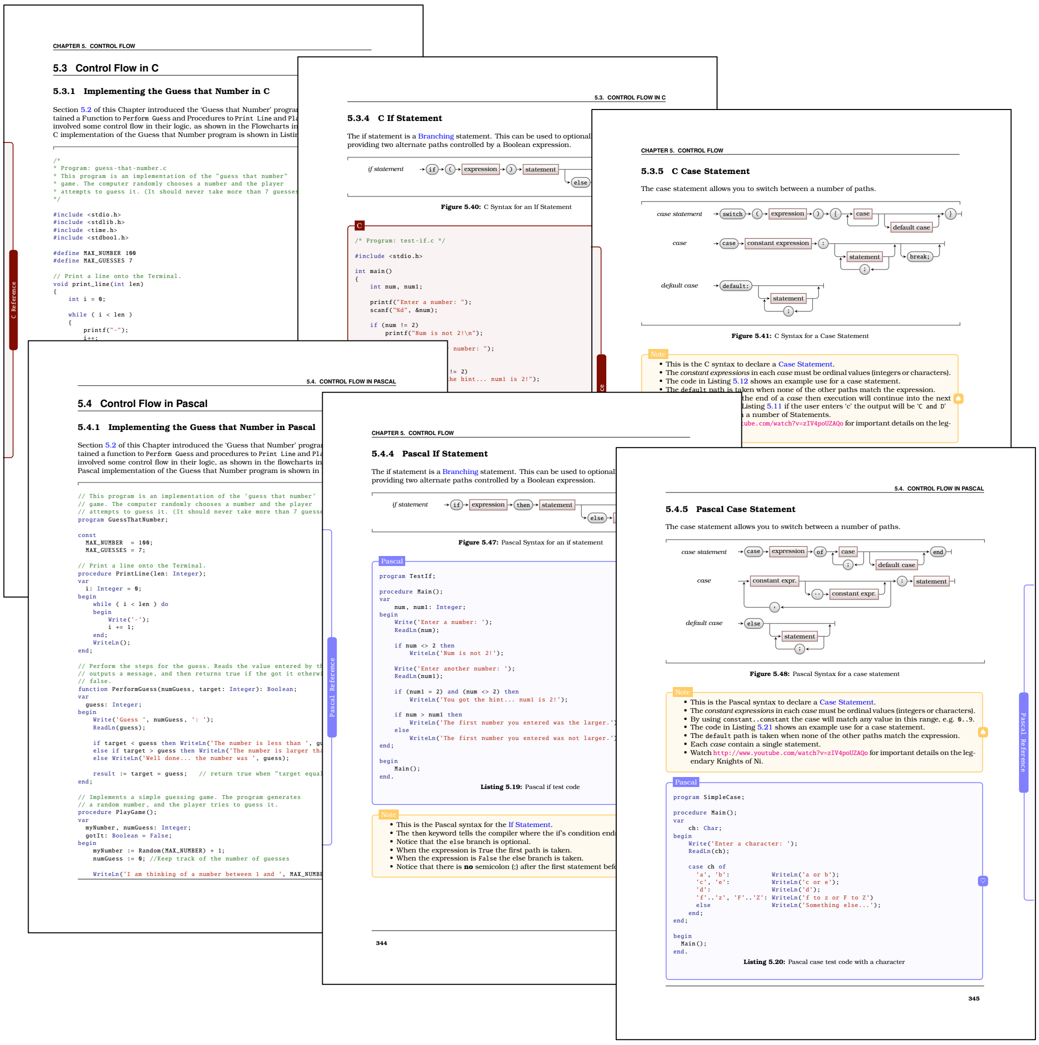
Note
- Code indentation makes it easier to read, and helps locate many common issues.
- Tab you code in within a **structured statement**.
  - Indent the code in the branches of an If Statement and Case Statement.
  - Indent the code within the body of the **While Loop** and the **Do While** or **Repeat Until** loops.
- Make this a habit. When you code a Branching or Looping statement automatically indent the next line of code.
- Always keep you code neat, make it look good.
- The C code for `Perform Guess` is shown in Listing 5.2.
- The Pascal code for `Perform Guess` is shown in Listing 5.3.
- Notice how the two code samples are laid out in a similar way. The indentation makes it easy to identify which statements are associated with each of the branches through the Function.

314

Figure 6.24: Example pages related to applying the concepts from the Programming Arcana

In each chapter explanatory text accompanies the design, and highlights how the concepts covered contribute to the end result. This discussion also presents a way of approaching problems using the concepts to introduce students to the ideas they can use in approaching the design of their own programs.

Syntax Having covered concepts, and how they can be used to design a program, the next two sections deal with realising these concepts using the C and Pascal programming languages. The syntax sections start with an implementation of the program designed in the section on applying the concepts, in this way students see design through to its implementation with the aim of helping them develop appropriate conceptual connections (P1). This is followed by details of each part of the language syntax, providing this section with a clear focus (P4). An example of pages from this section is shown in Figure 6.25.



CHAPTER 5. CONTROL FLOW

5.3 Control Flow in C

5.3.1 Implementing the Guess that Number in C

Section 5.2 of this Chapter introduced the 'Guess that Number' progra
tained a Function to Perform Guess and Procedures to Print Line and Pla
involved some control flow in their logic, as shown in the Flowcharts in
C implementation of the Guess that Number program is shown in Listi

```
/*
* Program: guess-that-number.c
* This program is an implementation of the "guess that number"
* game. The computer randomly chooses a number and the player
* attempts to guess it. (It should never take more than 7 guesses
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

#define MAX_NUMBER 100
#define MAX_GUESSES 7

// Print a line onto the Terminal.
void print_line(int len)
{
    int i = 0;

    while ( i < len )
    {
        printf("-");
        i++;
```

C Reference

5.3. CONTROL FLOW IN C

5.3.4 C If Statement

The if statement is a Branching statement. This can be used to optional
providing two alternate paths controlled by a Boolean expression.

Figure 5.40: C Syntax for an If Statement

C

```
/* Program: test-if.c */

#include <stdio.h>

int main()
{
    int num, num1;

    printf("Enter a number: ");
    scanf("%d", &num);

    if (num != 2)
        printf("Num is not 2!\n");

 number: ");

 != 2)
he hint... num1 is 2!");
```

CHAPTER 5. CONTROL FLOW

5.3.5 C Case Statement

The case statement allows you to switch between a number of paths.

Figure 5.41: C Syntax for a Case Statement

Note

- This is the C syntax to declare a Case Statement.
- The *constant expressions* in each *case* must be ordinal values (integers or characters).
- The code in Listing 5.12 shows an example use for a case statement.
- The default path is taken when none of the other paths match the expression.
- the end of a *case* then execution will continue into the next
- Listing 5.11 if the user enters 'c' the output will be 'C and D'
- a number of Statements.
- ube.com/watch?v=zIV4poUZAQo for important details on the leg-

5.4. CONTROL FLOW IN PASCAL

5.4 Control Flow in Pascal

5.4.1 Implementing the Guess that Number in Pascal

Section 5.2 of this Chapter introduced the 'Guess that Number' progra
tained a function to Perform Guess and procedures to Print Line and Pla
involved some control flow in their logic, as shown in the flowcharts in
Pascal implementation of the Guess that Number program is shown in

```
// This program is an implementation of the 'guess that number'
// game. The computer randomly chooses a number and the player
// attempts to guess it. (It should never take more than 7 guesse
program GuessThatNumber;

const
  MAX_NUMBER  = 100;
  MAX_GUESSES = 7;

// Print a line onto the Terminal.
procedure PrintLine(len: Integer);
var
  i: Integer = 0;
begin
    while ( i < len ) do
    begin
        Write('-');
        i += 1;
    end;
    WriteLn();
end;

// Perform the steps for the guess. Reads the value entered by th
// outputs a message, and then returns true if the got it otherwi
// false.
function PerformGuess(numGuess, target: Integer): Boolean;
var
  guess: Integer;
begin
    Write('Guess ', numGuess, ': ');
    ReadLn(guess);

    if target < guess then WriteLn('The number is less than ', gu
    else if target > guess then WriteLn('The number is larger tha
    else WriteLn('Well done... the number was ', guess);

    result := target = guess;   // return true when "target equal
end;

// Implements a simple guessing game. The program generates
// a random number, and the player tries to guess it.
procedure PlayGame();
var
  myNumber, numGuess: Integer;
  gotIt: Boolean = False;
begin
    myNumber := Random(MAX_NUMBER) + 1;
    numGuess := 0; //Keep track of the number of guesses

    WriteLn('I am thinking of a number between 1 and ', MAX_NUMBE
```

CHAPTER 5. CONTROL FLOW

5.4.4 Pascal If Statement

The if statement is a Branching statement. This can be used to optional
providing two alternate paths controlled by a Boolean expression.

Figure 5.47: Pascal Syntax for an if statement

Pascal

```
program TestIf;

procedure Main();
var
    num, num1: Integer;
begin
    Write('Enter a number: ');
    ReadLn(num);

    if num <> 2 then
        WriteLn('Num is not 2!');

    Write('Enter another number: ');
    ReadLn(num1);

    if (num1 = 2) and (num <> 2) then
        WriteLn('You got the hint... num1 is 2!');

    if num > num1 then
        WriteLn('The first number you entered was the larger.')
    else
        WriteLn('The first number you entered was not larger.')
end;

begin
    Main();
end.
```

Listing 5.19: Pascal if test code

Note

- This is the Pascal syntax for the If Statement.
- The then keyword tells the compiler where the if's condition end
- Notice that the else branch is optional.
- When the expression is True the first path is taken.
- When the expression is False the else branch is taken.
- Notice that there is **no** semicolon (;) after the first statement bef

344

Pascal Reference

5.4. CONTROL FLOW IN PASCAL

5.4.5 Pascal Case Statement

The case statement allows you to switch between a number of paths.

Figure 5.48: Pascal Syntax for a case statement

Note

- This is the Pascal syntax to declare a Case Statement.
- The *constant expressions* in each *case* must be ordinal values (integers or characters).
- By using constant..constant the case will match any value in this range, e.g. 0..9.
- The code in Listing 5.21 shows an example use for a case statement.
- The default path is taken when none of the other paths match the expression.
- Each *case* contain a single statement.
- Watch http://www.youtube.com/watch?v=zIV4poUZAQo for important details on the legendary Knights of Ni.

Pascal

```
program SimpleCase;

procedure Main();
var
    ch: Char;
begin
    Write('Enter a character: ');
    ReadLn(ch);

    case ch of
      'a', 'b':           WriteLn('a or b');
      'c', 'e':           WriteLn('c or e');
      'd':                WriteLn('d');
      'f'..'z', 'F'..'Z': WriteLn('f to z or F to Z')
      else                WriteLn('Something else...');
    end;
end;

begin
  Main();
end.
```

Listing 5.20: Pascal case test code with a character

345

Pascal Reference

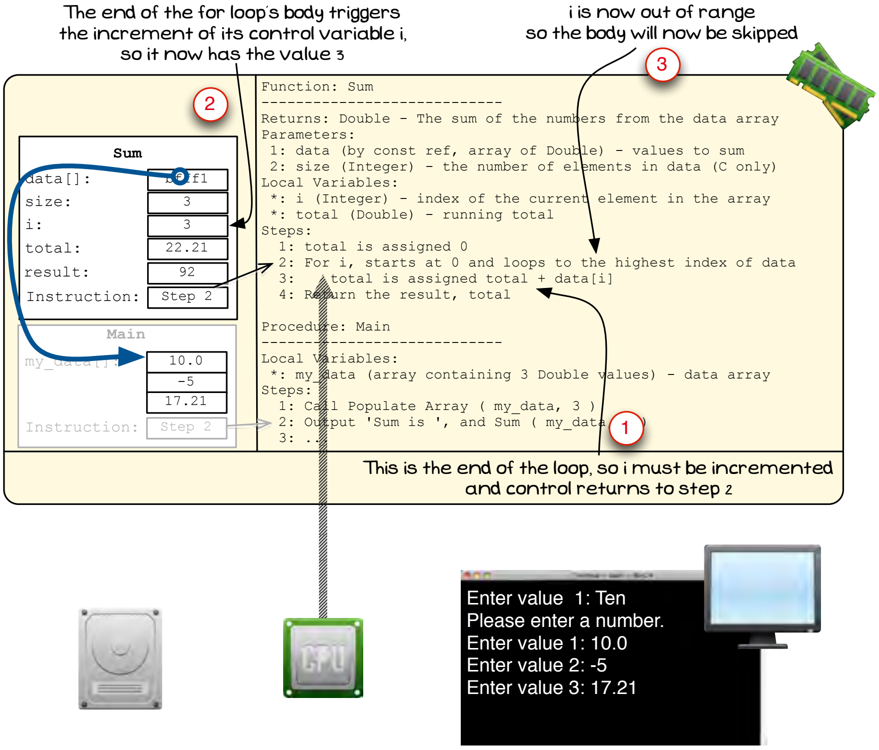
Figure 6.25: Example pages from the Programming Arcana showing C and Pascal syntax and examples

Maintaining a clear focus on the most important aspects continues in the presentation of the syntax (P4). The design aimed for each aspect of the syntax to be presented in a single page. The grammar, and examples, presented focus on best representing the concepts, which in many cases means only presenting a small subset of what is possible with the programming language.

Understanding the Concepts In order to think and act as experts, the goal of constructivist education, it is important for students to understand how the instructions affect the machine they are programming (P1). This imperative was highlighted by Ben-Ari (1998, 2001) in their analysis of constructivism in computer science education, in which they indicated the critical importance of ensuring students develop an effective model of computation. For the introductory programming unit, the intended learning outcomes indicated that students need to be able to *explain* their programs, this aimed to encourage students to develop, and communicate, their model of computing, and required resources they could draw upon to highlight these lower level operations (P2).

The notional machine represents an ideal computer in which the programming constructs being taught are realised (DuBoulay 1986). To help students realise the goal of *understanding* how to program this machine, the next section of each chapter in the Programming Arcana provided a series of illustrations. These illustrations aim to communicate the state and behaviour of the notional machine as it executes the example program developed in earlier sections. Figure 6.26 shows an example of the notional machine from the Programming Arcana, the machine contains a persistent storage device, central processing unit (CPU), terminal for input and output, and memory that is divided into sections for global values, stack, instructions, and heap – in later chapters.

Figure 6.27 shows some examples from the chapter on control flow. The illustration of the notional machine focuses on memory, and the instruction the computer is executing. Instructions from the pseudocode are executed, one by one, with each instruction being explained on a single page. The state of the machine is discussed after each instruction, with each page including a short description of what is occurring, a visualisation of the notional machine, related notes on the steps taken by the machine, and any language specific notes. Annotations were added to the visualisation of the notional machine to help link the comments to changes in the machine's state.


The end of the for loop's body triggers the increment of its control variable i, so it now has the value 3
i is now out of range so the body will now be skipped
2
3
1
Sum
data[]:
size:
i:
total:
result:
Instruction:
Main
my_data[]:
Instruction:
This is the end of the loop, so i must be incremented and control returns to step 2

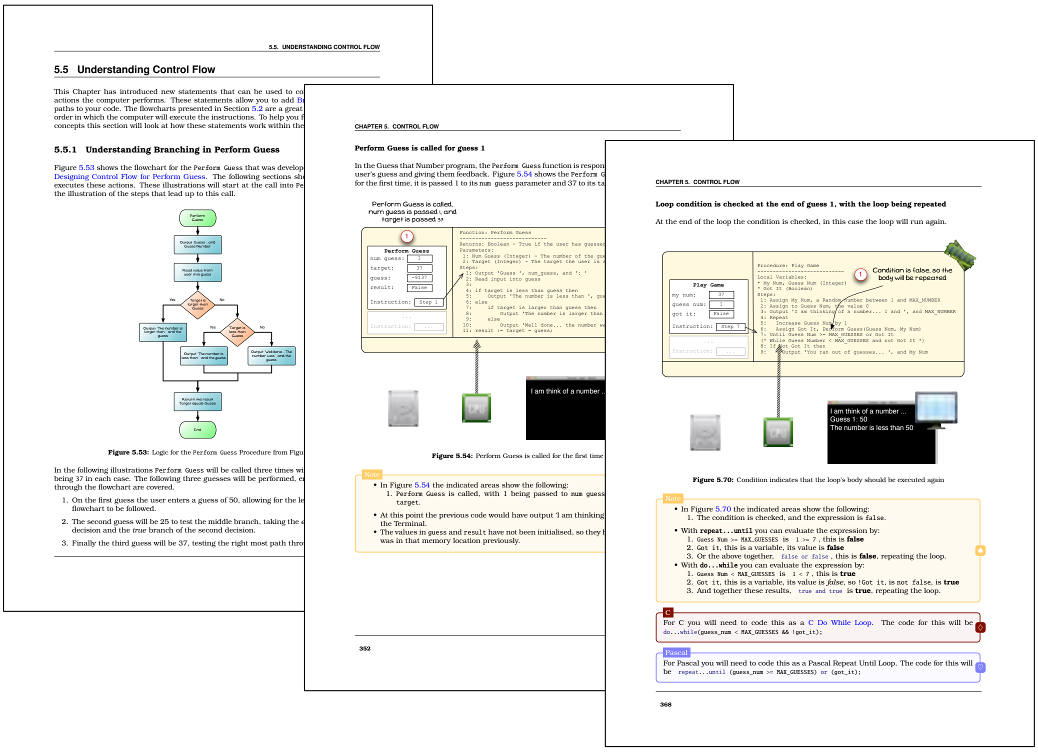
```
Function: Sum
----------------------------
Returns: Double - The sum of the numbers from the data array
Parameters:
 1: data (by const ref, array of Double) - values to sum
 2: size (Integer) - the number of elements in data (C only)
Local Variables:
 *: i (Integer) - index of the current element in the array
 *: total (Double) - running total
Steps:
  1: total is assigned 0
  2: For i, starts at 0 and loops to the highest index of data
  3:     total is assigned total + data[i]
  4: Return the result, total

Procedure: Main
----------------------------
Local Variables:
 *: my_data (array containing 3 Double values) - data array
Steps:
  1: Call Populate Array ( my_data, 3 )
  2: Output 'Sum is ', and Sum ( my_data )
  3: ...
```

Sum	
data[]:	...
size:	3
i:	3
total:	22.21
result:	92
Instruction:	Step 2

Main	
my_data[]:	10.0 -5 17.21
Instruction:	Step 2

```
Enter value  1: Ten
Please enter a number.
Enter value 1: 10.0
Enter value 2: -5
Enter value 3: 17.21
```

Figure 6.26: Example of the visualisation of the notional machine used in the Programming Arcana


5.5. UNDERSTANDING CONTROL FLOW

5.5 Understanding Control Flow

This Chapter has introduced new statements that can be used to co actions the computer performs. These statements allow you to add Br paths to your code. The flowcharts presented in Section 5.2 are a great order in which the computer will execute the instructions. To help you f concepts this section will look at how these statements work within the

5.5.1 Understanding Branching in Perform Guess

Figure 5.53 shows the flowchart for the Perform Guess that was develop Designing Control Flow for Perform Guess. The following sections sh executes these actions. These illustrations will start at the call into Pe the illustration of the steps that lead up to this call.

Figure 5.53: Logic for the Perform Guess Procedure from Figu

In the following illustrations Perform Guess will be called three times wi being 37 in each case. The following three guesses will be performed, e through the flowchart are covered.

1. On the first guess the user enters a guess of 50, allowing for the le flowchart to be followed.
2. The second guess will be 25 to test the middle branch, taking the e decision and the *true* branch of the second decision.
3. Finally the third guess will be 37, testing the right most path thro

CHAPTER 5. CONTROL FLOW

Perform Guess is called for guess 1

In the Guess that Number program, the Perform Guess function is respon user's guess and giving them feedback. Figure 5.54 shows the Perform G for the first time, it is passed 1 to its num guess parameter and 37 to its ta

Figure 5.54: Perform Guess is called for the first time

Note
- In Figure 5.54 the indicated areas show the following:
  1. Perform Guess is called, with 1 being passed to num guess target.
- At this point the previous code would have output 'I am thinking the Terminal.
- The values in guess and result have not been initialised, so they h was in that memory location previously.

352

CHAPTER 5. CONTROL FLOW

Loop condition is checked at the end of guess 1, with the loop being repeated

At the end of the loop the condition is checked, in this case the loop will run again.

Figure 5.70: Condition indicates that the loop's body should be executed again

Note
- In Figure 5.70 the indicated areas show the following:
  1. The condition is checked, and the expression is false.
- With **repeat...until** you can evaluate the expression by:
  1. Guess Num >= MAX_GUESSES is 1 >= 7 , this is **false**
  2. Got it, this is a variable, its value is **false**
  3. Or the above together, false or false , this is **false**, repeating the loop.
- With **do...while** you can evaluate the expression by:
  1. Guess Num < MAX_GUESSES is 1 < 7 , this is **true**
  2. Got it, this is a variable, its value is *false*, so !Got it, is not false, is **true**
  3. And together these results, true and true is **true**, repeating the loop.

C
For C you will need to code this as a C Do While Loop. The code for this will be do...while(guess_num < MAX_GUESSES && !got_it);

Pascal
For Pascal you will need to code this as a Pascal Repeat Until Loop. The code for this will be repeat...until (guess_num >= MAX_GUESSES) or (got_it);

368

Figure 6.27: Examples pages from the Programming Arcana illustrating how the concepts worked to instruct the notional machine

6.3.3 Use and Evaluation of the Programming Arcana

The Programming Arcana was able to meet all of its core requirements, though the use of multiple media formats could be enhanced if the book transitioned to an e-book format. Reflections from teaching staff indicate that it has provided a valuable tool in teaching the introductory programming unit. Staff have received a number of messages from students indicating how valuable they found the resource, a sentiment often reflected in reflections from student portfolios. The following list outlines how the Programming Arcana helped with the delivery of the example units, supported the model described in Chapter 4, and reflected the principles stated in Chapter 3.

- The Programming Arcana embodies the following principles:
  - Explanations aim to guide students to deeper understanding of the concepts associated with the introductory programming unit. (P1, P2, and P11)
  - Each section focuses on important aspects, avoiding unnecessary details. (P4)
  - Examples and explanations of concepts, syntax, and execution help support student learning. (P6)
  - Resources from the Programming Arcana provided material used in lecture notes. (P8)
  - The Programming Arcana focuses on communicating procedural programming concepts, in an appropriate manner for the two programming languages. (P10 and P12)
  - Chapters are organised so that later chapters built upon concepts presented in earlier chapters. (P11)
- The Programming Arcana supported the use of the model in delivering the introductory programming unit by:
  - Providing details that were removed from lectures, to enable these to be more interactive.
  - Aligning content presented in the text, with the material from lectures and core tasks, thereby providing a consistent experience for students.
  - Supporting the focus on concepts, providing students with additional explanations and examples.
  - Providing resources to aid students with the core tasks.

6.3.4 Summary

The Programming Arcana textbook demonstrates how the concept-based approach of the model can be embedded down to the syntax level. The text provides students with the details related to programming concepts, how they apply to program design, the associated syntax, and details on how they work within a notional machine. A range of learning styles are supported through the presentation of the syntax and concepts using both images and text. Overall, the Programming Arcana supported the concept-based approach to teaching introductory programming by providing students with details on the concepts, their application, syntax and operations.

6.4 Video Podcasts to Support the Programming Text

While it was able to meet most of its requirements, the one area where the Programming Arcana was limited was its use of static text and images, a limitation of its format. To provide students with an alternative medium, a number of video podcast series were created and made available to students via iTunesU. Three series were created in total, and each took a different approach to what should be presented. The first series, "Object Oriented Programming", covers object oriented programming principles and focused on communicating these concepts with little coverage of language syntax. "Learning Programming with SwinGame" was the second series created, and focuses on communicating Pascal syntax. The third series, "Introductory Programming", focuses on a combination of the two, presenting the concepts and the syntax together. Each of these series is discussed in the following subsections.

6.4.1 Requirements

The following list outlines the requirements for the video podcasts. This is divided into general requirements for all podcasts, and specific requirements for the individual series.

- All video podcasts were required to:
  - Clearly present a number of concepts or syntax.
  - Provide live coding demonstrations.
  - Be small in size, enabling fast downloads, while ensuring code was still readable.
  - Incorporate Swinburne's introduction and summary video material.

  - Be accessible published to iTunesU with associated meta-data.
- The Object Oriented Programming series was required to:
  - Provide all content traditionally delivered in lectures.
  - Enable a classroom “flip”, where lectures are used to discuss content and the video podcast takes the place of the traditional lecture.
- The Learn Programming with SwinGame series was required to:
  - Demonstrate Pascal programming syntax.
  - Each video should focus on a single piece of syntax, and demonstrate its use in general and in relation to programming small games.
- The Introductory Programming series was required to:
  - Provide a summary of lecture content of lecture content.
  - Demonstrate C programming syntax.

6.4.2 Video Podcasts Solution

Each of the Video Podcast series was developed using a number of video editing tools. Video from coding demonstrations, images of presentation slides, voice over audio, and character animations were all combined together in the video editing software to create each video podcast. Typical processing involved first recording coding demonstrations, and exporting presentation slides, then recording voice over audio. There were then combined with a video editing tool, which involved adjusting slide and video speed to match audio.

Table 6.3 provides an overview of the three podcast series. The Object Oriented Programming series was used by the object oriented programming unit, and provided traditional lecture style content. Learning Programming with SwinGame focused on presenting Pascal programming language syntax, as a more dynamic extension of the Programming Arcana. Whereas the Introductory Programming series provided both lecture style content, and coding demonstrations.

Table 6.3: Video podcast series details

Series Name	Episodes	Episode Length
Object Oriented Programming	19	6-36 minutes
Learning Programming with SwinGame	26	2-15 minutes
Introductory Programming	7	17-38 minutes

6.4.3 Use and Evaluation of Video Podcasts

All three podcast series were used to support students learning in the example units discussed in Chapter 5. The podcasts provided students with an alternative medium for approaching the concepts and syntax that was included in the Programming Arcana, with the expectation that students would be able to use both resources to support the construction of their knowledge.

The following list relates the video podcasts to the principles stated in Chapter 3.

- Video podcasts provided an additional source of material students could use to help construct their own knowledge. (P1)
- Content presented in the video podcasts aligned with the intended learning outcomes of the example programming units. (P2)
- The video podcasts provided another set of resources to help support student learning efforts. (P6)
- In each case, episodes demonstrate appropriate use of programming languages. (P12)

In addition to the above list, it was felt that the Learning Programming with SwinGame series had also demonstrated use of the following principles. These principles were not present in the other series which tended to focus on information provision, with a wider set of concepts.

- Episodes focused on demonstrating a single statement, and provided a number of examples to illustrate its use. (P4)
- The small focus of these podcasts helped reduce time needed to produce the series. The specific nature of the series also allowed it to be re-used when activities and lecture sequences changed. (P8)
- Podcasts provided support for language syntax, allowing other activities to focus on underlying concepts. (P11)

Interestingly, the Learning Programming with SwinGame series proved to be the most useful of the three series. Its clear focus on communicating programming syntax allowed it to be used more flexibly, as it did not depend on other teaching and learning activities. In contrast, the other two series inclusion of concepts meant that they had a greater dependence on how, and in which order, topics were covered in the unit. The use of a larger number of more specific episodes in the Learning Programming with SwinGame series, also made it easier to incorporate the videos in relevant lecture and laboratory notes.

Teaching staff indicated that future video podcasts would continue to use the format of the Learning Programming with SwinGame series. Specifically, the narrow focus and short duration. New series are currently planned to help demonstrate how to achieve certain effects in game programs developed using the SwinGame library. These podcasts would demonstrate how to combine a number of concepts in creating slightly larger programs.

6.4.4 Summary

Video podcasts provided students with an alternative means for studying unit concepts and programming language syntax. It was found that specific, short duration, podcasts had a longer lifespan as they worked together with teaching and learning activities.

6.5 Chapter Summary

This chapter has discussed four resources used to support *how* and *what* was taught in the example units from Chapter 5. Doubtfire helped to support the use of formative feedback during the semester, with visual feedback being used in lieu of marks to help motivate students. SwinGame provided a game library to help students create rich and interactive programs. The Programming Arcana provided students with details on the concepts, their application, associated syntax, and operation on the notional machine. Which was also supported by the video podcasts, which provided students with an alternative means of study.

Chapter 7 presents a further evaluation of these resources, the teaching and learning activities from Chapter 5, and model from Chapter 4.

7

Evaluation of the Teaching and Learning Context

This chapter presents the results from a number of small studies into the effectiveness of the model presented in Chapter 4. The units analysed match those described in Chapter 5, and they made use of the resources discussed in Chapter 6.

Section 7.1 describes the action research method, and thematic analysis approach, used in this work and outlines how the ethical issues related to analysing student work were addressed. This is followed by Section 7.2 that presents a discussion of the evolution of the model, and the associated resources and assessment criteria, across all iterations of the action research process. Section 7.3 and Section 7.4 provide the results and discussions from the thematic analysis of student portfolios from two teaching periods. Section 7.3 discusses issues that students reported in their reflections, while Section 7.4 examines student progress as depicted by the burn down charts included in student portfolios.

7.1 Research Design

7.1.1 Action Research

Due to the practical and applied nature of this research – with its focus on student learning, and the embedded reflective process – it was decided to follow a Practical Action Research (Creswell 2008) design based on Mills' (Mills 2010) *dialectic action research spiral*. This model, shown in Figure 7.1, includes a four step process: (1) identify an area of focus, (2) collect data, (3) analyse and interpret the data, and (4) develop an Action Plan.

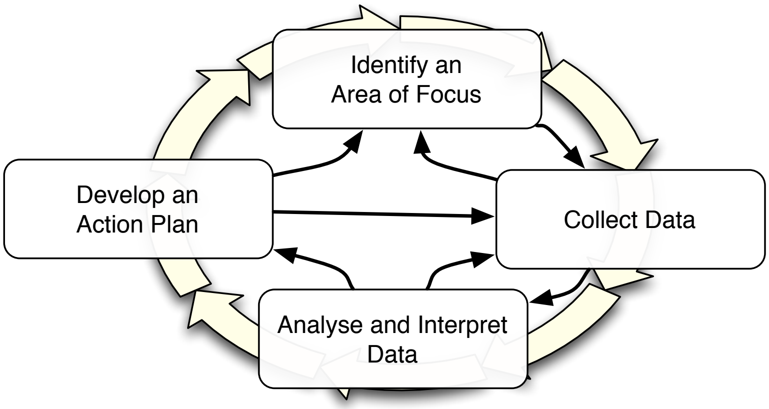


Figure 7.1: A visual representation of Mills (Mills 2010) Dialectic Action Research Spiral

Iterations in the work presented aligned to teaching periods that included the delivery of one or more of the programming units discussed in Chapter 5. Each iteration included an *action plan* related to implementing the approach from Chapter 4, which influenced the *focus* for the iteration, the data collected, and the analysis performed.

The overall focus for this research was on the development, application and iterative improvement of the model from Chapter 4. The iterative nature of the action research process meant that the specific focus in each teaching period addressed relevant aspects of the model, based on its state at that time and feedback from previous iterations.

Data collection included analysis of student portfolios, student grades, unit documentation and staff reflections, as illustrated in Figure 7.2. The Unit Outline, prepared prior to the start of the teaching period, documented the intended learning outcomes and assessment criteria around which the unit delivery was focused. Student work

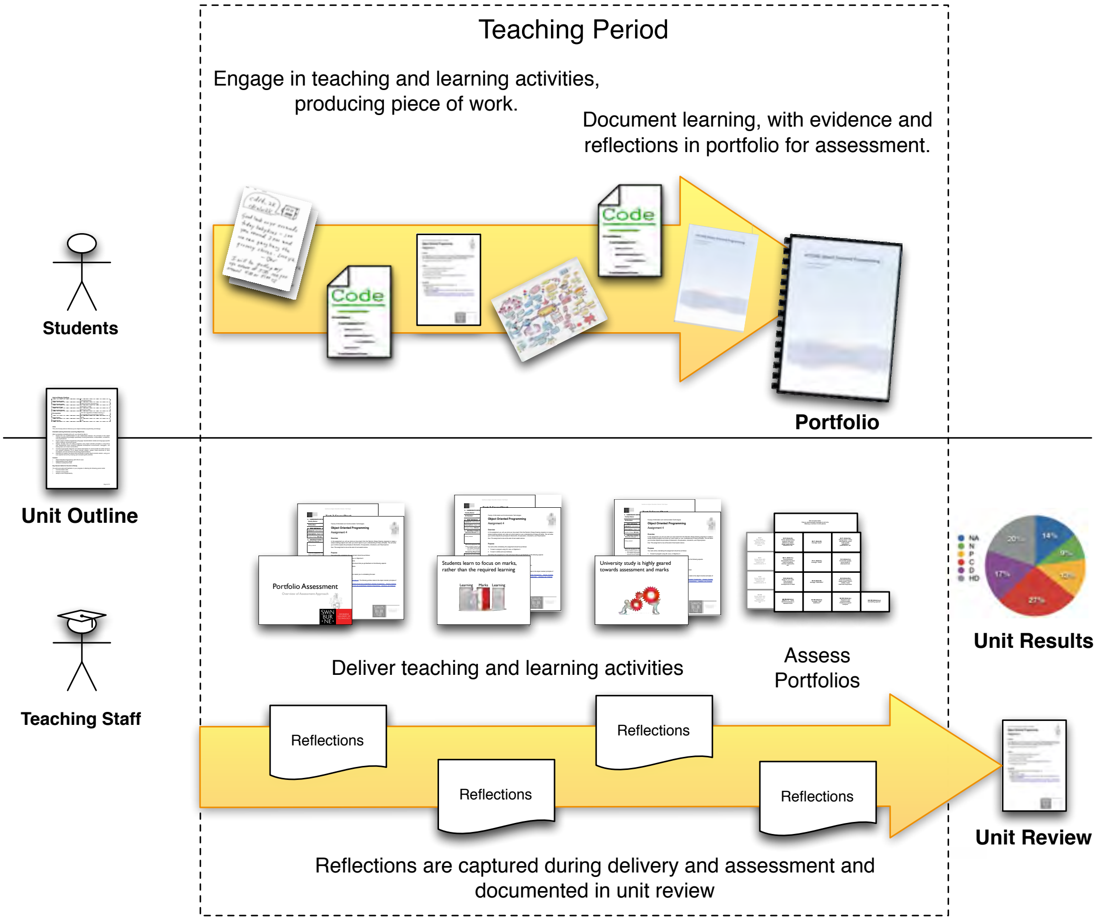

Teaching Period
Engage in teaching and learning activities, producing piece of work.
Document learning, with evidence and reflections in portfolio for assessment.
Students
Code
Code
Portfolio
Unit Outline
Deliver teaching and learning activities
Assess Portfolios
Unit Results
Teaching Staff
Reflections
Reflections
Reflections
Reflections
Unit Review
Reflections are captured during delivery and assessment and documented in unit review

Figure 7.2: Various documents used in the data collection for this research

from the teaching period was collected and submitted for assessment in student portfolios. Students could opt to make this work available to the action research project using the process outlined in Section 7.1.3 to address ethical considerations. Reflections from Teaching Staff, Unit Results and Unit Reviews also provided data for the action research project.

Thematic analysis (Braun & Clarke 2006) was used to examine reflections in student portfolios, which is discussed further in Section 7.1.2. As the thematic analysis of student portfolios was a time consuming process, it was not performed in all iterations. In teaching periods where a thematic analysis of the portfolios was not performed, the student grades and staff reflections provide insights into the composition of student portfolios.

Unit documentation included the Unit Outline and Unit Review documents. The Unit Outline documented the intended learning outcomes and assessment criteria used in the given teaching period. This document was provided to students prior to the commencement of classes and was an actively used and referred to by both teaching staff and students. At the conclusion of the teaching period, and after results were reported, a Unit Review document was created. This document captured details of student perceptions of the teaching, teaching and learning approach, results, unit management, and any planned changes for future delivery of the unit. These documents were prepared by the Unit Panel, and included input from all teaching staff.

Staff reflections indicate the overall and particular qualities exhibited in the student portfolios for a given semester. Staff reflections were captured both during the semester and after portfolios were assessed. These reflections were recorded in personal log book notes, which in most cases were summarised in the Unit Review document.

Student grades provide an indication of how well students performed in the given semester. Together with the staff reflections, results provide some insight into the learning outcomes students achieved – insights not available by considering students grades alone.

Figure 7.3 illustrates the interactions between the activities in the action research project, and the teaching and learning activities for an individual teaching period. The action research activities, on the left, provided inputs to help refine the model and inform adjustments to the Unit Outline. These activities coincided with the teaching and learning activities associated with defining the intended learning outcomes, and constructing the assessment criteria.

Interestingly, the Unit Review document provided input into the data collection, while also incorporating inputs from the analysis and interpretation of the data. This was achieved in two phases, initially the teaching staff prepared the unit review incorporating their reflections on the teaching period as well as data from student surveys and unit results. The Unit Review document was then used as part of the action research project, which further analysed the data available. The outputs of this analysis were then added to the Unit Review, to ensure they helped inform subsequent iterations.

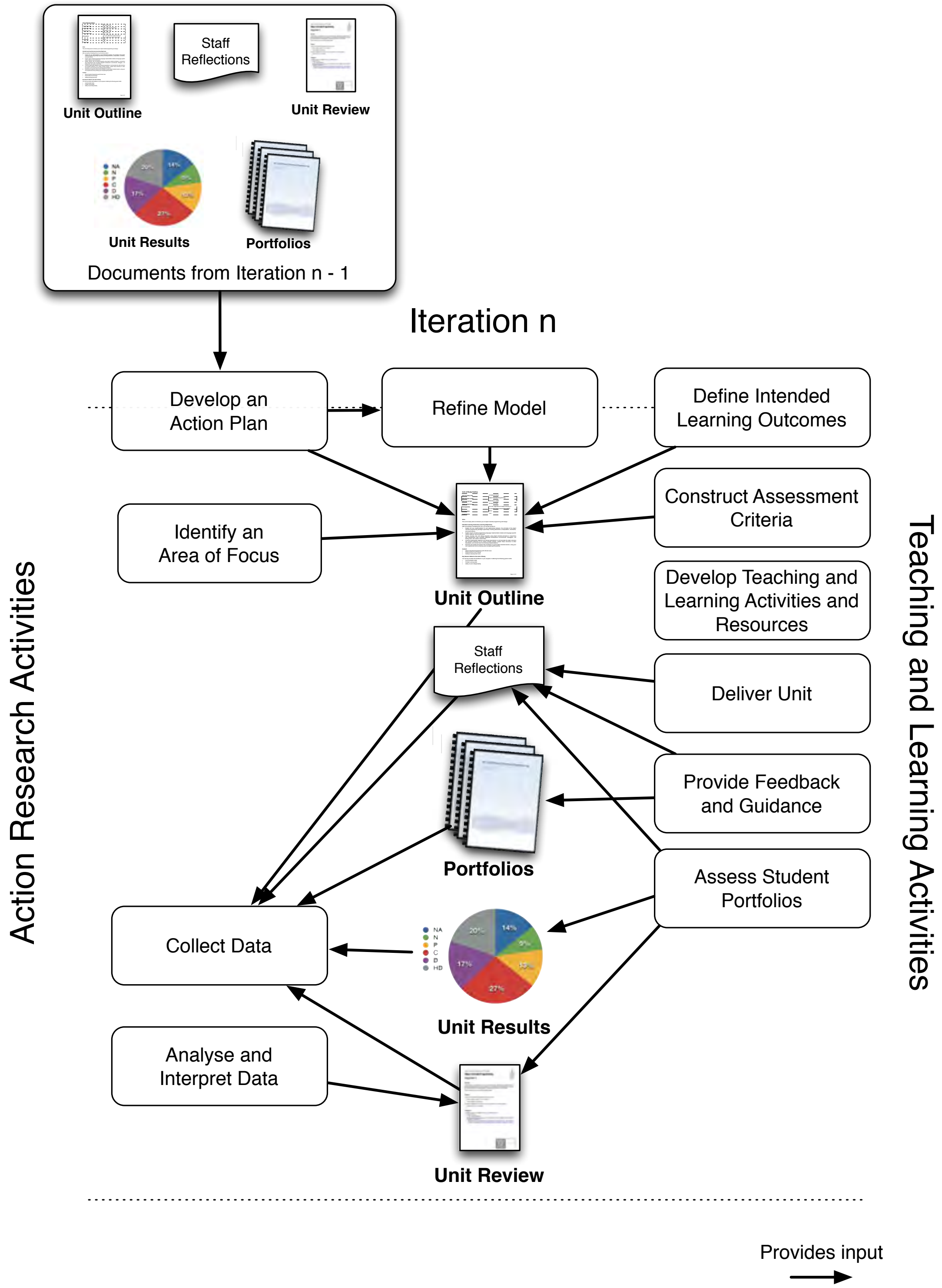

Unit Outline
Staff Reflections
Unit Review
Unit Results
Portfolios
Documents from Iteration n - 1
Iteration n
Develop an Action Plan
Refine Model
Define Intended Learning Outcomes
Identify an Area of Focus
Construct Assessment Criteria
Unit Outline
Develop Teaching and Learning Activities and Resources
Staff Reflections
Deliver Unit
Portfolios
Provide Feedback and Guidance
Assess Student Portfolios
Collect Data
Unit Results
Analyse and Interpret Data
Unit Review
Action Research Activities
Teaching and Learning Activities
Provides input

Figure 7.3: Interactions between Action Research Activities and Teaching and Learning Activities, and their input into various documents.

7.1.2 Thematic Analysis of Reflections

Reflections in student portfolios provide a wealth of information. To help identify themes and patterns in the portfolios it was decided to perform a thematic analysis using the process outlined by Braun & Clarke (2006). This process involves six phases, with some terminology[1] adapted for clarity:

1. Familiarising yourself with the data
2. Generating initial themes,
3. Searching for strong themes
4. Reviewing themes
5. Defining and naming themes
6. Producing the report

In each teaching period where a thematic analysis was performed, familiarity with the data was obtained early in the process with all portfolios being read as part of the unit assessment. At the end of the unit assessment, teaching staff made notes related to general issues, progress, and the overall quality of portfolios. This was part of standard unit delivery procedures, with the resulting reflections being summarised in the Unit Review document as mentioned in Section 7.1.1.

Once the portfolios were made available for this research initial themes were generated by revisiting the reflective component of each portfolio and looking for the qualities under examination. These themes were then documented, and recorded in a spreadsheet. Spreadsheet software was used to collate the themes and record the portfolio details of where these themes had been mentioned, along with any illustrative comments using the students own words.

In phases 3 through 5 the identified themes were broadly grouped together, and then each broad group was examined for sub-themes. All of the themes identified in the examination of student portfolios were maintained in the final categorised results. Themes that did not clearly relate to any of the identified groups were grouped together as a miscellaneous group.

In the reporting of this analysis the raw results are presented, grouped into the identified themes. Illustrative quotes from student reflections are provided to help define the themes.

[1] The term "code" has been changed to "theme" to avoid confusion relate to the use of this term in computer science.

7.1.3 Addressing Ethical Concerns

In studies related to an educational context, the student-teacher relationship can be a source of potential ethical problems, with perceptions of coercion being the central concern where the research is also involved in teaching the students. Participants in this study were students of the investigators, and so it was important to design an appropriate process whereby students could offer informed consent to participate in the research with no risk of coercion.

This issue was further complicated as the researchers at times taught both the first and second programming units. For example students could undertake the introductory programming unit in the first half of the year, and the object oriented programming unit in the second half of the year. As most students who completed the first unit progressed to the second unit the following semester, there was the potential for the perception of coercion in this second unit.

An appropriate research protocol was developed, and granted ethical approval from Swinburne's Human Research Ethics Committee. Figure 7.4 shows an overview of the protocol approved and used.

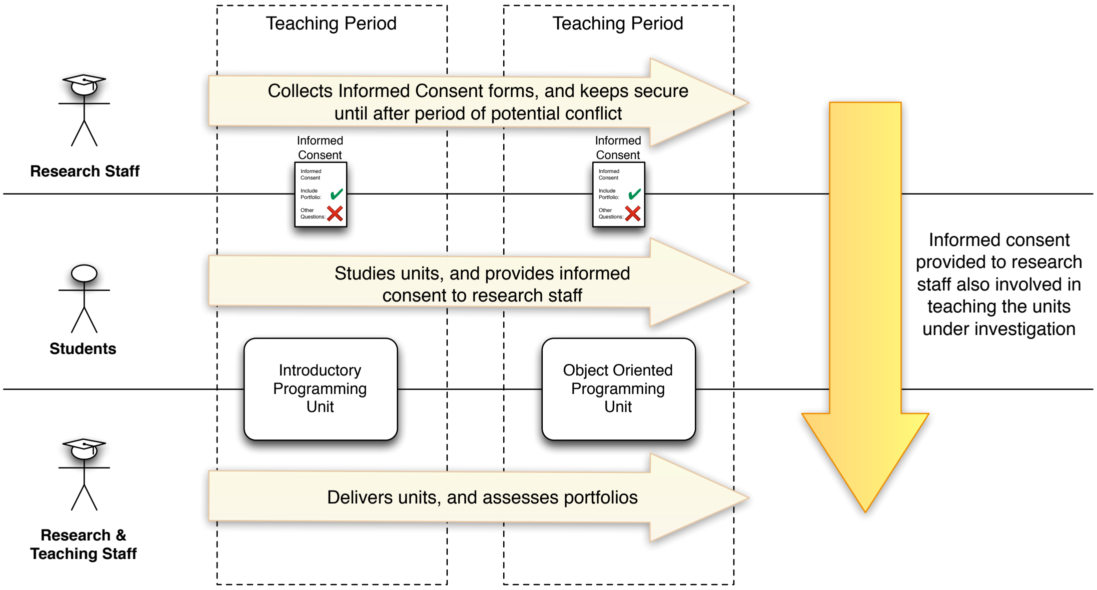

Teaching Period
Teaching Period
Research Staff
Collects Informed Consent forms, and keeps secure until after period of potential conflict
Informed Consent
Informed Consent
Include Portfolio: ✓
Other Questions: ✗
Informed Consent
Informed Consent
Include Portfolio: ✓
Other Questions: ✗
Students
Studies units, and provides informed consent to research staff
Introductory Programming Unit
Object Oriented Programming Unit
Research & Teaching Staff
Delivers units, and assesses portfolios
Informed consent provided to research staff also involved in teaching the units under investigation

Figure 7.4: Overview of the protocol used to avoid perceptions of coercion across all units involved in this research

Students were informed that participation in the research was voluntary, and their participation would in no way influence their results or relationship with the university. Informed consent was then gained using a printed Informed Consent form, which was distributed to students in lectures during each teaching period. Forms were completed by all students, and indicated their willingness to have their work included in the research. All students were required to complete and sign the form, so that it was not be possible to determine those who did, or did not, wish to participate simply by observing those who signed the form.

To avoid issues of coersion in the second programming unit, the Informed Consent forms were withheld from researchers involved in teaching these students until the period of potential influence was deemed to have passed. Until such time, the forms were kept in a sealed envelope in a locked cabinet at Swinburne by the researchers who were not directly involved in teaching of the units.

The period of potential influence was deemed to have passed based on the following:

- For the introductory programming units where the researchers were not involved in teaching a follow on unit, the period of potential influence was deemed to have passed once the results for the introductory programming unit were published.
- For the introductory programming units where the researchers were involved in the teaching the follow on unit, the period of potential influence was deemed to have passed once the results for the *second* unit were published.
- For the object oriented programming units, the period of potential influence will be deemed to have passed once the results for the unit were published.

7.2 Lessons Learnt through Action Research

The action research process was used in the development, application, and evaluation of the model presented in Chapter 4. A total of nine iterations were completed over a five year period, involving thirteen unit deliveries, with a total of 983 portfolios assessed. This section reports on the development of the model, its guiding principles, the teaching and learning activities and supporting resources.

7.2.1 The Units

Chapter 5 presented details of two example implementations of the model. These examples represent the current status of this research, which evolved iteratively from the delivery of four separate programming units: two introductory programming units, and two object oriented programming units. Table 7.1 shows the four different programming units, and the iterations in which they were involved. All of the units were taken by undergraduate students early in their degree programme and were convened by the author.

Table 7.1: Units in each iteration.

Units \ Iteration	1	2	3	4	5	6	7	8	9	Current
Introductory Programming (A)	✓		✓		✓	✓		✓		✓
Introductory Programming (B)							✓	✓	✓	✓
Object Oriented Programming (A)		✓		✓			✓		✓	✓
Object Oriented Programming (B)									✓	✓

General details of the four units follow, and any changes to individual iterations are presented in the following sections.

Introductory Programming (A)

Introductory Programming (A) was typically taken by students in their first semester of their degree programme, and introduced them to procedural programming, as outlined in Chapter 5. The intended learning outcomes included: the ability to read and interpret code; write small procedural programs; iteratively use modular and functional decomposition to break problems down; and the ability to apply the principles of structured programming (focusing on blocks of code and using sequence, selection, and repetition). Outcomes were expressed in a programming language neutral manner as the focus of the unit was on the underlying programming concepts.

Introductory programming (A) was taken by students studying a range of degrees. Most students were enrolled in a Bachelor of Science majoring in Computer Science, Professional Software Development or Games Development.

Object Oriented Programming (A)

Object Oriented Programming (A) had Introductory Programming (A) as a prerequisite and was predominantly taken by students in their second semester. As outlined in Chapter 5, the intended learning outcomes in this unit required students to: design, develop and test object oriented programs; communicate the underlying principles of abstraction, encapsulation, inheritance and polymorphism; use professional software development tools; and describe factors that influence the quality of an object oriented program. As with Introductory Programming (A), the outcomes were expressed in a language neutral manner and the focus was on underlying concepts.

The student cohort in Object Oriented Programming (A) consisted only of students that had completed Introductory Programming (A).

Introductory Programming (B)

Prior to iteration seven this unit was taught using a common textbook style approach with assignments and a final exam. In iteration seven the unit was adapted to a portfolio-based approach, and then combined with Introductory Programming (A) from iteration eight.

Intended learning outcomes for Introductory Programming (B) covered similar topics to Introductory Programming (A) but with specific reference to the C programming language. When this was combined with Introductory Programming (A) in iteration eight, the combined outcomes matched those from Chapter 5, and the focus shifted from language syntax to programming concepts.

The cohort of Introductory Programming (B) included students from a range of degree programmes. This included students studying for a Bachelor of Information and Communication Technology, Bachelor of Engineering and Bachelor of Science (Computer Science and Software Engineering). The unit was included in a number of other degrees as an elective.

Object Oriented Programming (B)

As with Introductory Programming (B), Object Oriented Programming (B) was taught using a specific language (C++), used a textbook style approach, traditional assignments and final exam. This unit covered the same topics as Object Oriented Programming (A), and in iteration nine the two object oriented programming units were combined into a single unit. This combined unit used portfolio assessment and its intended learning outcomes matched those from Chapter 5, and focused on programming concepts and principles. Students continued to enrol in the individual units, but were taught as a single cohort. Students enrolled in Object Oriented Programming (B) were required to include evidence in their portfolios of being able to apply the unit's concepts using the C++ language.

Relationship Between Units

Figure 7.5 shows progression paths through these units. The students we broadly classified as being enrolled in a *software development* focused degree took Introductory Programming (A) in the first semester of their first year, and then Object Oriented Programming (A) in the second semester of their first year. Introductory Programming (B) was taken primarily by student enrolled in an Engineering degree, who subsequently took an intermediate programming unit before studying Object Oriented Programming (B). The intermediate programming unit aimed to further develop students' programming knowledge with the C and C++ programming languages, and included a brief introduction to objects in the last few weeks. For the Engineering students, this sequence may be extended over more than three consecutive semesters depending on their degree programme.

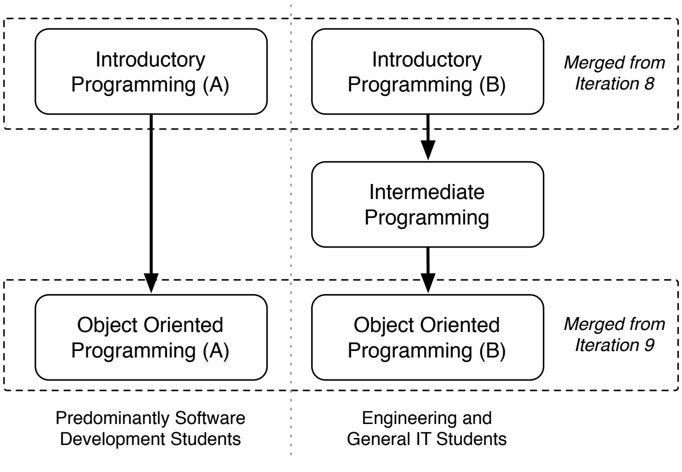

Introductory Programming (A)
Introductory Programming (B)
Merged from Iteration 8
Intermediate Programming
Object Oriented Programming (A)
Object Oriented Programming (B)
Merged from Iteration 9
Predominantly Software Development Students
Engineering and General IT Students

Figure 7.5: Progression pathways through the introductory programming units

7.2.2 Prior to Portfolio Assessment

Prior to Iteration 1, the introductory programming units at Swinburne were taught using some of the principles stated in Chapter 3. The principles incorporated in these units are outlined in Table 7.2, and described in the following list.

- Lectures incorporated a number of constructivist learning theories (P1), but the units used more traditional forms of assessment with assignments, tests, and exams.
- Teaching staff aligned assessment tasks to unit learning outcomes (P2), but this activity was done informally, was not documented, and did not involve the students.
- Lectures did focus on trying to communicate a few important aspects (P4), though it lacked the clarity that evolved in later iterations.
- Students were actively supported throughout the teaching period (P6).
- Staff were willing to change (P8), but were not focused on adopting principles from agile software development.
- The unit focused on concepts associated with a single paradigm (P10).
- Concepts (P11) were central to the teaching, but lacked the clear focus that developed in later iterations.
- Programs students created made appropriate use of the programming languages used (P12).

Table 7.2: Principles related to the introductory programming units prior to converting to constructive alignment with portfolio assessment. The ✓ indicates a principle in the form presented in Chapter 3, $\sim$ indicates partially present but neither in planned focus nor in final form.

Iteration	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Prior	$\sim$	$\sim$		$\sim$		✓		$\sim$		✓	$\sim$	✓

Assessment in the programming units consisted of assignments, tests, and an exam. While pass rates were good, the teaching staff felt that in many cases final results did not accurately represent student capabilities. Marking assignments and exams often identified significant misunderstandings that were not correctly reflected in the quantitative assessment schemes used. Often these issues were uncovered in the final exam, when it was too late to provide students with feedback. When misunderstandings were identified in assignment submissions, feedback appeared to be ignored by most students as they focused on the numeric grade attached to the work, an observation that is consistent with the findings of Black & Wiliam (1998).

Additionally, staff felt that examinations were not ideal for assessing programming capabilities, a sentiment also echoed by other academics teaching introductory programming units (Sheard et al. 2013). As discussed in Section 4.1.1, test and exams poorly align with constructivist learning theories, which see test performances as providing poor indicators of the conceptual models students had constructed Ben-Ari (2001). This, together with the general feeling that grades were not accurately reflecting learning outcomes, was the primary motivation for trialling portfolio assessment in Iteration 1.

It is interesting to note that the introductory programming units prior to Iteration 1 could be considered to have already embodied the core principles of constructive alignment. Teaching and learning activities aimed to actively engage students, adopting aspects from constructive learning theories (P1), and the unit content and assessment was aligned to learning outcomes by the teaching staff (P2).

In terms of constructivist learning theories, the introductory programming units made explicit use of a notional machine as described by DuBoulay (1986), a practice that Ben-Ari (1998, 2001) indicated as essential for the adoption of constructive learning theories in computer science. Additionally, the lectures made use of many of the strategies described in Chapter 4, including the use of story structure for lecture presentations (Atkinson 2007) and interactive coding demonstrations (Van Gorp & Grissom 2001). While these practices had previously been used, they were the result of intuition rather than explicitly related to constructive learning theories.

Ideas from aligned curriculum were also present, though the processes of aligning assessment tasks and lecture content to intended learning outcomes was done informally and was not documented as part of the unit delivery. However, in reflection it was possible for teaching staff to align each of the assessments and weekly topics back to the unit's intended learning outcomes.

7.2.3 Early Iterations

Iteration 1

Focus This first iteration attempted to implement constructive alignment, with aspects of portfolio assessment. This involved clearer implementation of constructivist learning theories (P1) and aligned curriculum (P2) as indicated in Table 7.3.

Table 7.3: Principles related to the introductory programming unit in Iteration 1. Focus is indicated by $\star$, $\checkmark$indicates the principles is in the form presented in Chapter 3, $\sim$ indicates partially present but neither in planned focus nor in final form.

Iteration	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Iteration 1	$\star$	$\star$		$\sim$		✓		$\sim$		✓	$\sim$	✓

Action Plan The approach used was an iterative step toward the general model presented in Chapter 4. Students were required to complete six assignments and six tests, with the *option* to include a portfolio. The aim of the assessment strategy was for students to demonstrate their understanding of core concepts in the assignments and tests, with the portfolio used to determine student ability in the higher grade brackets. Weights were applied to each of the assessment items and these were added together to calculate the final grade.

In terms of the model from Chapter 4, this iteration lacked many of the aspects defined. Intended learning outcomes had been defined, there were some assessment criteria (though they lacked the clarity of later iterations) teaching and learning activities were presented to students with lectures including interactive components, and students did present a portfolio of work for assessment. The major difference was the lacked of the iterative formative feedback process. The dynamic from this iteration was more consistent with "standard" teaching and learning environments teaching staff had experienced before implementing constructive alignment, with the portfolio representing yet another assignment.

Key differences from the portfolio model presented in Chapter 4:

- The unit included a total of eleven intended learning outcomes, each making use of active verbs and relating to specific parts of the unit.
- It used multiple assignments (six) over the semester.
- It included tests that were marked and contributed to the final grade.
- Submission of a portfolio was optional. All students who submitted a portfolio were interviewed.

Similarities with later portfolio iterations included:

- It included qualitative criteria for each grade, though these were described in general terms.
- Students included a self assessment against the criteria.

Key differences from the introductory programming unit described in Chapter 5 included the following:

- Focus was on programming concepts, but there was some crossover of topics in the early material.
- The teaching and learning activities made limited use of SwinGame.
- Procedures were introduced later (SwinGame was not used in the early parts of the teaching period).
- Students were only briefly introduced to the C programming language at the end of the unit.
- None of the supporting tools from Chapter 6 had been developed at this stage.
- Lecture slides closely followed the "Beyond Bullet Points" approach (Atkinson 2007), and included extensive notes on each slide.
- An earlier edition of the Pascal Language Reference (Van Canneyt 2013) was used as the unit text.

Data Unit results across all iterations are shown in Table 7.4, which lists the number of students receiving each grade over the nine iterations. Grades include those students who enrolled but did not submit a portfolio (NA) those who failed (N) and those who received Pass (P) Credit (C) Distinction (D) and High Distinction (HD) result. The results for Iteration 1 are shown in Figure 7.6. In this iteration a large percentage of students managed to receive an HD grade.

Reflections and Analysis Use of portfolios was limited in this iteration, with positive and negative results. Positive aspects of the unit delivery included the improved confidence of staff in the potential for using portfolio assessment in units related to software development. Overall, it was felt that portfolio assessment offered great potential, but that this iteration had not managed to create a suitable environment in which these benefits could be realised.

The two main issues were the weakness of the expressed assessment criteria and the combining of results from the assignments and tests. Together, these issues resulted in many students receiving a higher grade than staff felt was appropriate given the

Table 7.4: Unit Results Across Iterations 1 to 9

Iter.	Unit	NA	N	P	C	D	HD
1	Introductory Programming (A)	3	1	4	5	6	11
2	Object Oriented Programming (A)	19	7	14	19	13	11
3	Introductory Programming (A)	9	1	2	9	6	9
4	Object Oriented Programming (A)	4	7	3	17	6	5
5	Introductory Programming (A)	10	6	9	19	12	14
6	Introductory Programming (A)	14	3	28	20	14	5
7	Introductory Programming (B)	42	12	56	47	22	7
	Object Oriented Programming (A)	5	8	18	10	8	9
8	Introductory Programming (A)	11	5	20	21	17	14
	Introductory Programming (B)	39	47	84	36	20	9
9	Introductory Programming (B)	61	4	78	24	25	7
	Object Oriented Programming (A)	14	0	25	6	14	7
	Object Oriented Programming (B)	7	0	25	5	3	4

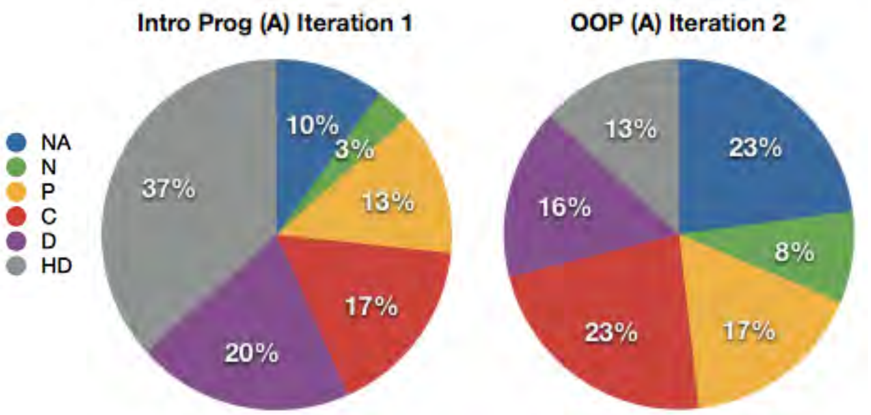


Figure 7.6: Result distributions from Iterations 1 and 2, note in particular the shift in the number of High Distinctions.

outcomes demonstrated in student portfolios. This is supported by the high portion of students with HD grades when informally compared with other unit results.

Staff reflections included several interesting aspects related to the assessment in this iteration:

- There were too many intended learning outcomes. This had made it difficult for staff to clearly communicate how teaching and learning activities related to the outcomes. Students had also found it difficult to relate their portfolio pieces back to the intended learning outcomes.
- Criteria were difficult to apply, being weakly defined, and student interpretations tended to weaken the criteria further in their self assessment.
- Significant effort had been put into creating "Beyond Bullet Point" slides and associated notes, which had been useful in terms of delivery but restricted opportunities to adapt the material.
- Assessing the portfolios was very time consuming due to the loosely defined criteria. Staff needed to "extract" value from the portfolios, rather than the emphasis being on students demonstrating understanding.
- The time consuming nature of the portfolio assessment meant that teaching staff did not feel the approach could scale to larger class sizes.
- Staff felt confident that a portfolio of work could provide a suitable means of assessing student outcomes in technical units, addressing a key concern at the time.
- Core assignments and tests:
  - Covered the minimum expectations for the intended learning outcomes.
  - Received high marks, which only indicated basic coverage of the intended learning outcomes.
- Students with weak portfolios, showing shallow coverage of the intended learning outcomes, were still able to receive high grades indicating the assessment strategy was poor.

Development of Principles The first experience of delivering a portfolio assessed unit had been informed by examining the principles of constructive alignment, and the experience provided a number of insights that helped form the principles stated in Chapter 3. These included:

- Constructive learning theories (P1) and aligned curriculum (P2) had been at the centre of this experience. However, the utility of both had been hampered by the large number of intended learning outcomes. This guided the importance of having a small number of more highly targeted outcomes.

- The portfolios had been able to capture student outcomes, but results had been impacted by the use of weighted assessment used to motivate students during the semester. While alternative schemes, or weightings, could have been developed, it was felt these incentives were not required and a greater use of formative feedback would be of greater benefit to students. This emphasis on formative feedback evolved into Principle 3 and influenced a change from a soft Theory X to Theory Y (P7) perspective.
- Teaching staff were *willing to change*, and wanted to be able to adjust teaching material in response to student issues. However, the significant effort that had gone into the develop of the "Beyond Bullet Points" lecture slides had created a resistance to change. This conflict started a change in attitude that resulted in the definition of Principle 8; the aim to be both *agile* and *willing to change*.

Iteration 2

Focus Iteration 2 included the delivery of Object Oriented Programming (A), and aimed to address several of the main concerns from Iteration 1 by having *fewer intended learning outcomes* around which everything would be based, assessing each student's outcomes as a whole with *100% portfolio assessment*, and using *specific assessment criteria* to express what was expected for each grade. In addition to this, the unit material was separated into teaching and learning *activities* and teaching and learning *resources*, in an effort to enable greater flexibility with future changes.

Table 7.5 shows the principles related to this iteration, highlighting the additional focus on frequent formative feedback (P3), adoption of Theory-Y perspectives on motivation (P7), and on means of effectively managing teaching and learning activities and resources (P8).

Table 7.5: Principles related to Iteration 2. Focus indicated by ⋆, present indicated by ✓, partially present indicated by ∼.

Iteration	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Iteration 2	⋆	⋆	⋆	∼		✓	⋆	⋆		✓	∼	✓

Action Plan The transition from assignments *plus* portfolio, to 100% portfolio assessment meant that this iteration did include most aspects of the model described in Chapter 4. Most notably, the iterative feedback process was initiated, and students were actively encouraged to develop pieces for their portfolios throughout the teaching period. Where the model did differ was in how the various activities were guided, as result of the absence of Principle 5 and the overly zealous application of constructive learning theories (P1).

In this iteration the following aspects of our model were included:

- The number of intended learning outcomes was reduced from eleven to five, reducing redundancy and providing a clearer focus (Principle 4).
- Assessment criteria were developed for each grade using the different levels from the SOLO taxonomy (Biggs & Collis 1982). This was presented in a format similar to that shown in Figure 5.3, though some details differed.
- Feedback was provided to students using weekly formative assessments and tests.
- Notes previously embedded in slides were shifted to a single document and distributed to students as a PDF.

The following aspects differed from Iteration 1:

- Each intended learning outcome had criteria for meeting it to differing standards: Marginal, Adequate, Good, and Excellent.
- A Credit grade required three intended learning outcomes to be addressed at an *Adequate* standard, Distinction required two at a *Good* standard (with all other adequate) and High Distinction required two *Excellent* and all others *Good*.
- A flipped classroom model (Baker 2000, Lage & Platt 2000) was adopted: student were provided with online videos covering the typical weekly lecture material and class room activities were predominantly interactive.

Key differences from the final version of the object oriented programming unit described in Chapter 5 include the following:

- A greater emphasis was placed on constructive learning theories, and a shift toward discovery learning. Concepts were presented using the video podcasts, and lecture activities included a greater emphasis on group discussions.
- Lectures used a number of interactive quizzes; questions were typically taken, then discussed in small groups, and retaken before a wider group discussion on any changes in understanding.
- While multiple languages were used, and students could only choose between the Java and C# programming languages.
- Laboratory exercises also had less guidance, and students explored their chosen language and how it could be used to implement object oriented programs.
- SwinGame was introduced early in the teaching period to enable students to

visualise object interactions through the creation of a drawing program.

Data Figure 7.6 shows the result distributions for Object Oriented Programming (A) in Iteration 2. The number of High Distinction results was closer to expectations, though the low pass rate was a cause for concern.

Reflections and Analysis Key staff reflections included:

- The shift to formative feedback had been challenging, with a great deal of anxiety for teaching staff about whether students would engage in the weekly tasks when they had not been allocated marks.
- Interviewing all students provided staff with high confidence that students had completed the work themselves, but meant that portfolio assessment was very time consuming.
- The general structure of the assessment criteria was suitable, but there was a disconnect in student perception of the required standard: the interpretation of "good" was significantly different between staff and students. Students felt that Good equated to having completed the work, whereas staff viewed this as the required standard for Adequate and Good required the work to be of a higher standard.
- The quality of work included in student portfolios was of a weaker standard than desired across all grades.
- Most students did not appear to benefit from the classroom flip, with few preparing adequately for the classroom discussions.
- It was felt that many students "coasted" along, and did not genuinely attempt the planned activities.
- Progress on understanding weekly topics was very slow, with the lack of guidance resulting in students not making the best use of their time.
- Separation of teaching and learning activities and resources had enabled a greater freedom in creating the interactive lecture, and the resources could be reused for future unit deliveries.

Staff felt that most of the issues from the semester could be attributed to the shift toward non-productive "discovery learning" (Anderson et al. 1998). In our effort to implement constructive learning theories we had reduced the amount of guided learning activities, and student productivity appeared to have been adversely affected.

It was still felt that portfolio assessment could be beneficial but that, again, we had failed to realise any significant benefits. In many ways the results from this teaching period, in terms of student learning outcomes, had felt like a backward step.

Development of Principles The teaching and learning environment created in Iteration 2 had been informed by the constructive learning theories, and a trusting Theory Y environment. At the end of this iteration it was felt that the Theory Y attitude was still appropriate, but that the overly zealous application of constructive learning theories had meant students were unable to appropriately apply themselves. The lack of guidance had resulted in many students spending too much time working out *what* it was they needed to learn, and not enough time *applying* the concepts related to object oriented programming to actually *construct* the required knowledge.

This experience influenced a number of the principles stated in Chapter 3.

- Constructive learning theories, central to Principle 1, were tempered to include stronger guidance along with the focus on the central role of the learner in constructing their own knowledge.
- We needed to more clearly communicate our high expectations of students (P5). In this iteration many students did not seem to be aware of what had been expected of them.
- The shallow responses and evidence in student portfolios also indicated the need to focus on depth of understanding (P4).

7.2.4 As the Model Stabilised

Experiences from iterations 1 and 2 had hinted at the potential for portfolio assessment, but the implementation had been unsuccessful. The changes to the model at the end of Iteration 2 resulted in a more successful application of portfolio assessment, and Iterations 3 to 6 all applied the model in a similar way.

Iterations 3 to 6

Focus The focus of Iterations 3 to 6 was on both extended how we taught the units, and developing resources to support what we taught. In relation to how we taught, the focus was on the development of assessment criteria, with the aim to ensure these were clear for both staff and students, and could be applied efficiently to assess student outcomes. At the same time, each iteration worked on extending the resources available to support what and how we taught.

Table 7.6 indicates the principles in focus for iterations 3 to 6. Iteration 3 was the first iteration in which all of the principles were present in some form, with most being actively incorporated to address limitations identified in Iteration 2. After this iteration the majority of the principles were in place, and subsequent the focus in subsequent iterations was on improving the incorporation of these principles. Iteration 5 marks the next change, with the incorporation of the hurdle tests and template for the learning summary report.

Table 7.6: Principles related to Iterations 3 to 6. This uses the same symbols as in Table 7.3 and Table 7.5: focus ⋆, present ✓, partially present ∼ .

Iteration	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Iteration 3	⋆	⋆	⋆	⋆	⋆	✓	⋆	⋆	⋆	✓	∼	✓
Iteration 4	✓	✓	∼	⋆	∼	✓	∼	∼	∼	✓	∼	✓
Iteration 5	✓	✓	∼	∼	∼	✓	⋆	⋆	⋆	✓	∼	✓
Iteration 6	✓	✓	∼	∼	∼	✓	✓	✓	✓	✓	∼	✓

In terms of the model described in Chapter 4, Iterations 3 onward incorporated all aspects. Each iteration provided greater insights into how to successfully deliver units in this approach, and the related guidelines were developed from the changes made from Iteration 3 to Iteration 9.

Action Plan Assessment criteria in each iteration adopted the changes from prior iterations, and made improvements in wording to better capture staff intentions and expectations for each grade criteria. An example of the overall assessment criteria is shown in Figure 7.7, and details of the assessment criteria related to an individual intended learning outcome is shown in Figure 7.8.

Specific changes included:

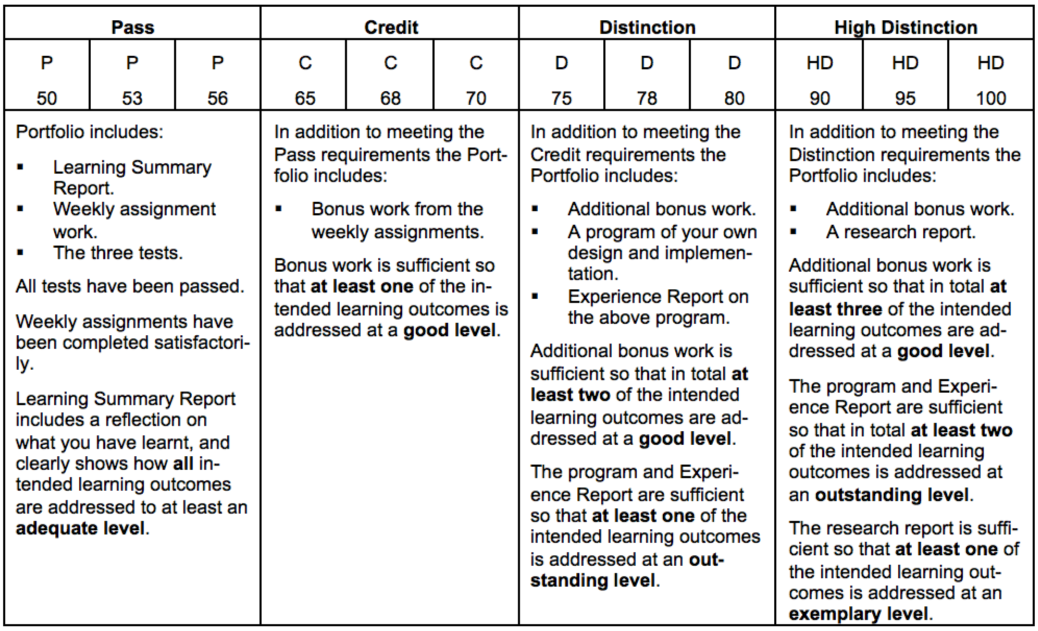
1. Iteration 3: Changes in this iteration aimed to address the poor outcomes from Iteration 2 by providing students with additional guidance, and communicating our high expectation of their outcomes (P5).
   - The flipped classroom idea was dropped, with videos now being used to support classroom activity. This aimed to use a more moderate form of constructive learning theories inline with the ideas communicated in Principle 1.
   - Adjusted the main category descriptors to: Adequate, Good, Outstanding, and Exemplary. Students had interpreted "good" to a much weaker standard than staff, shifting this down a category helped better indicate the desired standards.
   - A specific reflective report was added as required piece in student portfolios. The report included a self assessment, in which the student aligned the pieces they submitted to the intended learning outcomes, and general reflections (Principle 9). This aimed to help students identify how their work aligned with learning outcomes, while also assisting staff in assessing student portfolios.
2. Iteration 4: This iteration aimed to reduce the amount of work students needed to do in order to get higher grades, thereby enabling students to focus on building depth of knowledge. (P4)
   - Reduced the number of items expected for each grade: Credit required one Good, Distinction one Good another Outstanding, and High Distinction required one Good another Outstanding and a further one at Exemplary.
3. Iteration 5: The success of iteration 4 meant that iteration 5 made only slight adjustments aimed at improving productivity.
   - Pass and Credit students were no longer interviewed. Interviews had been used to assess all students up to this point, but the use of common tasks by all Pass and Credit students meant that staff felt the model could be adjusted to use hurdle tests and no longer require an interview for students aiming for Pass and Credit. (P8)
   - Tests became a hurdle requirement, and had to be completed to a satisfactory standard for students to be eligible to pass the unit. Students who did not pass the test first time, could sit a second test at a later date. (P3)
4. Iteration 6: Staff reflections from Iteration 5 indicated the Distinction and High

Distinction grade criteria could be better organised as most students favoured only one or two of the intended learning outcomes. It was decided to rethink the criteria to better define what each grade category meant.

- Short reports that discussed aspects related to each intended learning outcome were required to meet the *Good* standard.
- To meet the *Outstanding* standard for an intended learning outcome, students were required to develop a program of their own design and relate this to intended learning outcomes.
- Meeting the *Exemplary* standard required a research report related to the intended learning outcome.

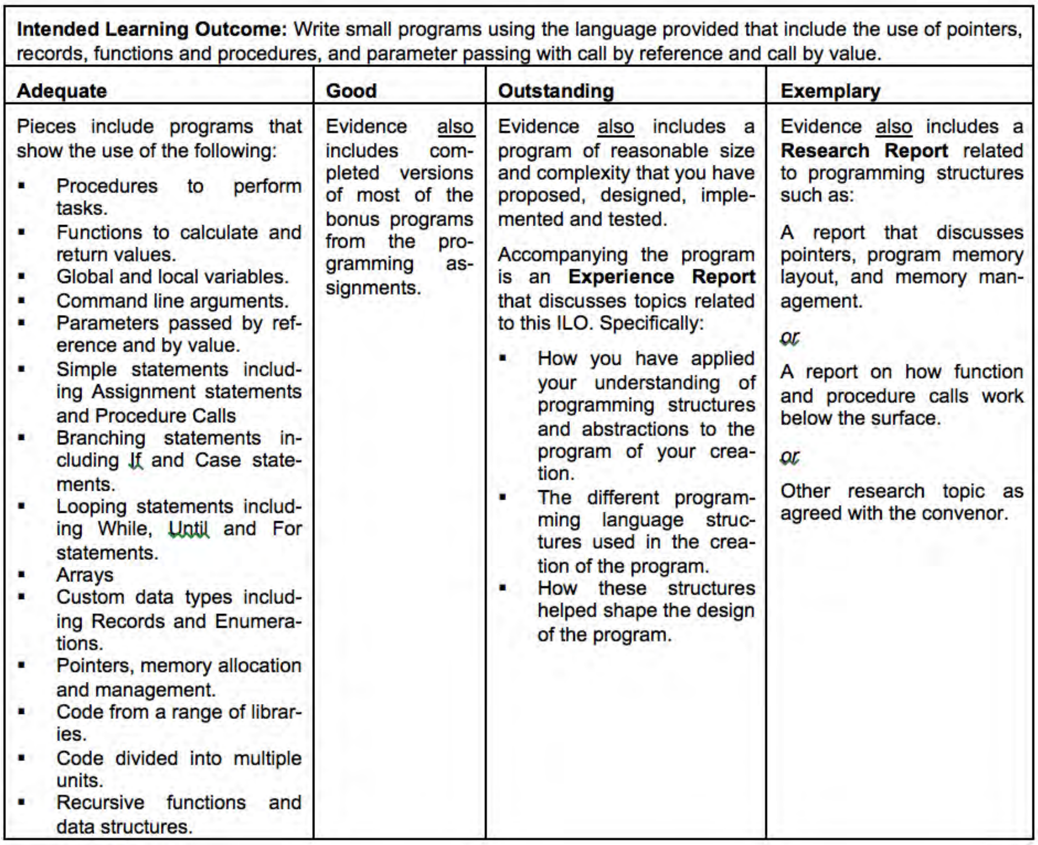
At the same time, the resources used to support the teaching of these units were also developed. Resources for Introductory Programming (A) were developed during its delivery in Iterations 3, 5 and 6, as outlined in the following list.

1. Iteration 3:
   - A first version of the "Programming Arcana" book was developed. This combined the syntax diagrams from the Pascal Language Reference manual (Van Canneyt 2013) with the notes that had previously been embedded within the lecture slide handouts.
   - Students continued to be introduced to SwinGame later in the semester, with a number of students choosing to develop custom SwinGame projects for higher grade outcomes.
2. Iteration 5:
   - Optional tasks were added to week 1 that encouraged students to start using SwinGame, and to go beyond the "basics."
   - The Learn Programming with SwinGame video podcasts were created and delivered to students during the delivery of the unit.
   - A template was provided for the Reflective Report (which later became the Learning Summary Report), see Figure 7.9. The template provided a coverage matrix for students to indicate how well they believed their portfolios had met each of the intended learning outcomes. This was followed by sections for each intended learning outcome where students documented how the specific pieces they had included demonstrated they had achieve the outcome to the level indicated in their coverage matrix.
3. Iteration 6:
   - Templates were provided for each of the short reports that were required to meet the Good standard for each intended learning outcome.
   - The Learning Summary Report template was expanded to include a specific section for students to reflect upon their learning.



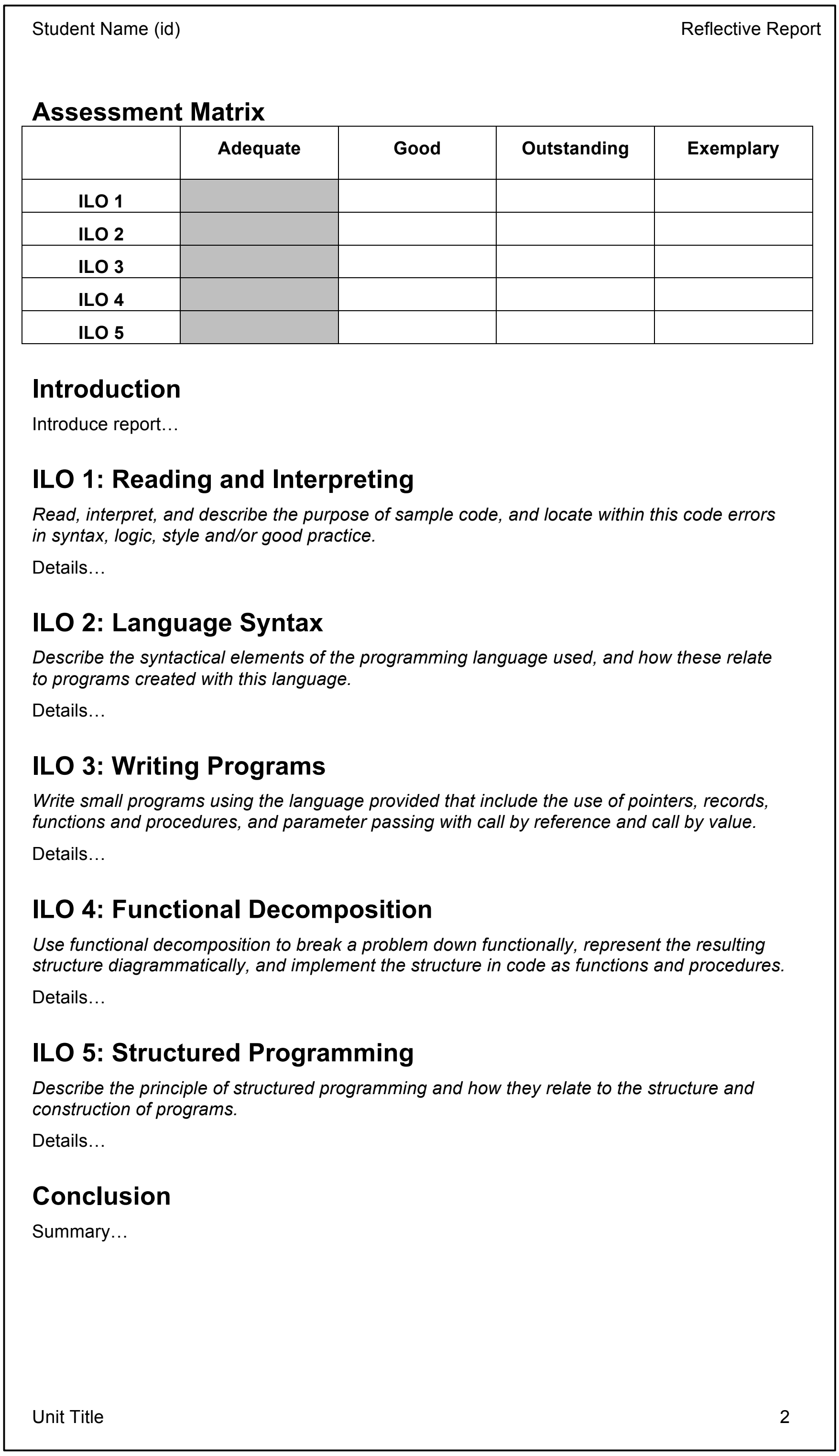
Pass			Credit			Distinction			High Distinction		
P	P	P	C	C	C	D	D	D	HD	HD	HD
50	53	56	65	68	70	75	78	80	90	95	100
Portfolio includes: ▪ Learning Summary Report. ▪ Weekly assignment work. ▪ The three tests. All tests have been passed. Weekly assignments have been completed satisfactorily. Learning Summary Report includes a reflection on what you have learnt, and clearly shows how **all** intended learning outcomes are addressed to at least an **adequate level**.			In addition to meeting the Pass requirements the Portfolio includes: ▪ Bonus work from the weekly assignments. Bonus work is sufficient so that **at least one** of the intended learning outcomes is addressed at a **good level**.			In addition to meeting the Credit requirements the Portfolio includes: ▪ Additional bonus work. ▪ A program of your own design and implementation. ▪ Experience Report on the above program. Additional bonus work is sufficient so that in total **at least two** of the intended learning outcomes are addressed at a **good level**. The program and Experience Report are sufficient so that **at least one** of the intended learning outcomes is addressed at an **outstanding level**.			In addition to meeting the Distinction requirements the Portfolio includes: ▪ Additional bonus work. ▪ A research report. Additional bonus work is sufficient so that in total **at least three** of the intended learning outcomes are addressed at a **good level**. The program and Experience Report are sufficient so that in total **at least two** of the intended learning outcomes is addressed at an **outstanding level**. The research report is sufficient so that **at least one** of the intended learning outcomes is addressed at an **exemplary level**.		

Figure 7.7: Overview assessment criteria provided to students in the unit outline of Introductory Programming (A) in Iteration 6.



Intended Learning Outcome: Write small programs using the language provided that include the use of pointers, records, functions and procedures, and parameter passing with call by reference and call by value.			
Adequate	**Good**	**Outstanding**	**Exemplary**
Pieces include programs that show the use of the following: ▪ Procedures to perform tasks. ▪ Functions to calculate and return values. ▪ Global and local variables. ▪ Command line arguments. ▪ Parameters passed by reference and by value. ▪ Simple statements including Assignment statements and Procedure Calls ▪ Branching statements including If and Case statements. ▪ Looping statements including While, Until and For statements. ▪ Arrays ▪ Custom data types including Records and Enumerations. ▪ Pointers, memory allocation and management. ▪ Code from a range of libraries. ▪ Code divided into multiple units. ▪ Recursive functions and data structures.	Evidence also includes completed versions of most of the bonus programs from the programming assignments.	Evidence also includes a program of reasonable size and complexity that you have proposed, designed, implemented and tested. Accompanying the program is an **Experience Report** that discusses topics related to this ILO. Specifically: ▪ How you have applied your understanding of programming structures and abstractions to the program of your creation. ▪ The different programming language structures used in the creation of the program. ▪ How these structures helped shape the design of the program.	Evidence also includes a **Research Report** related to programming structures such as: A report that discusses pointers, program memory layout, and memory management. *or* A report on how function and procedure calls work below the surface. *or* Other research topic as agreed with the convenor.

Figure 7.8: Example assessment criteria related to a single intended learning outcome



Student Name (id) Reflective Report

Assessment Matrix

	Adequate	Good	Outstanding	Exemplary
ILO 1	▨			
ILO 2	▨			
ILO 3	▨			
ILO 4	▨			
ILO 5	▨			

Introduction

Introduce report…

ILO 1: Reading and Interpreting

Read, interpret, and describe the purpose of sample code, and locate within this code errors in syntax, logic, style and/or good practice.

Details…

ILO 2: Language Syntax

Describe the syntactical elements of the programming language used, and how these relate to programs created with this language.

Details…

ILO 3: Writing Programs

Write small programs using the language provided that include the use of pointers, records, functions and procedures, and parameter passing with call by reference and call by value.

Details…

ILO 4: Functional Decomposition

Use functional decomposition to break a problem down functionally, represent the resulting structure diagrammatically, and implement the structure in code as functions and procedures.

Details…

ILO 5: Structured Programming

Describe the principle of structured programming and how they relate to the structure and construction of programs.

Details…

Conclusion

Summary…

Unit Title 2

Figure 7.9: Main contents page from the template provided in Iteration 5

Resources for Object Oriented Programming (A) were developed during its delivery in Iteration 4:

- Additional video podcasts were created to add support for the newly introduced Objective C programming language.
- Rather than develop a custom text, a range of textbooks and online resources were made available to students. This enabled the support of a range of languages, without the student overhead of developing additional resources.
- Previous exercises distributed to students with the lecture notes were moved online.

Data Figure 7.10 shows the grade distributions for Introductory Programming (A). The pass rate improved over these iterations: from 69% in Iteration 1, to 72%, 77%, then 80% in Iteration 6. At the same time the percentage of students receiving Distinction and High Distinction grades decreased from 57% in Iteration 1, to 42% then 37% and 23% by Iteration 6. These results relate to the changes for the various iterations, with additional guidance and experience helping a greater number of students achieve passing grades, while tightening of criteria for higher grades required students to demonstrate progressively deeper learning over these iterations.

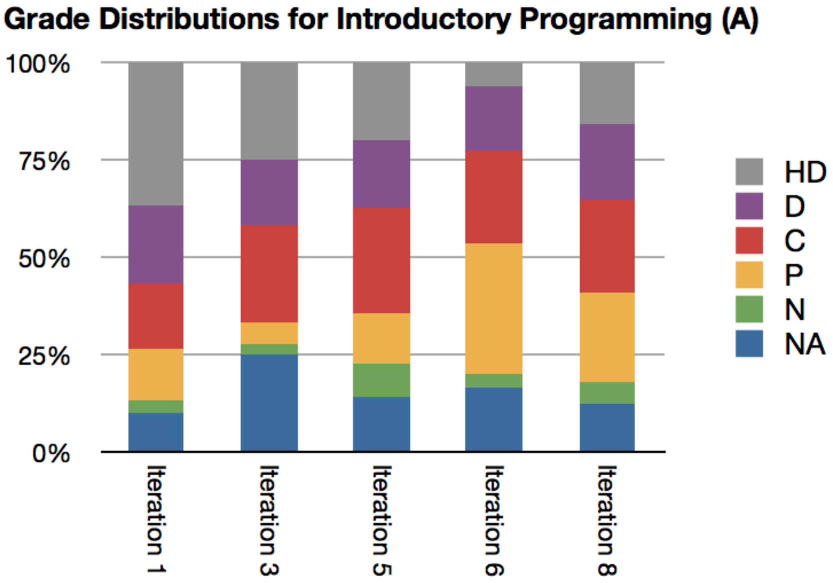


Figure 7.10: Result distributions for Introductory Programming (A) from Iterations 1, 3, 5, 6 and 8.

Figure 7.11 shows the grade distributions for Object Oriented Programming (A). In Iteration 2 the pass rate was 69%; this improved in Iteration 4 to 74%. The percentage of students achieving Distinction and High Distinction dropped over this time from

29% to 26% in Iteration 4. As with the introductory programming unit, these results help demonstrate improvements in both guidance and expectations.

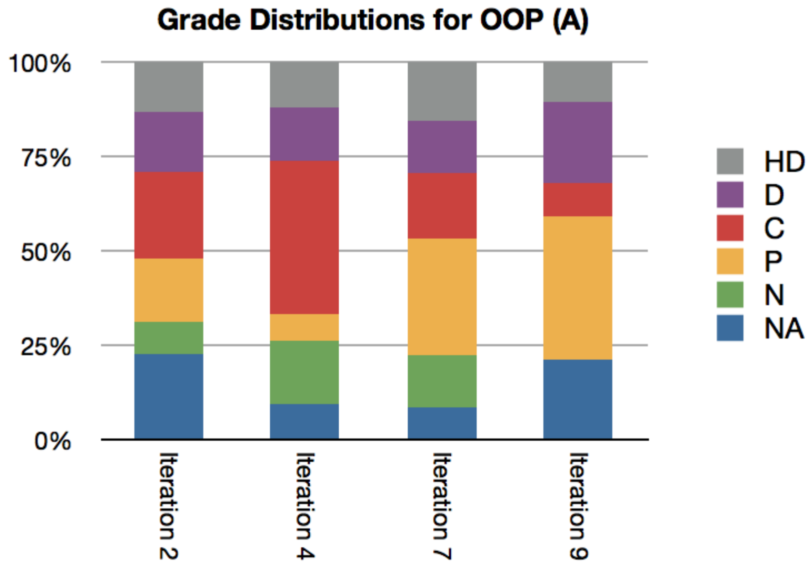


Figure 7.11: Result distributions for Object Oriented Programming (A) from Iterations 2, 4, 7 and 9.

Reflections and Analysis Staff reflections from these teaching periods indicated the improvements to assessment criteria had helped reduce the time needed to perform the portfolio assessment, and this improved with each iteration. In terms of student learning, the lost productivity from Iteration 2 was not present in these iterations and student portfolios demonstrated continually improving outcomes. We believe this can be attributed to our developing experience with portfolio assessment, student social sharing of their experiences, the improvements in the clarity of the assessment criteria, and the availability of prior portfolios as examples of what was required.

Having addressed the issues encountered in Iterations 1 and 2, staff reflections over these iterations started to indicate the benefits we had hoped this teaching environment may achieve. Staff felt that student outcomes better matched their expectations, and that the quality of student work improved each semester as better guidance was provided.

More interestingly, however, was a change in attitude to assessment. Traditionally, examination periods had been dreaded by staff involved in teaching these programming units. The thought of having to work through large numbers of student exams had never been a pleasant prospect. In many respects, reflections on Iterations 3 and 4 had similarly dreaded work in the examination period with significant work interviewing the Pass and Credit students, who all exhibited similar qualities in their portfolios. This changed in Iterations 5 and 6, with interviews focusing on the Distinction and High Distinction students. These students consistently impressed staff with their creativity, imagination, and clear demonstrations of programming competencies. The examination period changed to become one of the more enjoyable aspects of teaching these units.

The other major change, in terms of the learning environment, was the students' active incorporation of the feedback they received. Teaching staff felt that this changed the nature of assessment itself, making it more collaborative than confrontational. Students no longer argued about why they had *lost* marks on assignment tasks, but instead worked to *improve* their grades, and understanding, by incorporating feedback they received from teaching staff. The portfolio assessment model enabled staff to work together with students to help them improve their learning outcomes. Interesting, the number of students querying their final grades dropped significantly, with most teaching periods having no students wanting to know why they had not received a better mark for the unit.

Development of Principles The environment created in Iterations 3 to 6 had aimed to address the issues identified in Iteration 2, and then worked on improving the assessment criteria and available resources. At the end of these iterations it was felt that portfolio use was an effective means of assessing student outcomes.

By the end of Iteration 6, it was apparent that the separation of assessment criteria by individual intended learning outcome was overly confusing, and had not productive for either staff or students aiming for higher grades. The criteria to meet the *Outstanding* standard had required students to create a program of their own design. To achieve this, students had to apply their understanding of the concepts covered by the unit, but the assessment criteria had then required separate documentation for each intended learning outcome. This was time consuming for the students to prepare and for staff to assess. At the same time the separation of these reports made it difficult for students to relate to "real world" use of these concepts.

This experience influenced a number of the principles stated in Chapter 3.

- The model had demonstrated an effective combination of constructive learning theories (P1) and aligned curriculum (P2).
- Clearly communicating high expectations (P5) had helped with the Theory Y environment (P7) and the focus on formative feedback (P3).

- The use of a reflective component in the portfolios, which relates to Principle 9, had been positive, and helped students to reflect upon what they had learnt.
- Improvements in both how we taught and what we taught was greatly aided by reflective practice (P9) and the focus on creating a productive learning environment (P8).
- The separation of the assessment criteria by intended learning outcome had negatively impacted on students ability to demonstrate depth of knowledge, working against Principle 4.

7.2.5 Latest Iterations

Iterations 3 to 6 had started to demonstrate the benefits of portfolio-based assessment. Over these iterations the units had been delivered to students enrolled in a degree with a focus on computer science. At the start of Iteration 7 the university was looking to consolidate programming units, and it was decided to trial the portfolio assessment approach with engineering students, in Introductory Programming (B). As such, these iterations aimed to continue to develop the model while also testing it with a wider range of students.

1. Iteration 7: Introductory Programming (B) was taught using the C programming language.
2. Iteration 8: The two introductory programming units were combined into a single unit. This used two programming languages, starting with Pascal and then moving to C later in the semester.
3. Iteration 9: Combined together the two object oriented programming units. Prior to this iteration Object Oriented Programming (B) had been taught using a textbook style approach, focusing on language syntax, and was assessed using assignments and a final exam.

Iterations 7, 8, and 9

Focus The focus for iterations 7 to 9 was on expressing assessment criteria that required a consolidation of knowledge across the unit's intended learning outcome. In prior iterations, the separation of assessment criteria by individual intended learning outcomes, as illustrated in Figure 7.8, had been in conflict with the desire for students to demonstrate an integrated understanding of the concepts. The solution came in the realisation that higher grades could require students to apply the concepts related to the unit's intended learning outcomes in the development of a project of the students own design. The clarity of this helped in communicating expectations, and greatly simplified the assessment criteria.

The second focus of these iterations was the incorporation of other units into this approach: Introductory Programming (B) and Object Oriented Programming (B). This increased the number of students to which this approach was delivered, and broadened the cohort to include students not necessarily interested in software development. This required a secondary focus on addressing issues of scale, and potentially additional issues related to plagiarism.

In relation to the principles stated in Chapter 3, Table 7.7 illustrates the principles on focus for each iteration. Limitations identified in Iteration 6 resulted in changes to the expectation (P5) of students, and the focus (P4) of the formative feedback process (P3). At the same time, the larger class numbers renewed the need for productive learning environments (P8), and the further use of multiple languages helped guide the clear focus on concepts (P11).

Table 7.7: Principles related to Iterations 7 to 9: focus ⋆, present ✓, partially present ∼.

Iteration	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12
Iteration 7	✓	✓	⋆	⋆	⋆	✓	✓	⋆	✓	✓	⋆	✓
Iteration 8	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Iteration 9	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Action Plan For these iterations the assessment criteria was as reported in Chapter 5 and shown in Figure 5.3. This was accompanied by an explanation of what was required from the individual components: weekly tasks, tests, own program, and research report.

For the assessment criteria, the main challenge in iterations 7 to 9 had been on trying to find clear requirements for the Credit grade. This needed students to demonstrate good coverage of the intended learning outcomes, while limiting the required workload. The following list shows what was used as the criteria for the Credit grade over these iterations.

- Iteration 7: Students were required to complete a piece of their own creation that demonstrated good coverage of all intended learning outcomes. The Unit Outline was not specific, but suggested that this could include reports, concept maps, glossaries, or any other piece the student wanted to create.
- Iteration 8 and 9: Weekly tasks included core tasks, and extension tasks. The core tasks included strong guidance, whereas the extension tasks required a greater

level of independence. The Credit grade required all weekly tasks to be completed, as well as a selection of the weekly extension tasks. This also kept the other piece of the students own creation as in Iteration 7. Figure 7.17 shows an example of the assessment criteria from Introductory Programming (B) in Iteration 9.

In terms of teaching and learning resources, these iterations included the redevelopment of a number of resources, and the development of the Doubtfire task tracking system.

1. Iteration 7:
   - A new version of the Programming Arcana was created. This version used SwinGame and focused on procedures first and the C programming language.
   - SwinGame was used from week 1 in core lab tasks, enabling the procedures first approach.
   - Syntax diagrams created for the Programming Arcana were used in the Lecture slides.
   - Weekly exercises were moved into the Programming Arcana to help encourage student to make greater use of the resource.
   - The Introductory Programming video podcasts were created to support the lecture material.
2. Iteration 8:
   - The new Programming Arcana was extended to include both the C and Pascal programming languages.
   - Weekly exercises were moved back into the laboratory handout for a number of reasons including practicalities related to changes, and issues with student engagement. Changing the exercises in the Programming Arcana required the entire text to be recompiled and re-distributed to students, which made it more difficult to make changes during the delivery of the unit. Students also seemed less engaged with these exercises, and appeared to view these as "textbook questions" that could be skimmed over – as they were related to the "textbook" not specifically designed for the unit.
   - Additional documentation was provided to demonstrate the research process, and how a small research project could be conducted and documented.
3. Iteration 9:
   - Weekly exercises were adjusted to include Lab exercises, Core exercises, and Extension exercises. The Lab exercises were designed to be worked through in a laboratory class with the guidance of a tutor. These exercises did not need to be included in student portfolios.

- The Doubtfire tool was developed, and Core exercises were used as the weekly tasks for the burn down charts. To be eligible for Credit students needed to have all Core exercises signed off.

Data For software development students the pass rate continued to rise through these iterations, with 82% of students passing Introductory Programming (A) and 79% passing Object Oriented Programming (A) in Iteration 9.

In Iteration 7, Introductory Programming (B) was introduced and achieved a 71% pass rate. However, student portfolios were generally considered to be weaker than for those in Introductory Programming (A) and fewer students achieved high grades. This can also be seen in Figure 7.12, which shows the results for the combined units in iterations 8 and 9.

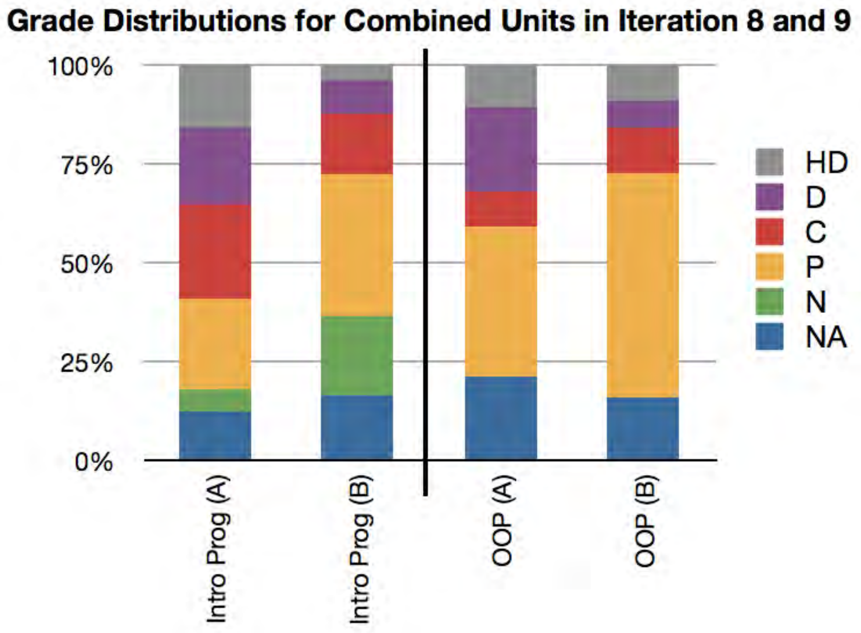


Figure 7.12: Result distributions for combined units in iterations 8 and 9

Reflections and Analysis With Introductory Programming (A) the quality of submitted portfolios continued to improve through these iterations. In Iteration 8 extra guidance was provided on how to conduct and document a small research project, and this seems to have been beneficial with an increased number of High Distinction portfolios.

With other student cohorts, the results seem less positive. Staff indicated a difficulty in engaging students not enrolled in a degree that focused on software development.

This was most pronounced in the combined introductory programming unit, as can be seen in Figure 7.12. Given that both groups of students had been delivered the same material, in the same classes, it had been assumed that the distribution of results should be similar.

The clarity of the assessment criteria and the clear delineation of what is required for the Pass, Distinction and High Distinction grades further reduced the time needed for staff to assess the portfolios. This meant that the majority of the assessment could be performed quickly at the end of the teaching period, allowing teaching staff to concentrate on assessing the Distinction and High Distinction portfolios. Interviewing these students remained a pleasurable experience, as portfolios demonstrated what the most successful students had been able to achieve.

Over these iterations, the teaching and learning environment retained its positive, supportive, and student-centred focus as reported for Iterations 3 to 6. The clarity of the assessment criteria enabled even closer collaboration between students and staff as they worked *together* to ensure students succeeded at demonstrating the required understandings.

While staff felt that the portfolio assessment was very successful through these iterations, the criteria for the Credit grade remained unclear and tended to require students to complete *more work* rather than demonstrate *better understanding*.

Development of Principles These iterations followed the model described in Chapter 4, and delivered the units as outlined in Chapter 5. Implementations across these iterations embodied all of the principles from Chapter 3. The focus on creating a productive learning environment incorporating reflective practice (P8 and P9) continued to ensure that the process itself improved in each iteration.

7.2.6 Current Iteration

Focus This teaching approach is currently being used to deliver Iteration 10. In this iteration the focus is on achieving better outcomes in the combined units, and to reduce the workload required by the Credit criteria while still maintaining the a high standard.

As with iterations 7 to 9, Iteration 10 incorporates all of the principles outlined in Chapter 3. Similarly, Iteration 10 is using the model described in Chapter 4, and incorporates all activities and guidelines. The exemplar units from Chapter 5 are based the units being delivered in this iteration.

Iteration 10 differs from Iteration 9 only in its strengthening of the use of the Glossary as the criteria for Credit.

Plan One of the success stories from Iteration 9 was the use of a Glossary covering programming terminology, abstractions, and statements. This was used as a means of getting students to *describe* and *explain* principles and associated concepts, and seemed to be an effective means of both engaging the students with the material and assessing their learning outcomes. As a result, Iteration 10 will use the glossary for the Credit criteria. It is hoped this will help students engage appropriately with the teaching and learning activities.

7.2.7 Summary

This section has presented results and analysis from nine iterations of an action research project that examined the implementation of portfolio assessment. The overall focus of this work was on development, application, and ongoing evaluation of the model from Chapter 4.

Initial attempts at portfolio assessment failed to demonstrate expected outcomes, and student portfolios were generally weaker than desired. Over subsequent iterations, assessment criteria were evolved to more tightly define what was expected of each grade and students portfolios improved to meet these expectations. In the later iterations, staff were able to quickly assess portfolios, and felt that grades accurately reflected student outcomes.

This work provides additional evidence of the strength of portfolio assessments for assessing and supporting learning. In each iteration the assessment criteria helped guide students in preparing their portfolios, and as the criteria evolved the evidence in student portfolios improved. Experience delivering portfolio assessed units resulted in criteria that clearly relates to active verbs at the multi-structural and relational levels of the SOLO taxonomy, providing a strongly aligned assessment of intended learning outcomes.

Our experience highlights the importance of ensuring intended learning outcomes are expressed clearly, and capture the core concepts and principles that need to be demonstrated in student portfolios. The assessment criteria then maps the intended learning outcomes to statements of required levels of achievement. Together the intended learning outcomes and assessment criteria express what needs to be done and how well it needs to be done by the students to achieve different grades.

The model presented in Chapter 4 provided an effective means of achieving constructive alignment. For students the process provides support and encouragement through iterative formative feedback, gives them clear expectations of what they need to achieve, and results in meaningful grades. After the initial investment, the intended learning outcomes and assessment criteria provided a dual means for staff to express expectations, and the resulting environment encouraged reflective practice. During the teaching period, staff and student efforts were both directed toward the one goal: helping students achieve the intended learning outcomes to the best of their ability.

7.3 Issues Identified in Student Reflections

Student reflections provide a rich opportunity to identify issues that are relevant from the students' perspective. The investigation presented in this section analyses issues identified in student reflections from the Introductory Programming (A) unit in Iteration 6, and provides support for recommendations to help inform the development of units using this approach.

7.3.1 Method

This study examined student reflections from the Introductory Programming (A) unit from Iteration 6, the current iteration at the time of the study. It was decided to use portfolios from a single iteration for a number of reasons. Time was a primary consideration, with the thematic analysis being a time consuming process. By limiting the analysis to a single iteration this would also provide a snapshot of issues related to this particular iteration. Future studies could then repeat the analysis for future iterations, with this first study providing a point of comparison.

This section is divided into three parts to clearly describe the details of *Introductory Programming (A) in Iteration 6*, the *Student Cohort and Research Participation*, and the *Thematic Analysis of Reflections*. In the Introductory Programming Unit section we provide details of the unit that was investigated as part of this research. The Student Cohort and Research Participation section details the student body undertaking this unit and how they were recruited to be part of this research. Finally the Thematic Analysis of Reflections section outlines the process followed to extract and analyse the data from the student portfolios.

Iteration 6: Introductory Programming (A)

In Iteration 6, Introductory Programming (A) had implemented the large majority of the principles and processed discussed in Chapter 3 and Chapter 4. The teaching and learning activities differed slightly from the introductory programming unit described in Chapter 5, though the emphasis on concepts over syntax was present. The topics for the twelve lectures for this teaching period are shown in the following list as they relate to the topics mentioned in student reflections.

1. Programs, Procedure, Compiling and Syntax

2. User Input and Working with Data
3. Functions, Procedures, and Parameters
4. Branches and Loops
5. Custom Data Types
6. Functional Decomposition
7. Case Study
8. Pointers and Dynamic Memory Management
9. Structured Programming
10. Recursion and Backtracking
11. Portfolio Preparation
12. Review and Future Studies

The unit's delivery included an early introduction topic of "understanding syntax", where students were taught how to read programming language syntax using the visual "railroad" diagram syntax notation. This allowed later lecture topics to focus on concepts, with syntax being offloaded to programming demonstrations and supplied notes, which included railroad diagrams and small code examples for each programming statement.

As described in Chapter 5, the allocated classes were designed with the goal of actively engaging students. Lectures typically included a review of previous topics, a short presentation using "Beyond Bullet Points" style lecture slides, an interactive programming demonstration, and group activities. Laboratory sessions involved code reading activities, guided coding activities, and practical hands-on exercises.

The approach to assessment included weekly submissions and formative feedback to help students develop their understanding, three hurdle tests to ensure basic competence, and portfolio assessment for final grades.

Students were asked to reflect on their learning in a Learning Summary Report, and a template document was provided to assist students in preparing their comments. The template prompted students to describe the pieces they had included in their portfolio, to describe how these pieces related to the unit's intended learning outcomes, and then to reflect on what they had learnt from the unit.

To help students in writing their reflections, the following instructions were provided in the template.

> Think about what you have learnt in this unit, and reflect on what you think were key learning points or incidents. Answer questions such as: What did you learn? What do you think was important? What did you find interesting? What have

> you learnt that will be valuable for you in the future? Which activities helped you most? Has this changed the way you think about software development? Did you learn what you wanted/expected to learn? Did you make effective use of your time? How could you improve your approach to learning in the future? Etc.

Note that there were no prompts for students to include details on issues they had encountered, meaning that any issues expressed should have been significant to the learning experience of the student.

Student Cohort and Research Participation

In Iteration 6 the Introductory Programming (A) unit was undertaken by 84 students, 70 of whom submitted a portfolio for assessment. Participation in the research was voluntary, with informed consent being sought in lecture 11 as outlined in Section 7.1.3.

Table 7.8 shows the number of portfolios made available to this research, the number that included comments related to the theme of "issues" and the distribution of grades. The grade distribution is also shown in Figure 7.13, and will be discussed in Section 7.3.3.

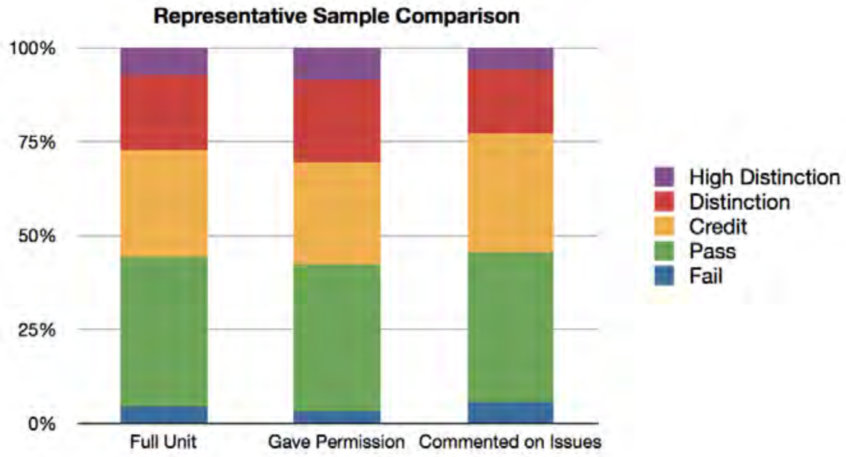


Figure 7.13: Distribution of grades for the full unit, for those students who agreed to participate in the research, and for those who commented on issues.

Thematic Analysis of Reflections

The thematic analysis of student reflections followed the process outlined in Section 7.1.2. Initial themes were generated by examining the reflective component of each portfolio and looking for all explicit mention of issues the student faced. Each new issue identified was matched to a theme and recorded in a spreadsheet. To ensure that all issues were reported in the results, the process of grouping themes did not remove or ignore any issues raised. All issues that could not be grouped into an existing theme were collected together as a miscellaneous "other" theme. The Results section outlines the different themes identified, and how these themes relate to the comments raised by students in their reflections.

7.3.2 Results

A number of themes emerged from the analysis, and can be broadly classified as either general *learning issues* or *programming related issues*. (See Table 7.8 and Figure 7.14.) Each of these categories is presented in Table 7.9 along with the number of students who raised these issues, broken down by grade. The following sections describe the individual themes in more detail.

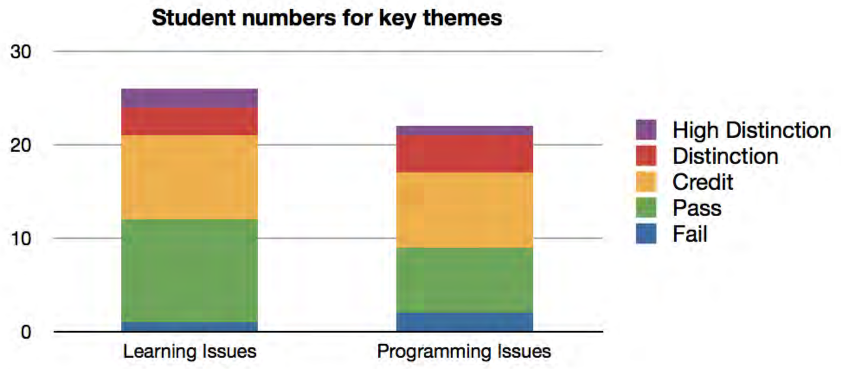


Figure 7.14: Number of students mentioning learning issues and programming issues. See Table 7.8.

General Learning Issues

The *general learning issues* capture all of the comments made by students that do not relate directly to a given programming topic or technical aspect of the unit, but instead relate to the students' learning experience in general. In this category the themes that appeared include *time management*, *getting started* with the unit, and *learning through mistakes*. The issue counts and grade distribution of these are included in Table 7.9, and can also be seen in Figure 7.15.

Time management issues identified in the students' reflections included comments about

Table 7.8: Portfolios submitted, issue comments and grade distribution.

	Total	HD	D	C	P	F
Submitted Portfolio	70	5	14	20	28	3
Agreed to participate	59	5	13	16	23	2
Commented on Issues	35	2	6	11	14	2
- Learning Issues	26	2	3	9	12	1
- Programming Issues	22	1	4	8	7	2

Table 7.9: Issue count results for grade and theme. Values of interest are indicated using bold format.

Theme	Description	Total	HD	D	C	P	F
Learning Issues	Issues related to learning in general.						
- Time Issues	Time constraints, or issues with time management.	14	1	0	**5**	**8**	0
- Getting Started	Comments relating to initial weeks, or tacking early hurdles.	8	0	1	2	**5**	0
- Learn through mistakes	Specifically commented on having issues and learning from these.	7	1	2	2	2	0
- Other	Other learning related concepts not allocated to other themes.	5	0	0	2	2	1
Totals			2	3	11	17	1
Programming Issues	Issues related to programming topics, or technical areas.						
- Pointers	Use of pointers and dynamic memory allocation functions.	11	1	2	**4**	3	1
- Parameters	Mentions parameters, or parameter passing	8	0	1	**4**	3	0
- Program Design	Algorithm and program structure design	7	0	1	1	3	2
- Other (Syntax)	Other issues, but related to the language syntax or concepts.	7	0	0	2	3	2
- Other (General)	Other programming issues not allocated to other themes.	5	0	0	2	2	1
- Recursion	Declaration and use of recursive functions or data structures.	4	0	1	2	1	0
Totals			1	4	14	11	3

aspects such as “staying on task”, wishing they had “asked for help earlier”, or the general need to improve their time management to enable them to achieve higher grades. It can be seen that the majority of these concerns were raised by students who obtained either a Pass or Credit grade. These comments are further supported by observations from teaching staff, who noted concerns about students not working consistently through the semester and not seeking help in a timely manner.

The next largest general learning issue was *getting started* with programming. These comments specifically indicated issues related to the initial hurdle of getting started with the unit. One student noted this as their first experience using a computer, while others commented on the difficulty of the first few weeks’ lab exercises. Again, these findings are supported by observations from the teaching staff who noted that a number of students withdraw from the unit before census date,[2] and there was a general drop in enrolment numbers around this time. This may indicate that a larger number of students faced these issues but did not continue with the unit, though further work would be needed to verify this.

The last main issue in this section related to students reflecting on the mistakes or struggles that provided them with an opportunity to learn something important, referred to as *learning through mistakes*. For example, one student’s reflection noted that:

> “...I suddenly gained insight [into the code] I had been struggling with ...”

The reflection continued on to comment that having overcome these issues they gained a clearer understanding of the concepts taught up to that point, and that subsequent programs were easier to understand.

The following *other* issues were raised by individual students:

- transitioning to university life and study,
- finding information in the online learning management system,
- seeking help in general,
- keeping up with the pace of the unit, noted as “challenging but good”, and
- adjusting to portfolio assessment.

[2] This is the date when the university records enrolment numbers, typically a few weeks after the start of the semester to allow for changes of enrolment.

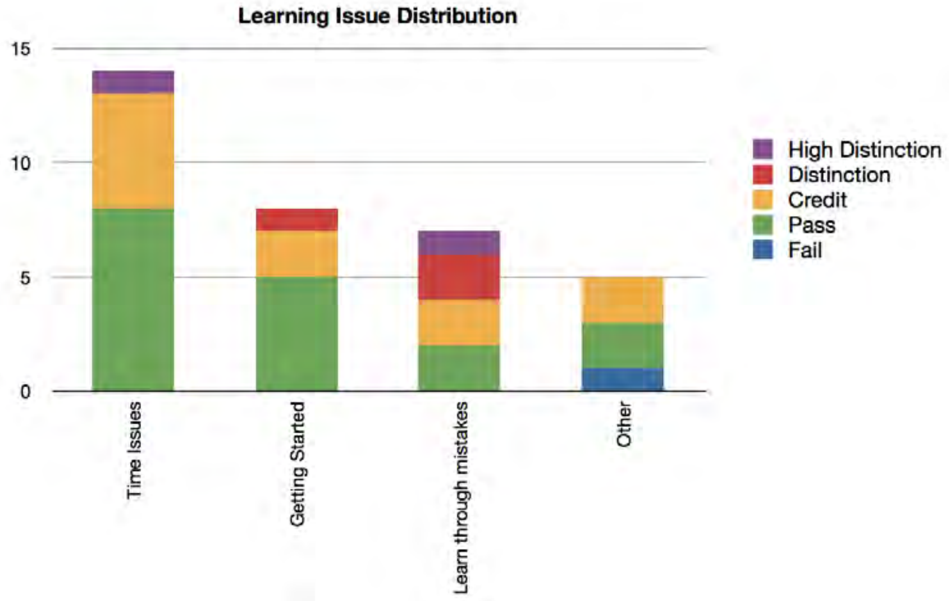


Figure 7.15: Number of students mentioning issues related to learning. See Table 7.9.

Programming Issues

As already mentioned, fewer students commented on programming or technical issues in their reflections than the more general learning issues. The programming sub-themes matched specific topics covered in the unit, including *pointers*, *parameters*, *program design*, and *recursion*. In this theme the *other* sub-theme featured more prominently, with a larger range of issues being located in the reflections of only one or two students. The data for these themes is listed in Table 7.9 and shown in Figure 7.16.

Amongst the identified programming issues, *pointers* featured most prominently. Comments typically referred to having issues with "pointers", with the more detailed comments discussing issues with knowing when to dereference pointers and being unsure of when to use pointers. This is further supported by notes from teaching staff indicating that pointers tended to be problematic even for students who demonstrated strong programming skills up to that point in the material.

Parameters were also mentioned by a number of students as being a topic that was particularly challenging. This included comments relating to tracing parameter values through a number of function or procedure calls, and issues of a single value having different names across different routines. From these comments there is a direct connection from parameter issues to a student's understanding of program structure, or more importantly execution flow.

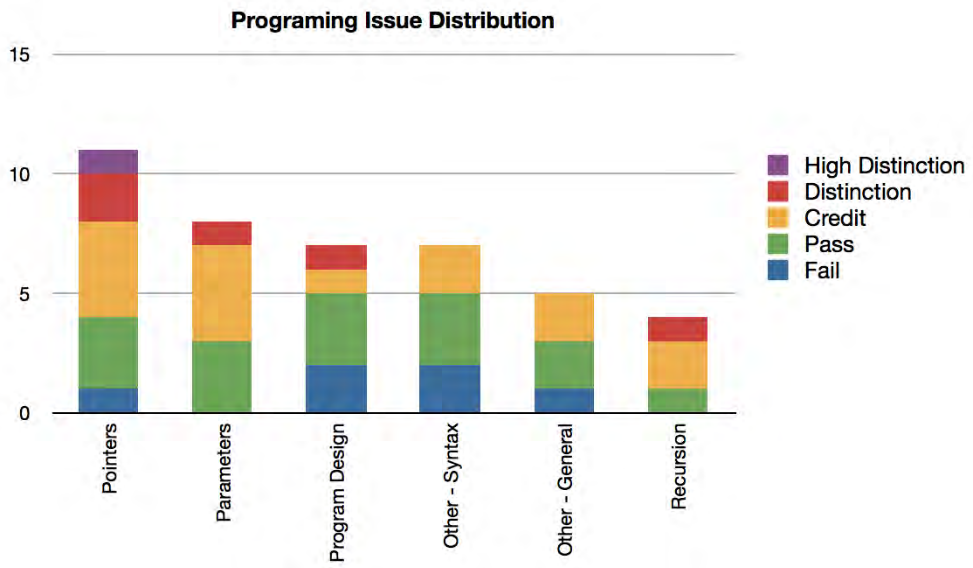


Figure 7.16: Number of students mentioning issues related to programming. See Table 7.9.

Issues relating to *Program Design* were also raised in the portfolio reflections. These comments related to aspects such as using functional decomposition, planning program structure, and designing algorithms.

The **other** issues for the programming category captured issues identified by one or two students. These were classified as relating either to **syntax and concepts** or **general programming** issues.

- Syntax issues included:
  - iteration and working with loops,
  - using arrays (two comments),
  - creating composite data types using records,
  - functions in general,
  - dealing with syntax errors, and
  - using units to divide programs into multiple files.
- General programming issues included:
  - "programming in general",
  - "following program code" in code reading exercises,
  - difficulties finding and using resources from the SwinGame library, and
  - the maths needed to achieve programming tasks.

There were also a number of reflections that raised the topic of **recursion**; these mentioned issues with both recursive functions and data structures.

7.3.3 Discussion

Investigation Focus and Sample Quality

Comments provided by students, when reflecting on their learning during any unit, can be valuable and interesting in many ways, especially with respect to the evaluation of a particular approach to teaching. Our investigation focused specifically on the theme of issues mentioned or identified by students in their reflective reports. Results of the thematic analysis, presented in Section 7.3.2, identified clear key themes. Additionally, several individual comments were selected.

The analysis considered a sample of reflective reports presented in a single semester unit. Of the 70 students in the class, almost 85% were willing to participate. Within the participant group, 35 students wrote one or more comments that matched the target theme. Table 7.8 and Figure 7.13 show that the relative distribution of grades in the contributing group matches closely to both the participant group and the entire results for the unit. This strongly supports that the results are a representative sample of the unit, at least with respect to grade distribution.

General Learning Versus Programming Issues

Beginning with the two key themes of general learning issues and programming issues (Table 7.8 and Figure 7.14) it can been seen that the distribution of student grades is very similar, with a slightly stronger representation of Pass students in the learning issues theme.

Overall, more students commented on learning in general. This is of particular interest given the relative emphasis of the course material, which focuses on teaching programming concepts over syntax details. Despite the relatively small time spent on syntax, students did not mention having related issues.

A closer examination of the issues related to programming strengthens this analysis further. Most student comments on programming issues (Table 7.9) concerned applying programming concepts, rather than issues of understanding syntax. Also, these comments were about when and how to use the related programming concepts rather than specifically how to apply the syntax of the language used.

Comparison of the grade distributions within the learning issues (Figure 7.15) and programming issues (Figure 7.16) suggests potentially interesting differences, such as issues specific to grade groups, and other issues across all grades. The sample size of this investigation limits any significant insight although some points are listed in later discussion.

Learning Issues

Time Management issues were identified by the largest number of students (Table 7.9). The grade distribution is skewed towards student's who achieved Pass and Credit results (bold values), suggesting that students who do achieve Distinction or High Distinction results managed time better, and that the unit structure requires good time management to achieve these outcomes.

Developing a portfolio that demonstrates the ability to apply concepts taught requires time: time to practice using the concepts, and time to demonstrate their use competently. For students to achieve Distinction and High Distinction grades, they needed to be able to organise their time effectively.

With more traditional forms of assessment, marks can be used as incentives. Using assessment due dates during the delivery of the unit has the effect of turning marks into time distributed weighted incentives. Marks no longer represent the importance of the learning outcome, but match allocation of incentive. Consider, for example, the allocation of marks for lab attendance. These marks do not help measure the students' learning outcomes, but are purely there to incentivise lab attendance. Similarly, assignments due within the unit delivery period assess the speed of acquiring and demonstrating the required knowledge.

With portfolio assessment the summative assessment is delayed until after unit delivery. This has the benefit of providing a more direct assessment of learning outcomes, but has a cost related to loss of incentives during delivery. While this is positive from a learning perspective, it can easily lead to students delaying their work on portfolio assessed units in order to address what they perceive as more time critical assignments in other units. Given the number of comments related to this issue, it appears to be easy for students to then lose sight of how they are falling behind in a unit with a relatively flexible portfolio assessment.

The prevalence of issue provided the incentive to implement the Doubtfire tool described in Section 6.1. The use of this tool is discussed in 7.4.

Getting Started is another issue facing many students (Table 7.9 and Figure 7.15). In the first few weeks of the semester, students face practical and conceptual issues. Practical issues include installing compiler software and text editors, learning to use command line tools, and issues with general computer use. At the same time students need to build a viable conceptual model of computing (Hoc & Nguyen-Xuan 1990), and relate this to the programs they are creating.

Early on students may also face challenges transitioning to university study and university life in general. In the first few weeks students are also more likely to have issues with syntax, and dealing with syntax errors. Together these challenges can present a significant hurdle for students.

These issues could be addressed in a number of ways. Shifting toward a single, easy to install, IDE could remove some issues related to the use of the command line compiler. However, IDEs are typically designed for professional audiences and would add overhead related to use of a more complex programming environment. These environments also obscure the underlying tools which and does not assist students in building their conceptual model of computing. The teaching staff also considered that students undertaking this unit do need to learn how to use a command line, and this early introduction meant that later units could expect at least some student familiarity with command line tools.

Learning Through Mistakes The students' active role in building their own conceptual model of a topic plays a significant role in constructive learning theories (Glasersfeld 1989). Effective teaching then becomes the ability to place students in situations where errors in their understanding can be challenged to help the students build viable conceptual models.

With this in mind it is interesting to note, as shown in Table 7.9, the number of students who commented on gaining significant understanding through making mistakes. In line with constructive thinking, these students encountered situations in which their conceptual model was inappropriate, and in addressing the associated problems they were able to gain a better, more robust, conceptual model.

Comments about learning through mistakes were distributed across all grades, from Pass through to High Distinction (Figure 7.15). This suggests that mistake-based learning experiences are beneficial to a wide range of students, albeit with some students gaining a better understanding than others through the process.

Other Learning Issues From the other issues students noted, many can be attributed to transitioning to university education. For example, learning to locate and use learning resources and to seek help, are all issues that students must come to deal with when shifting to university education.

It is interesting to note that one student did raise a complaint about portfolio assessment, indicating that it would be easier to sit an exam. While this is only a single student, it does highlight that the purpose of the ongoing assessment may not be realised by all. Tang et al. (1999) indicated that students tend to apply narrower learning strategies for examinations, focusing on memorising material covered in lectures. In contrast, Tang et al. (1999) also found that with portfolio assessment students adopted a wider perspective, making use of higher cognitive activities such as application, relation, and reflection. Students are likely to find these higher cognitive activities more challenging, and therefore those who wish to apply surface learning approaches are likely to prefer other assessment strategies.

Programming Issues

Pointers and Recursion Our results support those from Lahtinen et al. (2005) in indicating that students find learning pointers challenging. Issues related to using pointers and memory management featured across a range of grade results (Table 7.9 and Figure 7.16), indicating that this concept was challenging even for those students who managed to achieve good results in the unit.

Pointers require a good conceptual understanding of computing, and the mental ability to debug logical errors. Issues with pointers can often result in abrupt program termination, which can be very confronting for beginner programmers. Locating the cause of these errors is an additional challenge, that requires a students to build a mental model of what is happening within the programs they have written.

Issues with recursion were raised by fewer students than other issues, which is in contrast to the study by Lahtinen et al. (2005). This may be explained by the short time students had with recursion in this study. A deep exploration of recursion was not required for students to pass the unit. It is likely, therefore, that many students did not have sufficient time to explore more involved applications of recursion.

In addition to being complex, pointers and recursion both occur relatively late in the curriculum. With pointers, students had little time to develop the skills necessary to handle associated issues, whereas with recursion the short time meant students had little opportunity to develop programs of sufficient complexity to encounter issues. In either case, at the time of writing their reflections, issues with later lecture topics are perhaps more likely to be in focus.

Parameters require students to understand local scoping of variables, procedure and function calls, and methods for sharing these values between functions and procedures. This appears to be another point at which students need to expand their internal mental model of computing (Hoc & Nguyen-Xuan 1990).

While parameter concepts can take time to understand, issues are likely to be constructive in nature. When the logic for a program is contained within a single procedure, students can develop a simplistic model of what is occurring when other functions or procedures are called. When students need to design their own functions and procedures that require parameters, they are presented with situations that challenge their simplistic model. This suggests that parameters provide a significant learning opportunity from a constructive perspective.

The two different parameter passing methods are taught in the unit, with pass by reference being used to create procedures to swap parameter values, as well as allowing procedures to modify data within structures and arrays. Call by reference provides an early introduction to references.

Program and Algorithm Design Program and algorithm design are progressively taught throughout the unit, with the main focus being in the middle of the unit's delivery in topics related to functional decomposition and structured programming. Comments by students indicated several issues on how to practically apply the concepts covered to create programs.

We initially expected a larger representation of this issue, as design tasks require a deeper *relational* understanding of the concepts being used. However, the core tasks students had to submit for a Pass grade were accompanied with detailed instructions to help ease these design issues. Extension tasks required for a Credit grade did require some design components, and less guidance was provided. Students attempting their own program, necessary for a Distinction grade, needed to perform design activities as these programs were of their own design and creation.

Other Programming Issues raised by students can be classified as individual challenges. It seems that students are likely to learn at different paces, in different ways, and find different topics challenging. Again, general comments concerned the application of programming concepts, rather than with programming language syntax itself. Each of the raised issues indicated a point at which students had an an awareness of these and an opportunity to challenge and develop their conceptual understanding of programming and their model of computing.

Recommendations

Based on the thematic results and on the experiences of staff involved in the unit delivery, there are a number of implications and recommendations that can be made. These recommendations are listed below, and their explanations follow.

- Strongly avoid mixing formative with summative assessment
- Give students time to adjust to portfolio assessment
- Focus on student "awareness"
- Use a quick formative feedback process
- Avoid the "tutor debugging" phenomena
- Use visual methods to convey progress
- Make students aware of issues they are likely to face

These recommendations have been incorporated into the model described in Chapter 4, and in the exemplar units discussed in Chapter 5. For example, the use of the sign off process for core tasks aimed to raise student awareness, and is linked to the Doubtfire tool's visual burn down charts.

Always formative, lastly summative Separating formative feedback processes from summative marking has a clear value, and this is reflected in student comments. Our observation is that using a punitive marking system creates an incentive for students to hide faults and limits in their understanding. However, students need to know what they need to learn and need to expose their misunderstandings in order to help staff guide them in developing appropriate mental models. Related to this is the time a student may spend asking about marking schemes or lost marks – time better spent on learning.

Students need time to adjust In comments to staff, students have said that it takes time to get used to a portfolio based unit even if they understand the principles. If we consider that students are somewhat conditioned to respond to summative marking and due dates as a way of allocating their attention, an interesting question emerges: how do we help students maintain an active engagement with the unit activities without marks? The Doubtfire tool was one attempt at answering this question, though this remains an ongoing challenge and research opportunity.

Focus on student awareness Primarily, student awareness is the basis for positive engagement and an aware student has the opportunity to make appropriate choices. To support this, staff need to communicate the structure, activities and expectations of a portfolio-based unit to students as effectively as possible. Unfortunately students may essentially have poor habits that can take time to adjust. It is possible to help students with issues such as time management and, hence, learning outcomes.

Although formative activities many not have due date or marks (grade penalties), staff should still express clear expectations of when work needs to be done. In some cases this leverages a students' habits to their advantage as they feel compelled to do the work. Ideally, students should give these formative tasks as high a priority as assignments with marks.

Use quick formative feedback Very quick feedback helps to create strong reinforcement to each student that the process *really is* formative and personally valuable. In a students' experience summative marking is often a delayed process. If formative feedback takes a long time it is removed from the students' current learning and challenges, and so can be confused as summative marking. Students need to be engaged with the formative nature of these assessments, making use of the feedback to help develop their understanding.

Avoid tutor debugging A possible problem with quick formative feedback, and resubmission opportunities, is that students may submit poorly prepared "drafts" and use staff simply to "fix things." This issue has been described as "tutor debugging" by some of our staff, and should be actively discouraged. One approach to this is to set minimum submission standards for work submitted for feedback.

Use visual methods to convey progress Visual charting of tasks and completed work, calendar events and strong reminders of work and time limitations help to engage students. It is also possible that a "gamefication" approach, by recognising personal or group achievements and rewarding with awards, "badges" an other game-related concepts, can create a fun and personal incentive for students. We also recognise that there are also risks with gamefication, such as trivialisation of the value of core learning activities or distorting the value of learning activities through association to a gamefication artefact.

Tell students what to expect Finally, helping students understand the issues they are likely to face should help them prepare sufficiently for the more challenging tasks. This is particularly relevant to the issues related to getting started. The challenges early on in the unit may put a number of students off, and these students are likely to lose motivation and engagement with the unit. Making them aware that these challenges are "normal", and to be expected, may assist them in getting over early hurdles.

7.3.4 Summary

In this section we have presented a thematic analysis of reflective reports presented by students as part of their assessment in the Introductory Programming (A) unit from Iteration 6. The development and delivery of the unit followed the model from Chapter 4, being an implementation of the introductory programming unit described in Chapter 5. A good representation of students distributed across all result grades agreed to participate in the study.

Thematic analysis was directed specifically at the theme of *issues* identified by students. Overall results showed that more students raised learning issues than programming related issues. Significant learning themes included time management, getting started and mistake-based learning. The most common programming issues were related to pointers and parameters, with only a small number of issues related to syntax, and both these results were expected. Issues related to program design were raised less than expected.

The discussion considered a number of interesting results, and put forward recommendations and future directions for research in this area.

7.4 Evaluating Progress using Burn Down Charts

The work presented in this section examines the rate of student progress, as recorded using the Doubtfire task tracking tool and reported in student portfolio. By examining the various rates of progress and grades associated with these portfolios, we are able to gain some insights into the teaching and learning environment, and provide recommendations to help inform the development of units using this approach.

7.4.1 Method

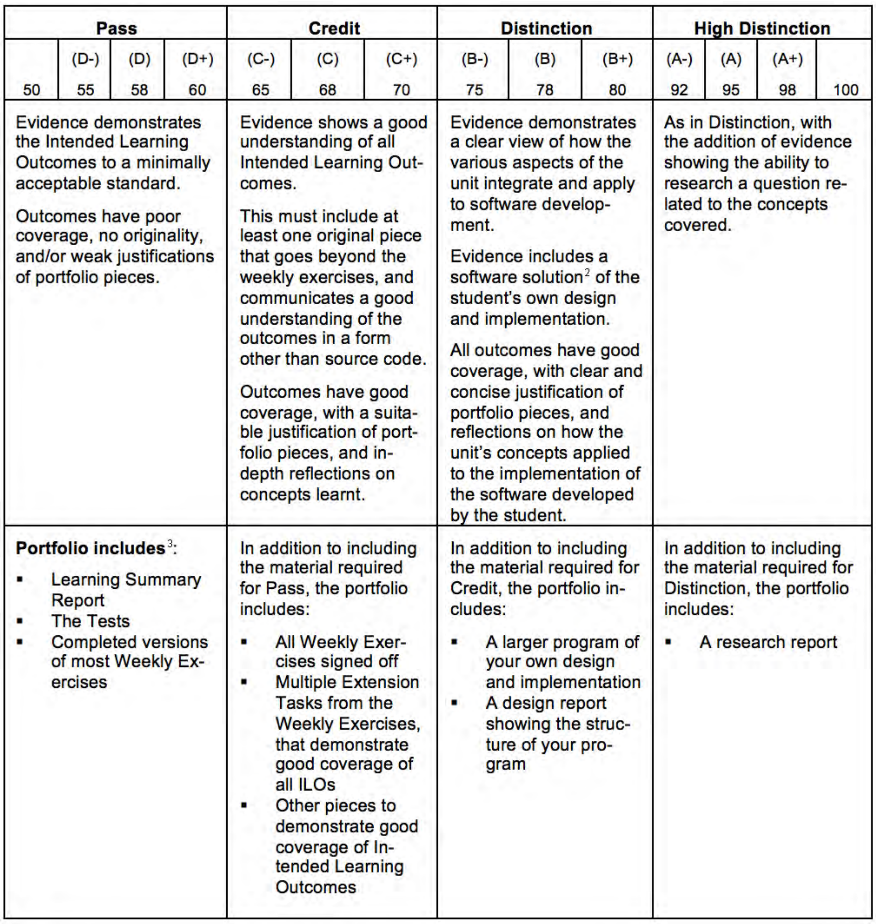
As with Section 7.3.1, this section is divided into three parts with the aim of clearly describing the teaching and learning context, the student cohort, and the details for the thematic analysis.

Teaching and Learning Context

This work analysed the progress of students from Introductory Programming (B) in Iteration 9. In Iteration 9, the teaching period consisted of thirteen weeks, twelve of which were teaching weeks, and a single week semester break in week six. Topics for the twelve lectures are shown in the following list.

1. Programs, Procedure, Compiling and Syntax
2. User Input and Working with Data
3. Control Flow: Branches and Loops
4. Procedural and Structured Programming
5. Arrays
6. Custom Data Types
7. Pointers and Dynamic Memory Management
8. Learning a New Language (C)
9. Portfolio Discussion
10. Arrays and Structures in C, and Recursion
11. Pointers in C, and Backtracking
12. Review and Future Studies

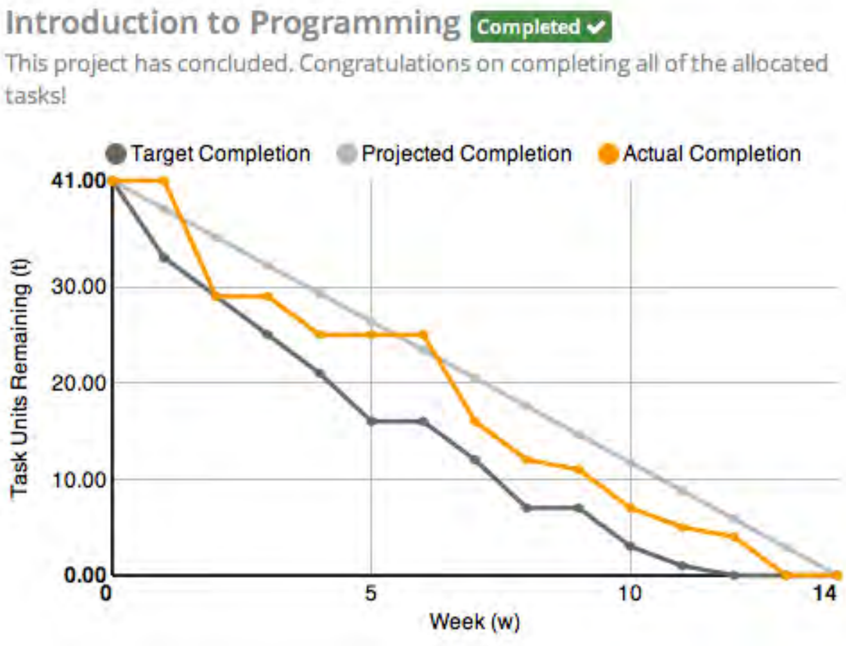
An overview of the assessment criteria for Introductory Programming (B) in Iteration 9 is shown in Figure 7.17. To be eligible for a pass grade students include a range of work of the weekly assessment tasks, and indicated how the pieces included demonstrated coverage of all intended learning outcomes. For Credit, student needed to complete all weekly assignments and include pieces that demonstrated good coverage of all intended learning outcomes. Distinction and High Distinction grades required students to go beyond these weekly tasks. Distinction grades were awarded for being able to apply the concepts learnt in the development a program designed by the student. High Distinction required students to undertake a small research project, and write this up in a short research report.



<table>
<tr><th colspan="4">Pass</th><th colspan="3">Credit</th><th colspan="3">Distinction</th><th colspan="4">High Distinction</th></tr>
<tr><td>50</td><td>(D-)
55</td><td>(D)
58</td><td>(D+)
60</td><td>(C-)
65</td><td>(C)
68</td><td>(C+)
70</td><td>(B-)
75</td><td>(B)
78</td><td>(B+)
80</td><td>(A-)
92</td><td>(A)
95</td><td>(A+)
98</td><td>100</td></tr>
<tr><td colspan="4">Evidence demonstrates the Intended Learning Outcomes to a minimally acceptable standard.
Outcomes have poor coverage, no originality, and/or weak justifications of portfolio pieces.</td><td colspan="3">Evidence shows a good understanding of all Intended Learning Outcomes.
This must include at least one original piece that goes beyond the weekly exercises, and communicates a good understanding of the outcomes in a form other than source code.
Outcomes have good coverage, with a suitable justification of portfolio pieces, and in-depth reflections on concepts learnt.</td><td colspan="3">Evidence demonstrates a clear view of how the various aspects of the unit integrate and apply to software development.
Evidence includes a software solution[2] of the student's own design and implementation.
All outcomes have good coverage, with clear and concise justification of portfolio pieces, and reflections on how the unit's concepts applied to the implementation of the software developed by the student.</td><td colspan="4">As in Distinction, with the addition of evidence showing the ability to research a question related to the concepts covered.</td></tr>
<tr><td colspan="4"><b>Portfolio includes</b>[3]:
▪ Learning Summary Report
▪ The Tests
▪ Completed versions of most Weekly Exercises</td><td colspan="3">In addition to including the material required for Pass, the portfolio includes:
▪ All Weekly Exercises signed off
▪ Multiple Extension Tasks from the Weekly Exercises, that demonstrate good coverage of all ILOs
▪ Other pieces to demonstrate good coverage of Intended Learning Outcomes</td><td colspan="3">In addition to including the material required for Credit, the portfolio includes:
▪ A larger program of your own design and implementation
▪ A design report showing the structure of your program</td><td colspan="4">In addition to including the material required for Distinction, the portfolio includes:
▪ A research report</td></tr>
</table>

Figure 7.17: Overview of assessment criteria provided to students in the unit outline

In Iteration 9, the Doubtfire tool was used to track student progress against the Core Tasks from the weekly exercises. An example of the charts from Introductory Programming (B) in Iteration 9 is shown in Figure 7.18. As described in Chapter 6, the charts show the cumulative amount of work remaining week-by-week, which decreases as work is completed.


Introduction to Programming Completed ✓

This project has concluded. Congratulations on completing all of the allocated tasks!

Figure 7.18: An example burn down chart from the online tool Doubtfire, showing progress against weekly tasks for Introductory Programming (B) in Iteration 9.

Student Cohort and Research Participation

At the end of the teaching period 139 students submitted portfolios for assessment. Of these, 87 agreed to participate in this research. Participation in the research was voluntary, with informed consent being sought using the process described in Section 7.1.3 in lecture 9.

Table 7.10 shows the grade distribution of the submitted portfolios, those made available to this research and of these, those who included their burn down chart in their portfolio.

Table 7.10: Grade distribution of portfolios submitted

	Total	HD	D	C	P	N
Submitted Portfolio	139	7	25	24	79	4
Agreed to Participate	87	6	22	17	39	3
Included Chart	80	6	22	17	33	2

Thematic Analysis to Identify Trends in Progress

The process from Section 7.1.2 was used in order to identify the themes and patterns associated with student progress. To gain familiarisation with the data, the burn down

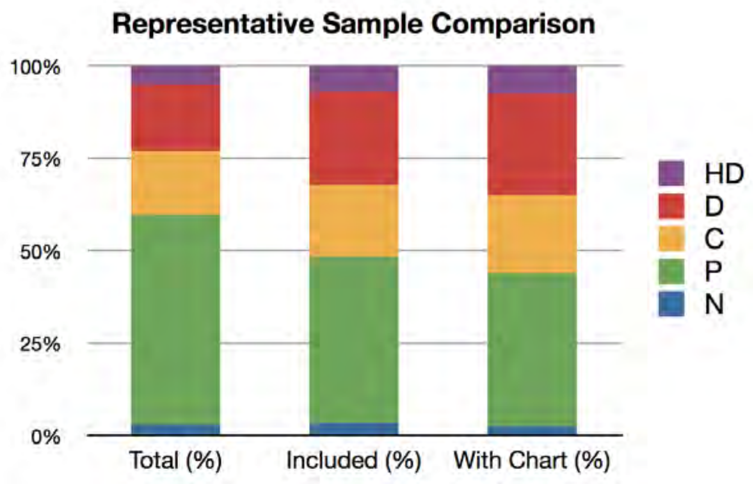


Figure 7.19: Distribution of grades for the full unit, for those students who agreed to participate in the research, and those who included the burndown chart.

charts from each portfolio were scanned. Each page of the resulting document included the chart and a code to identify the portfolio from whence it came. To ease the process of visual classification the charts were scaled to similar size. Each chart was then printed and spread out in a large visual space to enable "visual" themes to be identified, and to determine appropriate classifications.

Charts were classified based on the separation between the **Target Completion** line (target due dates), and the **Actual Completion** line, which indicated student progress based on the work having been signed off as complete by their Tutor. This process resulted in a number of chart classifications, and subclasses. Once the charts were classified they were examined again for any common features that occurred across classifications.

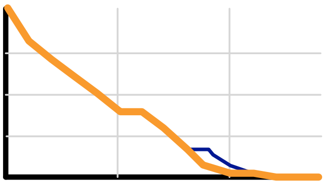
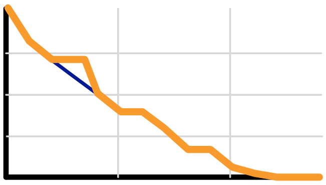
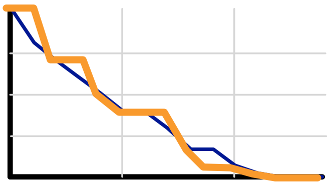
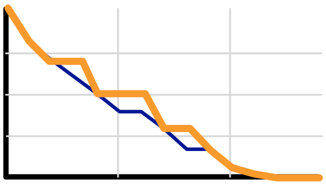
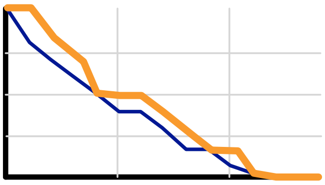
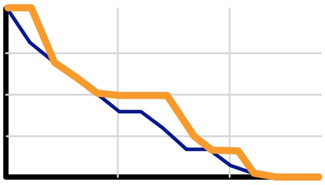
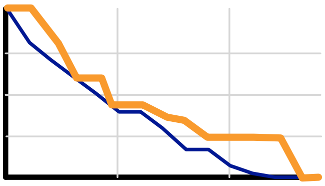
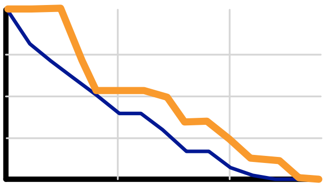
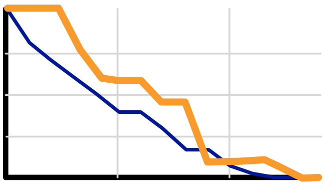
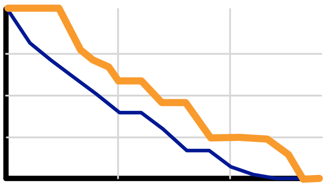
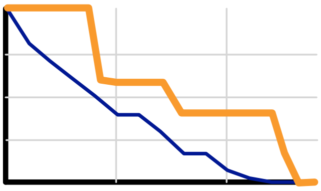
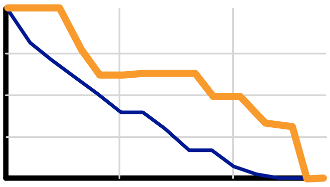
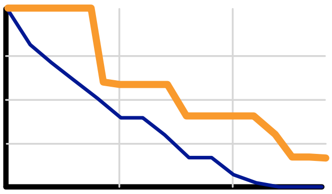
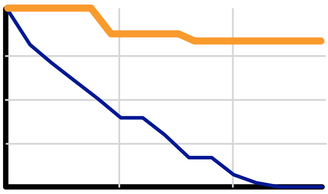
7.4.2 Results

Identifed Chart Classifications

Analysis of the charts made available to this research identified seven different classification related to the distance between the actual and target completion lines.Each classification is described in the following list, and illustrated in Table 7.11.

Table 7.12 shows the frequency of each chart class in the analysed portfolios, and their associated grade distributions. The pie chart in Figure 7.20 shows the distribution of

Table 7.11: Illustrations of the seven chart classifications identified in this work. The thinner blue line represents the Target Completion line, the thicker orange line the Actual Completion.

Class **A**			Tight trend with little diversion from the target completion line
Class **B**			Close to line and similar to Class A but with more deviation, though never more than one to two weeks delay before returning to the target completion line
Class **C**			Consistently trending down, but with sustained small gap(s) from the target completion line
Class **D**			Primarily off the target completion line, with occasional (rare) points where work caught up with the schedule
Class **E**			More distant from the line, but catching up toward the end of the unit
Class **F**			Mostly distant from the line, with progress made in large steps
Class **N**			Not complete, chart included but student was not able to get all work signed off

the classifications, with the highlighted section indicating the number of students who did not get all tasks signed off (Class N).

Table 7.12: Chart classification numbers, and grade distribution.

Class	Total	HD	D	C	P	N
A	7	1	6	0	0	0
B	7	2	1	4	0	0
C	10	1	5	3	1	0
D	9	0	2	5	2	0
E	6	0	3	2	1	0
F	9	2	4	2	1	0
N	32	0	1	1	28	2

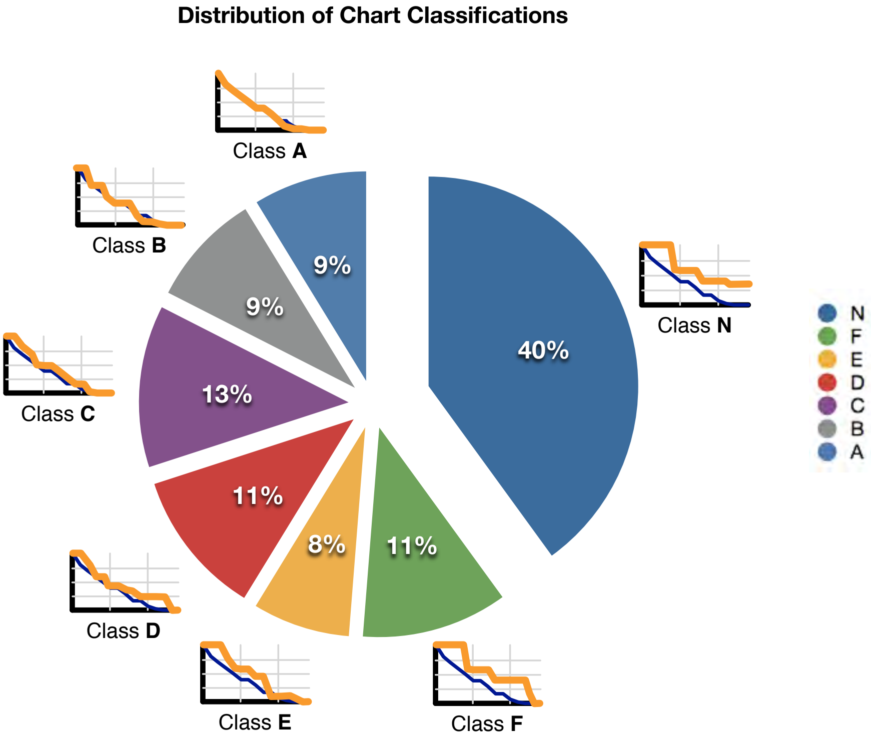


Figure 7.20: Distribution of portfolios according to chart class. Note that 40% of students did not have all weekly tasks signed off.

Chart Features and Subclasses

Five of the seven Class A charts demonstrated cases where students got ahead of the scheduled work. This was also evident in five of the seven Class B charts, and three Class C charts. Of note are two students with Class D charts, more distant from the target completion line, who also managed to get ahead. A total of fifteen students were able to get ahead at some stage, and in all cases this occurred around week nine, after the shift to the C programming language.

Two subclasses were evident in the Class C charts, as illustrated in Figure 7.21. The first subclass included a consistent gap from the scheduled line. This subclass constantly trended down, but did not rejoin the scheduled line during the semester. The second subclass exhibited a jagged shape with constant gaps coming back to the scheduled line at regular intervals. Interestingly none of the second subclass got ahead at any stage. Both subgroups consisted of five charts.

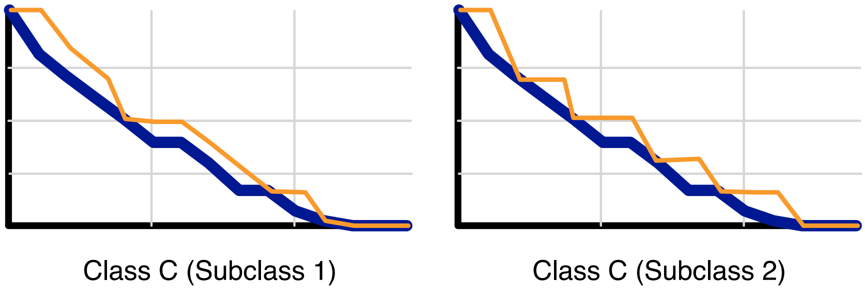

Class C (Subclass 1)
Class C (Subclass 2)

Figure 7.21: Illustrations of the two Class C subclasses, subclass 1 with a consistent gap and subclass 2 with a jagged sawtooth pattern.

The Class D charts had two identified subclasses, as illustrated in Figure 7.22. The larger subclass, subclass 1, had "golf club" shaped charts with effort at the end after a long period without progress (a flat line). The second subclass was characterised by a slow start, but catching up to the target completion line around the middle of the semester. The first subclass consisted of five charts, with four in the second subclass.

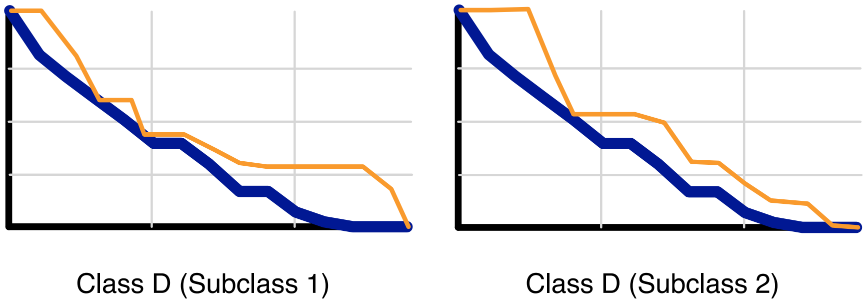
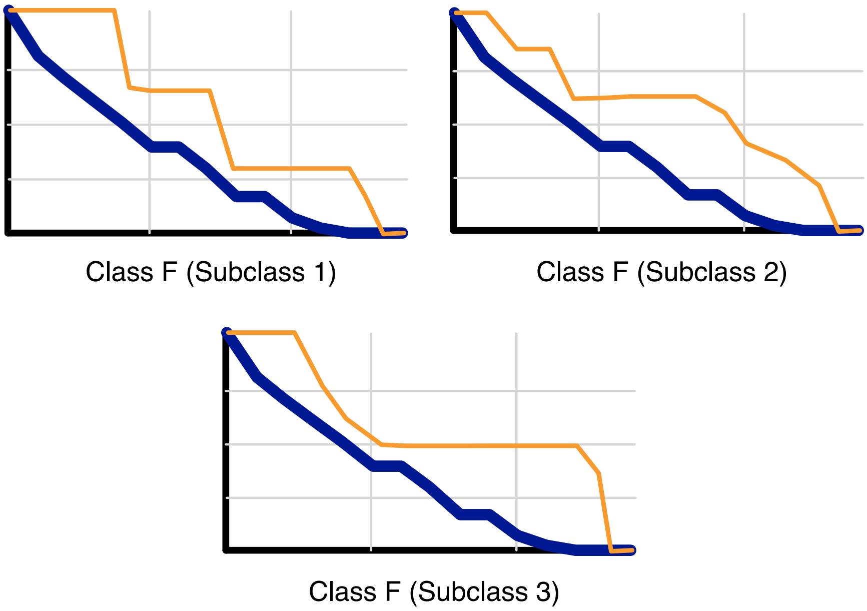

Class D (Subclass 1)
Class D (Subclass 2)

Figure 7.22: Illustrations of the two Class D subclasses, with the "golf club" shaped subclass 1.

Figure 7.23 illustrates the three subclasses that were identified for the charts most distant from the scheduled line (Class F). The subclasses included (1) three large steps to reach the end, (2) a plateau mid semester then progress toward the end, and (3) a similar "golf club" shape to Class D with a large amount of work being signed off at the end of the semester.

Of the 32 students who did not get all weekly tasks signed off (Class N) 13 (41%) had completed 75% or more of the work by the end of the unit. A further 9 had completed more than 50%, 8 had more than 25%, with 2 having less than 25% of the work signed



Class F (Subclass 1)

Class F (Subclass 2)

Class F (Subclass 3)

Figure 7.23: Illustrations of the three Class F subclasses, with three large steps for subclass 1, plateau in subclass 2, and larger "golf club" shape in subclass 3.

off. This is shown in visually in Figure 7.24, with the pie chart in Figure 7.25 illustrating the percentage of each of the end points. Two portfolios in Class N were awarded grades higher than Pass as a result of special consideration.

Of the other classes, most charts that were close to the line (Class B) had their deviation around week five, which coincided with the arrays topic. Whereas the group more distant from the line (Class E) typically had slow starts with strong finishes.

Grades by Chart Classification

Table 7.12 includes the result distribution for each chart class, shown graphically in Figure 7.26. Pass results were primarily from Class N, Credit results were distributed across classes B through F, Distinction across classes A through F, with High Distinctions coming from classes A though C and F.

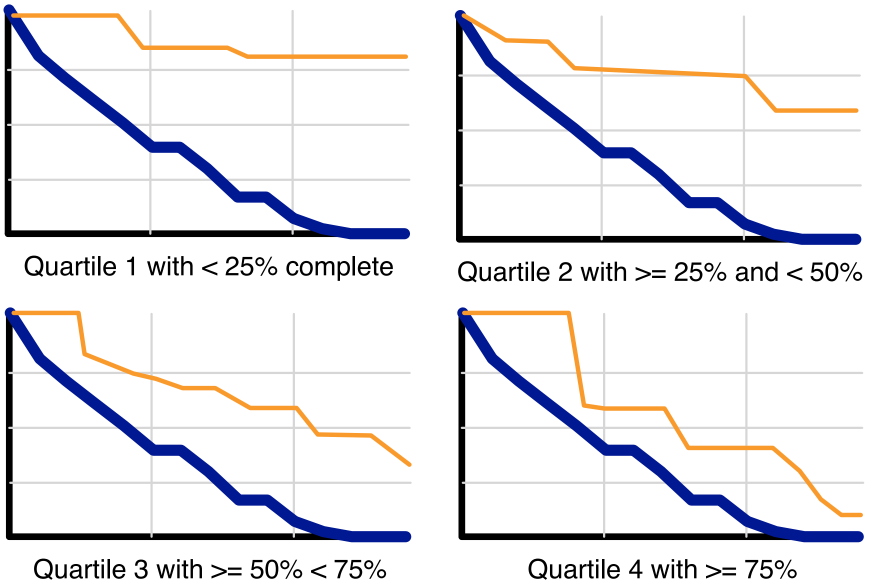


Figure 7.24: Illustrations of the different graph end points for Class N charts

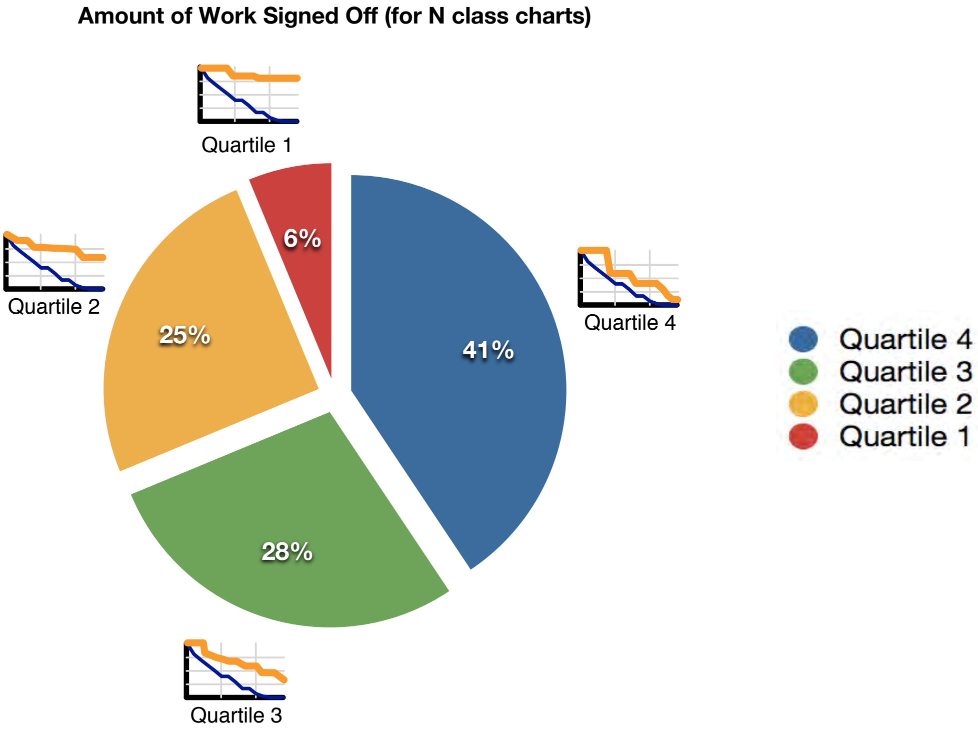


Figure 7.25: Percentage tasks signed off for students with Class N charts.

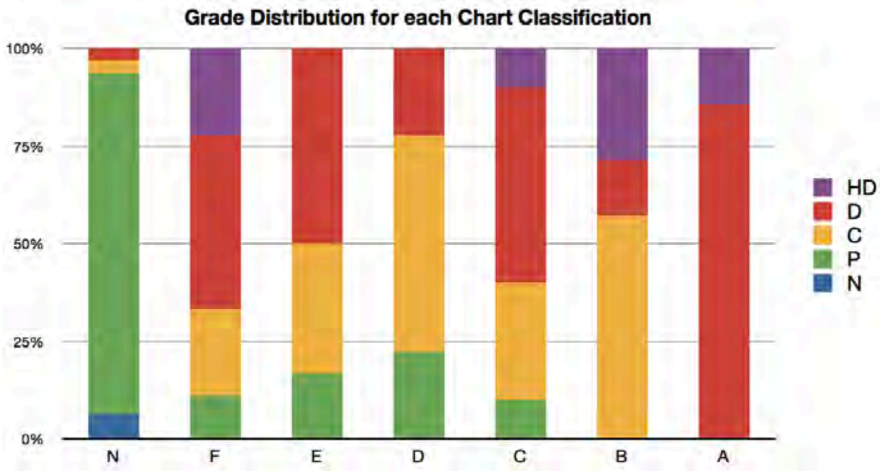


Figure 7.26: Distribution of grades for each of the identified chart classes

7.4.3 Discussion

Focus of Investigation and Quality of Sample Used

Understanding how students progress through an introductory programming unit can provide valuable insight into the strategies they are using and the methodology underpinning the teaching and learning context. Our investigation examined burn down charts included in student portfolios, which captured the rate at which students were able to complete formative assessment tasks and have them signed off by their tutors. Results of the visual thematic analysis, presented in Section 7.4.2, identified a number of different chart classes and presented details on associated result distributions.

The investigation examined a sample of the portfolios submitted in a single semester, with 80 (58%) of the portfolios being included in the analysis. The distribution of grades for these groups are shown in Table 7.10 and Figure 7.19. The relative distribution of grades in the group with charts is a reasonable representation of the entire unit, though the Pass grade is under represented. However, as the pass grade students tended to be clustered in Class N, it is likely that the sample captured the variations in general student progress across all grades.

Participation in Formative Assessment

A number of the charts indicate active student engagement with the formative assessment throughout the semester. This is evident in the chart classes A through C, D (subclass 2), E, and N (for those who completed between 75% but less than 100% of the scheduled tasks). This represented 47 (59%) of the 80 charts, and in all cases the charts indicated ongoing progress, which was only possible by engaging with the formative process.

Our interpretation of the charts from Class D Subclass 1, with its distinctive "golf club" shape, is that these students are likely to have their attention diverted mid-semester. A possible cause is that the due dates for first assignments in parallel, but unrelated, units typically fall in this period. Alternatively, these students may have needed additional time to bring all of the concepts together before continuing on with the C programming language. In either case, toward the end of the unit these students were then able to get back on track, and in some cases get ahead of schedule. It would be interesting for future work to examine the reflections of these students to verify these interpretations.

The large drops in the Class F charts may be evidence of confident students who only occasionally submit their work for assessment. This idea is supported by the large portion of D and HD results in this group.

A cause for concern is the large number of Class N charts, particularly where less than 75% of the formative tasks were completed. This group made up 30 (38%) of the 80 portfolios examined. The unit teaching staff indicated, overwhelmingly, that a lack of student engagement had been of particular concern for the semester under analysis. These students had little interest in learning to program, and seemed to have taken superficial approaches to learning and doing this work. This is evidenced in the both the final results and number of students with Class N charts.

Progress, Grades, and Formative Assessment

In reflecting on the distribution of grades across the different chart classifications, we believe that students have been able to achieve good results using a number of different strategies. If this is compared with units that use summative assessment during the semester, as illustrated in Figure 3.4, students with charts in classes C through F are likely to have lost some marks early on, as they were behind the target completion line at various stages during the semester. With the portfolio approach, 50% of these students were able to go on and achieve a Distinction or higher grade.

Of particular interest are the students with charts in class E, which was characterised by a slow start and a strong finish. These appear to be students who struggled with concepts initially but eventually gained sufficient mastery to quickly catch up later in the semester. It is interesting to note the large portion of these students who were subsequently able to achieve a Distinction result. Even with the slow start, these students were able to apply the concepts learnt to the creation of their own program. Had this been a traditional programming unit, using summative assessment during the semester, loss of marks early on may have lead to them not receiving a grade that represented their final learning outcome. If these students lacked confidence, then this negative reinforcement may possibly discourage them from attempting to master the concepts, and reinforce any negative opinion they have of the field in general.

Concept Based Approach

The charts also provide evidence of the effectiveness of the concept-based approach used. A total of 15 (19%) of the 80 charts showed students getting ahead of the schedule after the switch to the C programming language. This mostly consisted of students with charts in classes A and B, though it also includes several students in classes C and E.

After the language change, tasks were concerned with expressing previously presented concepts in the C language. We propose that students who committed to understanding the underlying programming concepts, rather than focussing on the Pascal language itself, were able to exploit this understanding when introduced to C. Analysis of reflections in these portfolios may provide additional evidence to help verify this.

Reflecting on Programming Issues

This work helps support two of the general learning issues, as identified in Section 7.3, facing students: getting started, and learning through mistakes. Other than classes A and B, most of the student's progress appears to indicate that they had issues getting started. This is understandable, given the highly interrelated nature of the programming concepts presented in early weeks. The progress students showed later in the semester also demonstrates that students are able to make progress while learning through mistakes.

In relation to time management – the number one issue identified in our previous work – further analysis of student reflections is needed to determine if the tool provided them with useful feedback and motivation to keep on track. That said, the tool was actively used and provided staff with useful information on student progress throughout the teaching period.

7.4.4 Summary

In this section we presented a visual thematic analysis of student progress as evidenced by burn down charts included in their final portfolios. Details of the unit, its delivery, assessment, and its use of burn down charts to track progress was described as a context for this work.

Thematic analysis identified several different chart classes and subclasses, based on the shape of the chart and its distance from the target completion line. These classes provided some indication of students' approach to learning, with most concentrating on language features rather than underlying programming concepts. The different chart classes also presented evidence that students struggle to get started with introductory programming, but that once concepts were mastered they were able to catch up and, in some cases, get ahead of schedule. Interestingly, the use of formative feedback resulted in students being able to achieve high grades even when they struggled initially.

8
Discussion

Chapter 3 presented twelve principles that extend the core principle of constructive alignment to help create a student-centred learning environment focusing on constructive learning theories. These principles were then used to guide the creation of the model of constructive alignment for introductory programming presented in Chapter 4. Chapters 5 and 6 provided details of two example units implemented using this approach, and some of the supporting resources used to assist students in constructing appropriate knowledge. Analysis of results from these units was presented in Chapter 7, which discussed the iterative evolution of the approach and assessment criteria, issues students faced, and the use of the Doubtfire tool for tracking student progress.

This discussion chapter considers the overall experience of developing and delivering units using this approach. Section 8.1 reiterates the importance of the principles stated in Chapter 3, and discusses the likely implications of failing to address each of these principles. Following this, Section 8.2 discusses the approach in relation to contexts beyond introductory programming, and provides support for Biggs' claim for the general applicability of constructive alignment. Section 8.3 relates the approach presented to prior work on constructive alignment, including Biggs' original work on constructive alignment with portfolio assessment, and to prior work on the teaching of introductory programming. Challenges for the wider adoption of this approach are then discussed in Section 8.4, followed by suggestions for how to approach the transition from traditional forms of assessment to portfolio assessment in Section 8.5. The chapter concludes with closing comments in Section 8.6.

8.1 Principles in Review

Chapter 3 provided twelve principles which underlie the creation of the student centred approach to teaching introductory programming presented in Chapter 4. While each principle has its own individual merits, we believe that they are all required to work together in order to achieve the outcomes presented; none should be removed. This section draws upon the experiences of teaching staff, as reported in Chapter 7, to discuss the potential impact of failing to include each of the principles on the overall learning environment.

8.1.1 Constructive Learning Theories (P1)

Principle 1 plays a central role in providing the motivation for all of the other principles. Where understanding is not seen as being constructed by individual learners, there is little need to attempt to create teaching and learning activities that actively engage students. Effective teaching becomes a matter of effectively delivering the required knowledge, resulting in teacher-centred activities coming to the fore.

Adopting constructive learning theories in teaching and learning activities addresses only part of the overall teaching and learning environment. The central role of assessment implies that such theories need to extend beyond teaching and learning activities to assessment. Applying traditional assignment and exam assessment schemes limits the opportunity for students to demonstrate their understanding in a way that is meaningful to them. Given students role in constructing their knowledge, they are best situated to determine what represents the best demonstration of their understanding. Implementing the other principles without adopting more flexible assessment approaches is likely, therefore, to limit the effectiveness of the learning environment overall as students are limited by the assessment approach.

These competing views can be best thought of as a continuum based upon communication; from knowledge transmission with objectivist learning theories, to discover learning with constructivist theories. This continuum is shown visually in Figure 8.1, which depicts the underlying concepts that drive the actions of teaching staff when they hold such views. At the objectivist end of the continuum communication is key, and teaching staff work to communicate all of the aspects students need to understand. This view is teacher-centred as the teacher communicates the required understanding for students to passively absorb. At the other end, the extreme constructivist view discards any value in communication, instead students are placed in situations and asked to “discover” the knowledge themselves. Adopting either of these extreme positions is not likely to result in an effective, student-centred, learning environment.

Principle 1 requires a middle ground approach. To embrace this approach, educators need to accepts the central role of the student in constructing their own knowledge, while also accepting that communication can be a powerful tool to help guide this construction. Communication then becomes a valuable tool, with teaching staff being encouraged to communicate as little or as much as is *needed* by the students at that stage of their learning. Communication is not used to transfer knowledge, but to help guide students.

Reflections from the evaluation of the unit deliveries across the various iterations in Chapter 7 provide support for taking this middle ground approach. Teaching staff attributed many of the failings of the object oriented programming unit in Iteration 2 to an overly zealous application of constructive learning theories. Moving back from this extreme, to a more moderate application of constructive learning theories addressed this situation in later iterations, and as staff expertise in guiding students improved, so did student results.

8.1.2 Aligned Curriculum (P2)

Aligning all aspects of the teaching and learning environment (P2) makes good intuitive sense. Failing to address this principle is likely to result in one of three potential outcomes based upon misalignment between the three component parts: teaching and learning activities, assessment tasks, and intended learning outcomes.

1. Where assessment aligns to intended learning outcomes, but activities do not, students are likely to be unprepared for the assessment tasks. Whilst assessment defines the curriculum for students, activities fail to provide a means of preparing them for this assessment.
2. Where activities align to intended learning outcomes, but assessment does not, students may construct appropriate knowledge but the assessment is unlikely to identify this. Final student grades fail to represent student learning outcomes.
3. Where neither assessment nor activities align with intended learning outcomes, students are not likely to learn what was intended. While students may effectively learning something valuable, and the assessment appropriately report this, the misalignment is likely to cause issues for students in later units, or professional life.

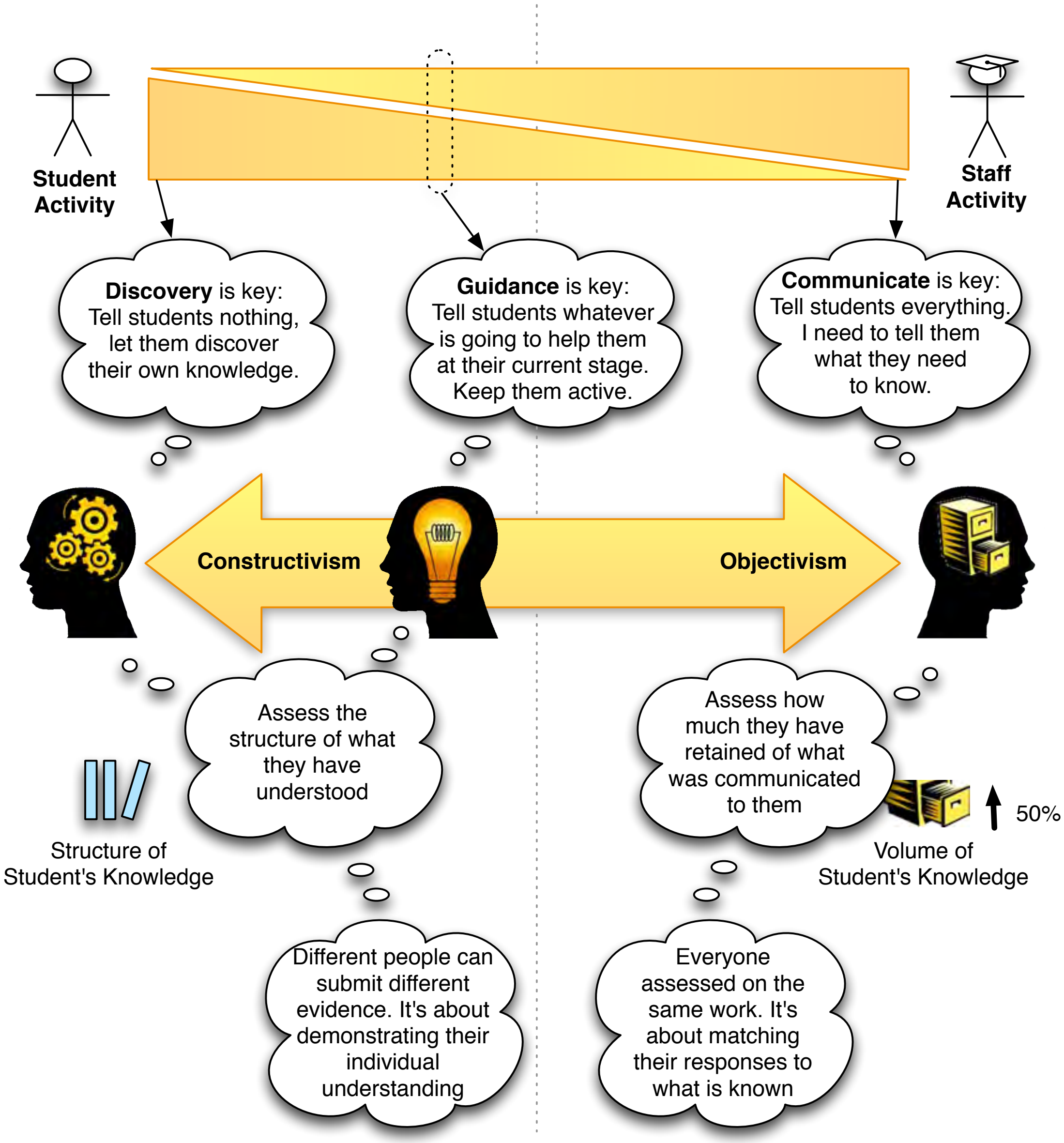

Student Activity
Staff Activity
Discovery is key: Tell students nothing, let them discover their own knowledge.
Guidance is key: Tell students whatever is going to help them at their current stage. Keep them active.
Communicate is key: Tell students everything. I need to tell them what they need to know.
Constructivism
Objectivism
Assess the structure of what they have understood
Assess how much they have retained of what was communicated to them
Structure of Student's Knowledge
50%
Volume of Student's Knowledge
Different people can submit different evidence. It's about demonstrating their individual understanding
Everyone assessed on the same work. It's about matching their responses to what is known

Figure 8.1: Thoughts that guide teaching staff at either end of the constructivism-objectivism continuum. Chapter 3 advocates a pragmatic approach to constructivism, somewhere on the constructivist side of the continuum.

Alignment is, therefore, a critical aspect in creating an effective learning environment. By aiming to achieve consistency between teaching and learning activities, assessment tasks, and intended learning outcomes, teaching staff provide students with the greatest opportunity to learn the required knowledge in an effective manner.

It is also critical to understand that students must also be involved in this alignment process. The interplay between Principle 1 and Principle 2 means that students, not staff, are in the best position to report on how teaching and learning activities and assessment tasks aligned with the intended learning outcomes. It is the students who followed the planned activities, carried out the assessment tasks all of which helped them in the construction of their knowledge. It is the students, therefore, who truly experience and know how these activities aligned.

The implication of this is that there is not likely to be one measure of alignment for a set of teaching and learning activities. Each student's learning will be unique, based upon their prior experiences and current knowledge structures, resulting in the activities influencing each student in a unique manner. Alignment reported by staff in carefully prepared matrices are, therefore, illustrative at best. It should be important for activities to provide students with a range of opportunities to engage with each learning outcome, giving students the best opportunity to actually achieve these learning outcomes.

The matrices for the units presented in Chapter 5 demonstrate this wide coverage of outcomes, as shown in Table 5.2 and Table 5.4. In these units students were provided with a number of opportunities to engage with each of the intended learning outcomes.

8.1.3 Assessing Learning Outcomes (P3)

Principle 3 aims to encourage educators to evaluate student learning outcomes in terms of a student's developed understanding at the end of the teaching period. To achieve this, Chapter 3 advocated the use of frequent formative feedback and assessment tasks that require students to articulate their understanding, in addition to practical application, of the concepts covered. The following list outlines three ways in which the principle of assessing student outcomes can be violated, each of which is then discussed.

1. Frequency of feedback is reduced.
2. Assessment might focus on product outcomes and not require students to artic-

ulate their understanding.

3. Summative tasks could be used during the teaching period.

With regards to point 1, there may be some temptation to reduce the frequency of formative feedback, to reduce staff and student workloads. However, rapid iterations are key to ensuring student learning remains "on track." As the frequency of formative feedback is reduced, there are less opportunities for staff to positively influence student outcomes, and overall results are likely to be weaker.

This temptation is also ill-founded, as frequent feedback does not aim to increase workloads but to distribute this work more consistently throughout the teaching period. Figure 8.2 illustrates this aim, showing that the goal is to provide smaller frequent feedback in order to maintain similar overall effort. Where this can be achieved, students are more likely to develop appropriate understandings, as misconceptions can be addressed earlier in the process. For the example units presented in Chapter 5, the use of weekly formative feedback helped ensure that each small task could be assessed quickly, thereby ensuring students received their feedback in a timely manner. Had larger tasks been used additional time would be needed to assess these thereby further delay feedback.

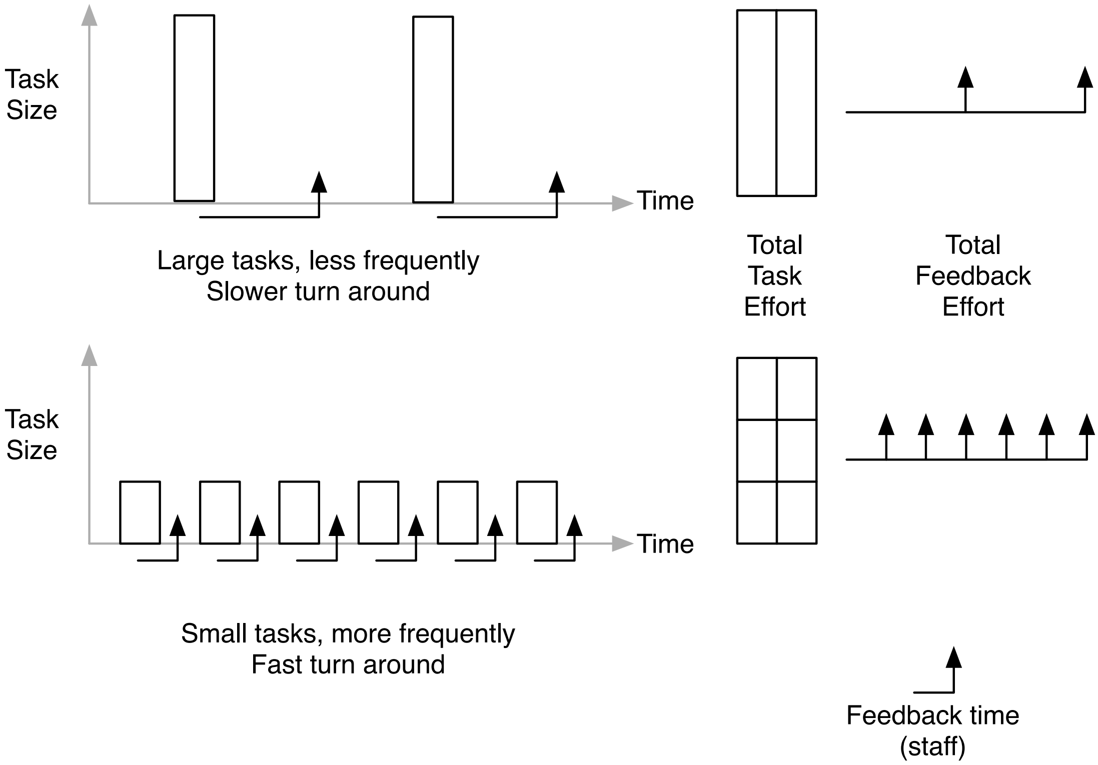


Figure 8.2: Illustration of time allocation to assessment tasks, with rectangle areas representing effort expended by students preparing submissions, and arrows representing effort for staff to provide feedback.

The second issue listed above relates to the assessment of student understanding, in addition to product outcomes. This is particularly relevant to programming units, where it is easy to assess the programs students create rather than attempting to assess their understanding. Assessing product outcomes alone encourages surface approaches to learning, as it is the product and not the understanding that is being assessed. Including some tasks that require students to articulate their understanding provides opportunities for students to engage appropriate cognitive levels, helping them develop the required understanding, while also communicating the importance of this understanding to the students. In this way, such tasks help to encourage students to appropriately engage with learning activities, and to use deep approaches to learning.

The final issue relates to the use of summative assessment, rather than formative feedback, during the teaching period. Including summative assessment during the semester breaks several critical aspects of the approach presented. Assessing tasks within the teaching period means that an overall assessment of student learning outcomes is no longer possible. This was evident in the results from the first iteration discussed in Chapter 7, with results failing to match outcomes demonstrated in student portfolios.

Using summative assessment during the teaching period also limits the likelihood of students incorporating feedback they receive. Summative assessment is, by its very nature, final, and so students are not encouraged to learn from such assessment. A positive aspect of using formative feedback during the delivery of the example units was that understanding became the key focus. Tasks were not complete until students had demonstrated the required understanding. There was no punishment for not having understood an aspect of a topic on the first attempt, freeing teaching staff to provide relevant feedback and guide students toward the required understanding.

While summative assessment during the teaching period works against the overall principles stated in Chapter 3, the distinction between formative feedback and summative assessment changes when using summative assessment that aims to provide a qualitative, holistic, assessment of student outcomes. Constructive learning theories require staff to gain an understanding of the likely level of understanding students have developed in order to provide formative feedback. As a result, teaching staff are able to perform summative assessment of student performances at any stage during the teaching period. In effect, the assessment at the end of the teaching period represents an arbitrary point in time at which this assessment does occur. Ideally, additional flexibility in education could allow this point to be adjusted for individual students further catering for a wide range of capabilities.

Formative feedback with qualitative, holistic, summative assessment of student outcomes is seen as critical to the success of the approach presented in this thesis.

8.1.4 Supporting Principles (P4 to P12)

As shown in Figure 8.3, principles 4 to 9 can be considered as providing underlying support for the central principles related to the adoption of constructive learning theories, alignment of curriculum, and assessment of learning outcomes. While embracing these principles helps support the central principles, it is possible that alternatives could offer similar outcomes.

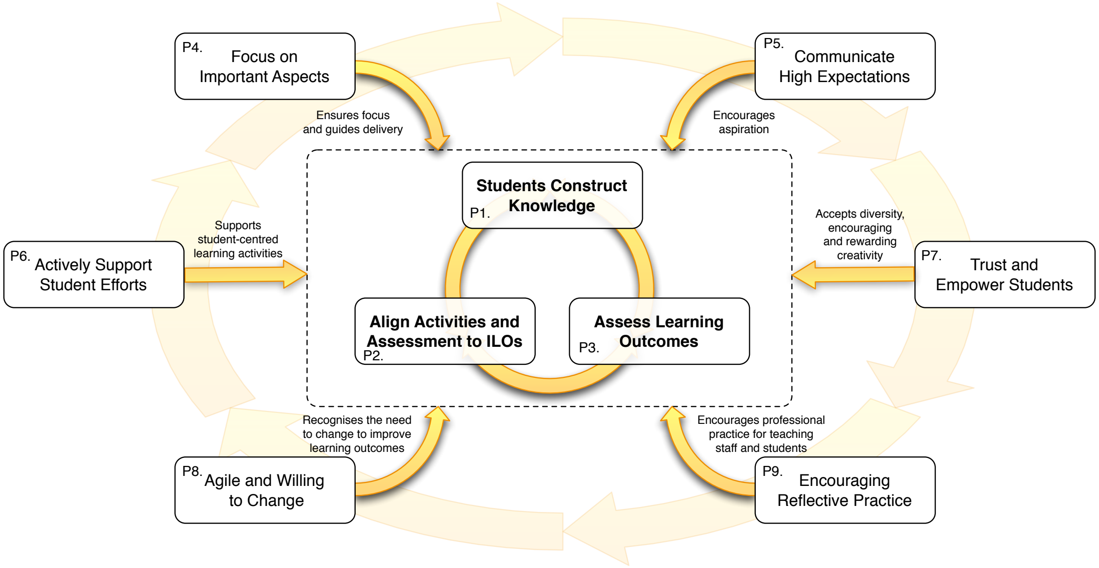

P4. Focus on Important Aspects
P5. Communicate High Expectations
Ensures focus and guides delivery
Encourages aspiration
Students Construct Knowledge
P1.
Supports student-centred learning activities
Accepts diversity, encouraging and rewarding creativity
P6. Actively Support Student Efforts
P7. Trust and Empower Students
Align Activities and Assessment to ILOs
P2.
Assess Learning Outcomes
P3.
Recognises the need to change to improve learning outcomes
Encourages professional practice for teaching staff and students
P8. Agile and Willing to Change
P9. Encouraging Reflective Practice

Figure 8.3: An alternative view of the principles outlined in Chapter 3, showing Principles 4 to 9 providing support for the central principles of constructive learning theories, alignment of curriculum, and assessment of learning outcomes.

Principle 4 encourages a focus on communicating only key concepts, and providing students with access to details which they can use as needed. This principle was instrumental in the displacement of lecture content to teaching and learning resources; an approach that also resulted in light-weight teaching and learning activities that made it easier to make iterative improvements in line with Principle 8. If this principle had not been embedded within the example units then the shift away from teacher-focused lectures, to more constructive activities with greater student engagement, would have been more challenging.

Principle 5 encourages teaching staff to communicate high expectations to their students, in an effort to promote deep learning. Staff expectations are communicated deliberately, through assessment tasks and feedback, and indirectly, through tone and attitude. In all cases these expectations encourage students to strive to achieve excellence. In the example units, high expectations were communicated throughout the iterative process, with work only being signed off when it was completed to a high standard. The handling of tests in the example units is a particularly relevant case, with students needing improve their answers even when they had successfully completed most of the test questions. Had these high expectations not been communicated, students would be more likely to lower their internal expectations, as was encountered in the first few iterations discussed Chapter 7.

Principle 6 aims to actively support student efforts. When constructive learning theories are adopted, teaching staff can no longer *tell* students what they need to know; instead staff need to *guide* students in their learning. Additional assistance helps students develop their understanding as they attempt to engage with learning activities. With challenging unit content, such as in introductory programming units, this support helps ensure a larger number of students will successfully complete the unit. Where this support is not provided results are likely to suffer, either in terms of overall expectations or number of students successfully completing the unit.

Principle 7 indicates a need to view students as being genuinely motivated to learn, a Theory-Y view of education. Where this principle is not held, staff are likely to resist the move away from summative assessment during the teaching period. Theory-X strategies of coercions and punitive assessment need to be avoided for the benefits of formative feedback to be realised.

Principle 8 recognises the need to be agile and willing to change. The iterative development of the example units reported in Chapter 7 provides a clear demonstration of the value of this principle. In many regards, initial implementations of portfolio assessment were seen as only partially successful, but by embracing change, iterative improvements resulted in the positive learning environment experienced in later iterations. Where change is not embraced, situations are not likely to improve on their own. However, once the approach taken does result in good learning outcomes, the need for change reduces.

Finally, Principle 9 encourages the use of reflection by both staff and students. Students are encouraged to reflect on their learning, while staff reflect on student outcome and use this feedback to direct change. Reflection works “hand-in-hand” with Principle 8 in shaping the units outlined in Chapter 5, as reported in Chapter 7. Failing to incorporate this principle would lessen the impact of any change. Similarly, failing to encourage students to reflect on their learning is not taking full advantage of the learning environment created.

8.1.5 Principles Related to "What" We Teach (P10, P11, and P12)

In addition to the nine principles related to "how" we teach, Chapter 3 presented three principles on "what" we teach. These aimed to work together with the "how" principles to create an effective environment for students learning introductory programming concepts. As each of these principles relates specifically to the teaching of introductory programming, they may not be relevant to teaching other units. This also implies that these principles need not even be addressed by other introductory programming units.

Principle 10 indicates that introductory programming units should aim to communicate a consistent set of programming concepts centred upon a programming paradigm. In the example units presented in Chapter 5, the procedural paradigm was used in the formation of the introductory programming unit, while object oriented programming principles were focused upon in a second unit. While this selection, and order, of paradigms could be adjusted, it would be hard to imagine an introductory unit that did not centre on an in-depth coverage of a single paradigm.

Each paradigm centres around a number of programming abstractions, with programs being conceptually constructed through the abstract definition and configuration of these abstractions. Principle 11 encourages educators to focus on communicating these conceptual structures, and associated programming concepts. Associated concepts, such as DuBoulay's notional machine (DuBoulay 1986), then become the focus, rather than the specifics of a programming language's syntax. The realisation of this principle enabled the example units to introduce students to a range of programming languages, helping them better understand the underlying concepts.

While this concept focus was important for these units, Principle 11 is not critical to the the overall approach. Ideally educators should aim to communicate underlying concepts, as these provide students with a broader understanding of programming in general. Where depth in a single language is desired, this could still be achieved with a focus on concepts. So there is little reason not to adopt this principle, though it does require a certain level of expertise from teaching staff, and availability of resources for students to use as they learn syntax themselves.

Finally, Principle 12 advocates for *appropriate* use of programming languages. Constructive learning theories indicate that the goal of education is to help students "think and act" like experts. This necessitates the appropriate use of the tools, and concepts, in teaching and learning activities. By not adhering to this principle, students can learn how *not to* do something. This is likely to result in students developing inappropriate understandings of how the relevant tools or concepts should function. These understandings must then be unlearnt before students are able to "think and act" as experts.

8.2 General Applicability of Approach

Chapter 4 described an approach that has been used to deliver a constructively aligned introductory programming units that embodied all of the principles from Chapter 3, and outlined the overall strategy taken to unit delivery and assessment. This section discusses the wider applicability of the approach, and constructive alignment in general.

8.2.1 Applicability of Constructive Alignment in General

Biggs' original proposal of constructive alignment concluded with the following question:

> "Can the principle of constructive alignment be generalised from the context of in-service teacher education?" Biggs (1996)

While others have applied the core principle of constructive alignment, as discussed in Chapter 2, this thesis has identified additional principles that help create the "web of consistency" phenomenon that inspired Biggs' constructive alignment model (Biggs 1996, 1999). The twelve principles stated in Chapter 3 underpin an approach to constructive alignment described in Chapter 4 which, together with the principles, guided the design, development, and delivery of the units described in Chapter 5. The results? A supportive, student-centred, teaching and learning environment in which, to use the words of Biggs & Tang (2007) (p.51), students consistently "*stun*" teaching staff with the "*rich and exciting*" work they demonstrate in their portfolios.

As outlined by Biggs (1996), the model of constructive alignment makes intuitive sense, and comes together as a whole when the following conditions are met.

1. Teaching staff are clear about the *intended learning outcomes*.
2. *Assessment criteria* are provided to indicate how these outcomes can be met at various levels of achievement, forming a hierarchy from barely satisfactory to most acceptable.

3. Students are required to *perform activities* that are likely to elicit the required understandings.
4. Students *provide evidence* that their learning has matched the stated outcomes.

Chapter 4 demonstrated how the guiding principles described in Chapter 3 can be applied to create an approach to teaching introductory programming that meet, and in many regards go beyond, Biggs' four conditions. The processes described started with the clear expression of intended learning outcomes, with the development of assessment criteria providing the required performance objectives required for different grade outcomes. The development of teaching and learning activities aimed to provide students with tasks likely to engage them in activities that will enable them to construct appropriate understandings, and produce evidence they can include to demonstrate their newly gained knowledge. This evidence could then be collected together and presented in student portfolios as a means of demonstrating how the stated objectives had been met.

Therefore, the approach presented in Chapter 4, along with the example implementation discussed in Chapters 5 to 7, provide additional support for Biggs' suggestion that constructive alignment using portfolio assessment can be generalised to a range of educational contexts.

8.2.2 Applicability of Approach to Other Units

The general applicability of constructive alignment gives rise to the question: Can the approach presented in Chapter 4 be used beyond the context of introductory programming? We believe so, and have been working with others to bring this approach to a wider range of units. To date, this approach has been used in the design and delivery of the following units:

- Artificial Intelligence for Game used intended learning outcomes related to the use of Artificial Intelligence in creating immersive gaming experiences. Student portfolios included a number of programs to demonstrate various techniques, with higher grades demonstrating the application of learnt concepts in the development of a program of the students own invention.
- Concurrent Programming covered the use and implementation of concurrency control mechanisms such as semaphores, barriers, and channels. Portfolios included implementations of these utilities, along with programs demonstrating solutions to classic synchronisation problems.
- Enterprise Software Development involved the use of a range of software tools

to implement larger, multi-tier, solutions to business scenarios. Portfolios included demonstrations of various technologies, architectural designs, and technical demonstrations of core components of these designs.
- Games Programming introduced concepts related to game design, and the implementation of game engine concepts. Portfolios included demonstrations of various programming techniques and optimisations related to game development, with students implementing game prototypes for higher grades.
- Mobile Software Development explored the implementation of software for mobile devices, and associated usability issues. Students applied concepts they learnt in the creation of their own programs for higher grades.
- Research Project involved undergraduate students undertaking and documenting a small research project. Portfolios included artefacts created from the research project, which consisted of, at least, a research proposal and plan, research report, artefacts associated with an oral presentation, and a learning summary report.

In each case the units involved incorporated the principles from Chapter 3, with the central role of programming paradigms (P10) in the development of introductory programming units being adjusted to focus on key aspects relevant for each unit. The use of portfolio assessment in each case meant that similar, in many cases identical, assessment criteria were used for the different intended learning outcomes.

These units have been successfully delivered using portfolio assessment, and have demonstrated similar student-centred learning environments that focus on encouraging and rewarding students for developing a depth of understanding. In general, these units exhibit many of the positive aspects from reported in Chapter 7, including:

- Portfolios demonstrating a range of capabilities from completing core aspects to creating custom projects and research reports.
- High pass rates, with few students failing.
- Good student evaluations in student feedback on teaching surveys.
- Staff indicate good productivity from applying portfolio assessment, with time being spent assisting students rather than assessing them.

Future work could look to evaluate the effectiveness of constructive alignment with portfolio assessment in these and other units as the approach is applied more widely.

8.2.3 Applicability to Team Work and Project Units

The use of the approach from Chapter 4 could provide a means of assessing learning outcomes for students in units that include significant use of group work, such as with team-based final year capstone projects. The focus on assessing learning outcomes (P3) over assessing product outcomes, and the portfolio's focus on students demonstrating how they have met the unit's intended learning outcomes, provide a means of assessing individuals.

In these units, students work as part of a team, with obvious challenges in assessing the learning outcomes from individual students as final work products are a team effort. Using the approach from Chapter 4 it would be possible to create a learning environment for these units that meet the following criteria:

- Intended learning outcomes capture the required technical and teamwork skills and understandings students need to demonstrate to successfully complete the unit.
- Assessment criteria indicate how these outcomes need to be demonstrated in order to achieve different grade outcomes.
- Students engage in teamwork activities, which are likely to elicit the required outcomes.
- Each student collects personal evidence that *they* have met all of the intended learning outcomes, and aligns their evidence in a Learning Summary Report, which is presented for assessment.

In this way, each student's grade would reflect how well they, as *individuals*, have met the intended learning outcomes. Creating such a scheme would require the embodiment of all principles stated in Chapter 3, particularly the need to trust and empower students in their learning (P7).

8.2.4 Applicability to Large Class Sizes

One place where we differ from the recommendations of Biggs & Tang (2007) is in the use of portfolios for larger class sizes. Incorporating frequent formative feedback, tracked by the online Doubtfire tool, made it possible to use portfolio assessment with classes in excess of 300 students (323 students completed the introductory programming units in iteration 8). The frequent formative feedback meant that student work submitted in their portfolios had *already* been checked, often multiple times, and if completed successfully this had been indicated in the Doubtfire tool. As a result, the majority of student work did not need to be re-checked in their final portfolios, and grades could be quickly determined from the Learning Summary Report and records of assessment feedback and test completion.

Reflections from teaching staff indicate that the process enabled student portfolios to be assessed in significantly less time than it took to assess the previously used exams – which consisted of multiple choice, short answer, and coding questions. It was also felt that the grades awarded aligned with the capabilities students had demonstrated during the teaching period.

Given this, and ongoing improvements through reflective practice, it is also believed that the use of portfolio assessment could scale to significantly larger class sizes.

8.2.5 Applicability of Overall Strategy

Section 4.1 described the overall strategy used to underpin the development and delivery of the units later presented in Chapter 5. This strategy centred around the use of portfolio assessment as an assessment approach, and utilised a variety of student-centred delivery approaches to help embed constructive learning theories in unit delivery. This section discusses these decisions, and if and how alternatives could be incorporated.

Portfolio Assessment

Biggs' early work on constructive alignment strongly advocated for the use of portfolio assessment (Biggs 1996, 1999). Biggs & Tang (2007) provided a range of alternative means of implementing constructive alignment under the premise that portfolio assessment is not applicable for large class sizes. However, as shown in Chapter 7, our approach has been able to scale from teaching tens of students in early iterations, to hundreds of students in later iterations through the implementation of the principles described in Chapter 3.

Alternative assessment schemes involving the use of summative assessment during the teaching period, and examinations at its end, are not able to reproduce the same "web of consistency" as they break several of the core principles outlined in Chapter 3. Existing work reporting on applications of constructive alignment are largely representative of this situation, with the large majority still producing students final grade as a combination of weighted assignment and exam outcomes. This traditional approach to assessment fail to meet the conditions outlined by Biggs (1996):

- Teaching staff may clearly express *intended learning outcomes*, but students will focus on the assignments and exams as "assessment defines curriculum" for the students (Ramsden 1992).
- *Assessment criteria* allocate marks for the assignments and exam questions, failing to provide a hierarchy of criteria related to performance against the intended learning outcomes.
- Instead of providing evidence that their learning has matched the stated outcomes, students provide solutions to the assignments and exam questions.

The beauty of constructive alignment with portfolio assessment is its *simplicity*. Intended learning outcomes provide the central focus for teaching staff and students, who work together during the teaching period to help students construct evidence that they can achieve these stated outcomes. Hierarchial assessment criteria then support this to encourage and reward students for demonstrating depth of understanding. This clear focus is essential in developing Biggs' "web of consistency", and are central to the successes outlined in Chapter 7.

This simplicity is lost with more traditional forms of assessment, even when carefully aligned with intended learning outcomes. Separating the assessment tasks from the intended learning outcomes diminishes the importance of the outcomes and focuses students on the assessment task and its marking criteria. From the students perspective what is important is maximising their marks on the assessment tasks, potentially using shallow approaches that fail to develop appropriate understanding. While portfolio assessment does not guarantee that all students will deeply approach learning, it is designed to encourage and reward students who do.

Delivery Approach

Section 4.1 outlined the incorporation of constructive learning theories into the teaching and learning activities using the Beyond Bullet Points approach for lecture presentations, interactive lecture demonstrations, and laboratory sessions with lab, core, and optional tasks. While each of these was seen as effective in the delivery of the example units, other units and different teaching staff are likely to need different strategies.

In deciding upon a delivery strategy, teaching staff need to identify tasks that are likely to actively involve students in constructing their own knowledge. Where previous activities may have relied upon knowledge transmission (such as the standard lecture) these need to be redesigned with constructivist ideals in mind.

8.2.6 Applicability of Activities within the Model

After outlining the overall strategy taken in this work, Chapter 4 presented the model of constructive alignment used, and described a number of processes that existed within this model. Given the use of portfolio assessment, a central aspect of the model as outlined in Section 8.2.5, these processes are likely to be appropriate for the delivery of a number of portfolio based units. For teaching staff, the model utilised the following steps:

1. Define Intended Learning Outcomes
2. Construct Assessment Criteria
3. Develop Teaching and Learning Activities and Resources
4. Deliver Unit
5. Provide Feedback and Guidance
6. Assess Student Portfolios

The outcome of this work suggests that each of these activities is essential in delivering any unit using constructive alignment with portfolio assessment. Intended learning outcomes need to be stated using active verbs, which can be drawn from the SOLO Taxonomy. These outcomes then become the central focus of the unit with students aiming to demonstrate they have achieved these outcomes in their portfolios. Assessment criteria define performance objectives for each grade outcome, and indicate the kinds of evidence students need to construct. Teaching and learning activities and resources need to be developed, or existing activities and resources identified and used. Delivery of the unit will need to incorporate frequent formative feedback to ensure that student work is most likely to meet intended learning outcomes, and finally student portfolios must be assessed in order to determine final grades.

8.3 Approach in Relation to Prior Work

8.3.1 In Relation to Work on Constructive Alignment

Previous work on applying constructive alignment, as reported in the systematic literature review in Chapter 2, has predominantly seen the application of constructive alignment as simply the staff centred task of aligning teaching and learning activities with the unit's intended learning outcomes. This differs vastly from the view of constructive alignment presented in this thesis, where constructive alignment is seen as a much greater shift from a teaching-centred to a student-centred focus, with all aspects working together to guide and support students in the construction of their own knowledge – as captured in the principles outlined in Chapter 3.

The introductory programming units delivered prior to adopting the approach outlined in Chapter 4 had incorporated the core aspects of constructive alignment. The unit used delivery approaches consistent with constructive learning theories for computer science, and teaching and learning activities and assessment tasks were aligned (informally) to intended learning outcomes. By adopting the wider set of principles described in Chapter 3 the introductory programming units demonstrated improvements in both the teaching and learning environment and learning outcomes, as reported in Chapter 7.

Staff involved in teaching the units prior to the change to portfolio assessment reported assessment as a negative experience, as indicated in the reflections in Chapter 7. Marking assignments and exams identified, often for the first time, a large range of student misconceptions. The illusion that lectures had been effective in transferring knowledge to students disappeared, but too late to affect learning outcomes. This was further reinforced by the arbitrary weights of assignments and exams, which often resulted in cases where teaching staff felt that student results did not match the knowledge, or lack thereof, that students had demonstrated *qualitatively* in the final assessment tasks. It is not surprising that traditional exam-based forms of assessment often resulted in disappointments.

Reflections from teaching staff, reported in Chapter 7, indicate that these frustrations abated with the shift in approach. Final student assessment shifted from a negative experience, to became a positive and rewarding experience for teaching staff. Use of frequent formative feedback meant that students misconceptions were addressed often, allowing teaching staff to direct students and guide them to better understand unit concepts. "Assessments" were no longer final, so students were encouraged and rewarded for incorporating feedback they received, with each student receiving individual feedback based upon their current level of understanding. Assessment criteria provided a means for teaching staff to set expectations, while still providing opportunities for students to pursue their own interests and to use their imagination and creativity. In these ways final assessment did not hold any *negative* surprises. Portfolios provided an opportunity for students to "*show off*" what they had learnt. Where students had achieved Distinction and High Distinction results, these portfolios often went well beyond staff requirements, making assessment a rewarding experience for teaching staff and students as students demonstrated just how much they had been able to achieve.

This change in assessment had a flow on effect to the teaching and learning environment as a whole. Students were no longer *losing* marks in their assignments, but could use feedback to *improve* their learning outcomes – both in terms of final grade, and depth of understanding. Improvements in clarity of assessment criteria, and tools to support the formative feedback process, enabled closer collaboration helping staff to better guide students to the desired learning outcomes. The system was in harmony, with all aspects focusing on students demonstrating the unit's intended learning outcomes. The *simplicity* of constructive alignment with portfolio assessment, had resulted in a clear embodiment of Biggs' "web of consistency".

It is hard not to draw parallels between these different approaches to education and different software development life-cycle models. Traditional assessment approaches, based upon assignments and exam, can be likened to the Waterfall approach (Royce 1970). Teaching and learning activities are delivered in sequence, with little feedback from students. When feedback is provided on summative assignments it is often overlooked (Black & Wiliam 1998), as students focus on the grade they achieved rather than on opportunities for deeper learning. Any accumulated misunderstandings are then propagated to the final examination. In contrast, the iterative process outlined in Chapter 4 is more akin to processes in Agile software development. Students and staff interact frequently, enabling staff to guide students with the focus being clearly on their learning during the teaching period. Final summative assessment in student portfolios assess to what level students have been able to achieve stated unit outcomes, with higher grades indicating a demonstration of deeper understanding.

This discussion raises an important question, what is constructive alignment? Is it predominantly an issue of alignment performed by staff, as appears to be the majority view of the literature reviewed in Chapter 2, or is it predominantly an an application of constructivist learning theories with student alignment, as originally proposed by Biggs (1996) and outlined in this thesis? The answer is perhaps both, but more explicit terminology would help communicate which variant is being reported upon in future research work. The differing focus of the alignment, being performed either by staff or students, suggests that could be a means of distinguishing between the two. *Constructive Aligned Teaching* could be used to refer to teacher-centred approaches, where teaching staff aim to embed constructive learning theories in activities but retain traditional approaches to assessment. Similarly, *Constructive Aligned Learning* could be used to refer to use of constructive learning theories in both activities and assessment, where the student performs the alignment and presents a body of work to demonstrate their ability to achieve the intended learning outcomes.

8.3.2 In Relation to Work on Introductory Programming

Engaging Deep Approaches to Learning

This work presents a system for implementing constructive alignment for teaching introductory programming that encourages and rewards students for engaging in deep approaches to learning. The use of holistic, criterion referenced assessment, as presented in Chapter 4 and discussed in Chapter 7, in the example units resulted in grades that represented clearly distinct learning outcomes. Students who achieve a Pass grade had demonstrated the ability to meet at the unit intended learning outcomes to a minimal standard. A Credit grade indicated the student had completed all set tasks to a high standard, had demonstrated their progress to their tutor throughout the teaching period, and had constructed a portfolio that demonstrated a good coverage of all intended learning outcomes. In additional to meeting these criteria, students who achieved a Distinction or higher grade were able to demonstrate the application of the unit learning outcomes to the design and implementation of a program of their own creation.

Given these clearly distinct grade outcomes, it is encouraging to note that 82% of Computer Science students who enrolled in the unit managed to achieve a passing grade in the introductory programming unit reported in Iteration 8, with 59% achieving a grade of Credit or higher and 35% achieving a Distinction or higher grade. For these students the portfolio based approach appears to have worked well, with them actively engaging with the unit content.

With the primarily engineering focused students, the approach has been less successful. The results from Iteration 9 indicate that 67% of these students were able to achieve a Passing grade, with 28% achieving a Credit or higher result. However, these results are not moderated in any way and still indicate that these students were able to demonstrate all of the units intended learning outcomes in their portfolios. This indicates the significant role of motivation for any student centred approach to learning that incorporates flexible assessment. For these students, passing the unit appears to have been their primary goal. Once they achieved this goal they were satisfied and did not strive to achieve better results.

The different results for these two cohorts, when taught using the same approach, supports the claims of (Bruce et al. 2003) that students need to engage deep approaches to learning introductory programming, as surface approaches alone are unlikely to be sufficient. Designing the assessment criteria to require students to demonstrate deeper levels of understanding should mean that the system is sensitive to the approach students take to their learning, with those who engage in deeper learning achieving higher grades.

Motivation, therefore, remains key to addressing the challenges associated with teaching introductory programming raised by (McGettrick et al. 2005). Where students can be motivated to engage in deep learning the approach presented in this work provides a framework for supporting and directing student efforts toward their attainment of the unit's intended learning outcomes. For these students the support and freedom offered provides a means for them to engage meaningfully in developing a deep understanding of the associated programming concepts.

Programming Difficulties Faced

The results reported in Chapter 7 support the work of Winslow (1996) and Lahtinen et al. (2005) in indicating that students struggle more with programming concepts than with programming language syntax. Reflections from student portfolios indicated that for these students the primary challenges they faced in their learning were associated with general learning issues, or with issues associated with programming language concepts.

These findings support the more concept-based approach to teaching introductory programming presented in Chapter 4 and Chapter 5. By moving language syntax details to supporting resources, such as the programming text described in Section 6.3, the approach presented frees additional class time to focus on the programming concepts students need to grasp in order to more fully understand the programs they are creating. The results from the example units show that these classes can be successful once students engage appropriately with the learning activities.

Applying Constructive Learning Theories

The approach presented in Chapter 4, and example units described in Chapter 5, demonstrate how constructive learning theories can be embedded within an introductory programming unit. The approach presented incorporates a range of practices recommended by Ben-Ari (2001), including the use of a notional machine, interactive lecture demonstrations, code tracing tasks, and other activities to encourage the students to develop viable mental models of computing and program structure.

The success of the introductory programming unit indicates the effectiveness of these constructivist activities. It remains the case that "it is what the student does that counts" (Biggs & Tang 2007), but when students do engage these activities, these tasks provide them with the guidance they need to succeed in learning the associated concepts. Incorporating constructivist learning activities with frequent formative feedback does create a system capable of supporting a range of student capabilities and interests.

8.4 Challenges for Wider Adoption

8.4.1 Adopting Constructive Learning Theories

Constructivist learning theories are central to the principles and approach outlined in this thesis. This requires that educators adopt key aspects of constructivism as their theory-in-use, not just their espoused theory (Argyris 1976). A shift from a primarily objectivist view of education, to one centred on constructive learning theories, requires a significant conceptual change and is likely to be a large challenge to the wider adoption of this approach.

Objectivist views provide educators with a number of convenient truths, for example knowledge transfer is conceptually simple: get a number of people in a room and tell them what they need to know. Guiding students in the construction of their own knowledge may seem like a much greater challenge. Accepting the students central role in constructing their own knowledge, means rethinking old strategies, and looking for new ways to engage students with the material.

Constructivist learning theories require a move from teacher-centred to student-centred learning environments, which in turn requires educator to release some control of the learning environment. With teacher-centred activities, such as lectures, teaching staff have complete control of the content, pace, and method of delivery. With traditional lectures there is the possibility for well scripted lectures to be delivered by teaching staff who are not experts in the area. As more student-centred activities are adopted there is a need to incorporate greater input from students, enabling them to shape the environment to their needs. This requires teaching staff who are able to dynamically adjust delivery, ensuring the direction and focus of class activities are likely to result in productive learning for as many students as possible. Teaching staff need to be experts of the subject matter, as well as likely student misconceptions and strategies to address those misconceptions.

8.4.2 Removing Mark-based Assessment

Another significant challenge is the loss of “marks” as a means to motivate students. As noted in Chapter 7 (Iteration 2) one of the great challenges in implementing a Theory-Y based approach had been the anxiety of teaching staff over this perceived loss of control. This challenges the commonly held, Theory-X based, idea that students only work on tasks allocated marks. If you want students to *do* some task, then that task must be allocated marks. With this view, marks are the “carrot and stick” for educators wanting to motivate students. Abolishing the practice of allocating summative marks during the teaching period removes this as a form of motivation, and requires teaching staff to adopt Theory-Y based perspective of student motivation.

As with the shift to constructive learning theories, abandoning mid-term summative marks is a large challenge. The systematic literature review in Chapter 2 indicated that most reported applications of constructive alignment had adopted constructive learning theories for teaching and learning activities, but that assignments and exams remained the dominant assessment strategy. Moving away from the common strategy appears to be a larger challenge than accepting constructive learning theories. In many ways this is reflected in the comments of Sheard et al. (2013), who indicated that while academics feel exams are not an ideal means of assessing learning outcomes in introductory programming units, there was a general resistance to considering other approaches to assessment.

Part of this challenge is confronting an educator’s prior experiences, a result of the self-fulfilling nature of the Theory-X focus on marks. Students want to achieve a good result from the unit, and unit grades provide an externally visible measure of this performance. Marks are awarded for successfully completing tasks, and these marks directly relate to the grade students will achieve. Consider the case where laboratory tasks are not marked, but assignments are marked. In this situation students are encouraged by the marks to focus on the assignment work, despite the valuable learning that may have taken place if they did the lab tasks. Any staff member using combinations of mid-term marked and unmarked work will have experienced this focus, leading to the common perception that students only focus on tasks that have marks associated with them.

8.4.3 Holistic Assessment over Piece-by-Piece Assessment

The approach to portfolio assessment described in Chapter 4 aims to holistically evaluate student learning outcomes, the core of Principle 3. This goes against the common education practice of allocating percentages to each artefact a student submits, and determining their final result from a combination of weighted tasks. With the approach presented in this thesis, it is the intended learning outcomes together with the assessment criteria that determine final grades, not the artefacts themselves. Using the portfolio assessment scheme outlined in this thesis requires a change in thinking about how students are assessed overall.

Consider the portfolio assessment of the Research Project unit. This unit required students to undertake a research project, and to submit a portfolio that consisted of a research proposal and plan, research report, artefacts associated with an oral presentation, a learning summary report, and other artefacts relevant to the student's project. Given this list of required components, traditional approaches would seek to allocate percentages to each of these components. In contrast, applying the guidelines from Chapter 4 requires the definition of standards of achievement for the different grade outcomes, with the artefacts acting as the student's evidence of their ability to meet the intended learning outcomes.

While this may be understood by the academics involved in the teaching of the unit, this distinction also needs to be communicated to administrators and other academic staff who oversee unit accreditation. Units delivered using the approach recommended in this thesis are likely to stand out as being "different", with a greater chance of encountering resistance despite good pedagogical foundations. Overcoming these obstacles is likely to require support at a larger institutional level.

8.4.4 Perceived Workload Issues

Frequent formative feedback and portfolio assessment can have the appearance of requiring significant extra work for staff, and students, making it easy to dismiss as impractical. Teaching staff do need to be given sufficient time to engage with students, but as outlined in Section 8.1.3 this need not result in significant extra effort. Time taken to *mark* assignments now goes into providing formative feedback, in which staff can focus on providing a relevant level of detail to improve learning. Focusing on formative feedback should take less time than performing a "complete" and detailed assessment that is not necessarily relevant to learning. Similarly, time allocated to *mark* exams goes to *grading* portfolios, which can draw upon the evidence of student learning collected from the formative feedback process.

Efficiencies can the gained through reflective practice, with each iteration aiming to improve staff, and student, productivity by ensuring teaching and learning activities meet student needs.

8.4.5 Availability of Experienced Teaching Staff

Units are typically taught by a team of teaching staff, including lecturers and tutors for example. When the approach is first implemented at a given institute, there is not likely to be any teaching staff who have experience with this approach. This adds additional overhead for the first few iterations, as staff and students learn about the new teaching and learning environment.

8.4.6 Combined Issues

Constructive alignment with portfolio assessment, as outlined in this thesis, helps create a productive student-centred teaching and learning environment, but…it is easy to find a reason *not* to implement this approach. Change is hard, and the approach presented in Chapter 4 requires teaching staff adopt key constructive learning theories, a Theory-Y perspective of student motivation, and abandon the common use of marked assignments for holistic assessment of a body of work against set criteria. Then, once an individual academic has made this transition there are potential institutional barriers that need to be overcome to allow the approach to be implemented.

Once the approach is implemented it is also important to realise that any change of this nature is likely to require a number of iterations before the full benefits are realised. In many cases it is possible that the first iterations will encounter unexpected difficulties, as was the case with the implementation of this approach in Iterations 1 and 2 as discussed in Chapter 7. As such, once the approach is undertaken it will require a degree of persistence to see the implementation through to productive outcomes.

These tight constraints, and the general resistance to changes in assessment strategy, represent significant challenges for the wider adoption of this approach. However, as aspects of the approach slowly gain in popularity, such as the use of constructive learning theories, this reduces the amount of change necessary for others to consider this approach.

8.5 Transitioning to Portfolio Assessment

This section provides an example of how an existing unit could transition from using assignments and exams, to portfolio assessment with frequent formative feedback. This example assumes existing intended learning outcomes are sufficient, and that assessment tasks currently align with the unit's intended learning outcomes.

Consider a unit that assesses students using two essays and a final examination. In order to shift to portfolio assessment, with frequent formative feedback, these assessment tasks could be broken down into a number of smaller tasks, with students receiving feedback at each stage. It is likely that these larger assessment tasks included a number of smaller activities; tasks that staff *assumed* students would perform. Making these activities the unit's weekly tasks would ensure students performed the required activities, while also generating artefacts that staff can use to provide students with specific feedback.

These smaller tasks would form the basis of the Pass and Credit criteria for the unit. To balance workload, the scope or scale of these small tasks may need to be adjusted, with the expectation that students would perform the task to a high standard in order to achieve a passing grade. The two essays could become the required artefacts for the Pass and Credit criteria, or could potentially become the Distinction (relational) criteria with output from the smaller tasks making up the Pass and Credit grades.

Where an essay was still desired, subtasks could be set that require students to demonstrate understanding of relevant background reading material, to prepare outlines for the final essay, and possibly to perform a peer review of other student work. An example set of subtasks is shown in the following list.

1. Read background material, summarise, and communicate in the form of a blog entry on <*insert topic*>
2. Prepare a draft outline of essay on <*insert topic*>which incorporates references from background reading material
3. Prepare complete draft version of essay on <*insert topic*>
4. Write a peer review of two other student's work on <*insert topic*>using the template provided
5. Submit final version for assessment in portfolio, along with peer reviews and reflections on alterations made.

By completing these tasks, staff would have multiple opportunities to guide student learning, ensuring that the resulting essays demonstrate that each student had attained at least the Pass grade standard.

Frequent student interaction should also help to address potential staff workload issues, as staff aim to provide a small amount of specific feedback on each of the subtasks. Blog entries could be quickly scanned to see that students had captured critical aspects from each of the reading tasks, outlines could be quickly checked to ensure appropriate structure, while more detailed feedback would need to be provided on completed drafts. Each of these could be marked off in a task tracking system, like Doubtfire, when they had been completed to a satisfactory standard. Assessing the final portfolio would simply involve first checking that work had been marked off, and considering the changes students had made in response to peer feedback.

Tasks appropriate for Distinction and High Distinction would also need to be considered, and staff and student workloads balanced with the core tasks. Distinction tasks can be devised by teaching staff considering what they *really* want student to be able to do by the end of the unit. For example, in the programming units described in Chapter 5 this resulted in the Distinction criteria requiring students to demonstrate the ability to create a program that demonstrated an appropriate application of the unit concepts. This could, for example, involve developing mathematical models for units on mathematics, preparing financial reports in units on accounting, or explaining metabolic processes in units on biochemistry. Assessment criteria can then be defined to require students to perform these tasks in order to achieve Distinction and High Distinction grades.

With the essay assignments now broken down into a number of smaller tasks, the overall strategy for delivering the unit should consider the kinds of activities likely to be the most effective in ensuring students can successfully perform these tasks. Using this strategy, the design of the activities themselves need to support students as they complete these tasks, providing them with guidance on associated concepts, tools, techniques, and approaches. Focusing on constructive learning theories, and “what the student does” (Biggs & Tang 2007), should ensure the teaching and learning activities aim to guide students rather than convey knowledge.

The examination would no longer be necessary as a means of assessing learning outcomes. Instead the exam could be converted to a hurdle test, as was the case in the introductory programming units, and used to verify that students had engaged in the work themselves. Where units had small student numbers the test may not be necessary as staff can closely monitor student work across the teaching period.

Having altered assessment, delivery can follow the activities outlined in Chapter 4.

The wording of intended learning outcomes would need to be checked to ensure they will be appropriately understood by students, which could incorporate the guidelines from Section 4.2.2. Assessment criteria would need to be defined using the guidelines from Section 4.2.3. The teaching and learning activities would to help students achieve the intended learning outcomes, as planned by staff with the core tasks in the assessment work. Iterative delivery would permit students to receive feedback on their progress, while also providing staff with a clear picture of student progress and potential areas of misunderstandings. Finally, student work can be captured in their portfolios and assessed against the criteria set out at the start of the unit.

While initial iterations are likely to encounter some issues, the incorporation of the principles from Chapter 3 should eventually result in a similar positive, student-centred, teaching and learning environment as reported with the introductory programming units in Chapter 7. The use of portfolio assessment is central to this success, as it enables staff and students to work together in helping students build appropriate learning outcomes.

8.6 Discussion Summary

This chapter provided a discussion of the approach presented in this thesis, and its underlying principles. The general applicability of the approach was considered, indicating that it should be applicable beyond the context of introductory programming to a wider range of educational contexts. The importance of each of the principles was outlined, along with a discussion of the likely impact of ignoring each principle on the resulting teaching and learning environment, indicating that key aspects from the principles are required for benefits to be realised. Similarly, the critical importance of the portfolio assessment and constructive learning theories in guiding the delivery approach were also discussed.

Challenges facing wider adoption are numerous, as the delivery and assessment approach differ from education norms. Where these challenges can be overcome, an effective teaching and learning environment can be achieved, one that encourages and rewards students for focusing on developing deep understanding. In this environment, students engage their imagination and creativity in meeting unit intended learning outcomes, working collaboratively with staff who provide students with guidance and feedback. The results, as Biggs (1996) indicated, are portfolios that consistently impress staff with the quality of the learning outcomes students have been able to achieve.

9

Conclusion and Future Work

Constructive alignment has been widely accepted as a valuable framework for improving the quality of teaching and learning in higher education. Chapter 2 of this thesis provided a comprehensive reviewed of prior work on applications of constructive alignment, finding that most applications adopted constructivist approaches to content delivery, but retained traditional approaches to assessment based around the use of assignments and exams. This review indicated that none of the reviewed papers had attempted to recreate the "web of consistency" reported in Biggs original work.

Chapter 3 presented twelve principles, nine related to *how* we teach and three related to *what* we teach. These principles became the foundation for the approach to constructive alignment presented in Chapter 4, which applied constructive learning theories to unit delivery and assessment. Chapters 5 and 6 demonstrated the application of the approach from Chapter 4 in the creation of two introductory programming units, and tools and resources to aid in their delivery. These chapters also demonstrated how the principles from Chapter 3 embedded within the approach, were realised in the teaching and learning activities and resources created.

The iterations from the action research projects were presented in Chapter 7, along with analysis of issues students faced, and analysis of student progress. Analysis shows the development of the approach, and its underlying principles, as a result of the reflective practice embedded within the iterative action research method used. Increasing student numbers in later iterations also helps demonstrate the applicability of the model presented to the teaching of units involving hundreds of students.

This thesis has demonstrated how the "web of consistency" associated with Biggs early work can be recreated through the application of the principles proposed in Chapter 3. As discussed in Chapter 8, this approach has enabled teaching staff to successfully deliver a range of introductory programming units. The results demonstrated the effective use of all twelve principles, illustrating how they work together to create a positive, student-centred, teaching and learning environment in which students are rewarded for demonstrating depth of understanding, and encouraged to use their imagination and creativity.

As outlined in the introduction, the key contributions of this thesis are:

- A systematic literature review of applications of constructive alignment, examining the areas in which this has been applied and strategies used for delivery and assessment.
- A set of guiding principles for the development and delivery of units that aim recreate the "web of consistency" evident in Biggs' early work on constructive alignment.
- An approach to constructive alignment, developed from the guiding principles, with strong links to constructivism in both teaching and learning activities and approach to assessment.
- An online task tracking tool to help students monitor their progress on tasks designed to provide them with feedback in units that delay summative assessment until after the teaching period.
- Evaluation of the resulting teaching and learning context and tools, that demonstrates how effective assessment criteria can be used to quickly evaluate student learning outcomes.
- An introductory programming curriculum designed using the principles of constructive alignment.
- An approach to teaching introductory programming, that embodies the identified principles, with guidelines for implementing this approach.
- Example implementations of the approach presented, demonstrating its application to teaching a number of introductory programming units.
- A concept-based approach to introductory programming, together with supporting resources including a concept-focused text, a game development framework, and a range of video podcasts.

This work is mearly a beginning and it is hoped that it will continue to explore the opportunities discussed in Section 9.1 in future work.

9.1 Future Work

Having developed the principles and an approach across a number of iterations, there are now opportunities to further analyse the portfolios collected, to examine ongoing changes in the activities used and their impact on student results, and to explore the wider application of the approach presented.

The structure literature review reported in Chapter 2 identified a range of work on constructive alignment. In collating the data for this review it was noticed that there appeared to be little referencing between the analysed papers, however this reference data was not collected and analysed. A further analysis, and reporting, of these connections may help provide a richer understanding of the field.

Work to date has collected hundreds of student portfolios, each of which captures an individuals learning outcomes from their engagement with the teaching and learning environment. While some analysis has been performed on these portfolios there are many other opportunities that can now be considered. Student reflections, their programs, reports, custom projects, and research reports all provide different opportunities to explore aspects of the teaching and learning environment from student perspectives.

While student portfolios have demonstrated their ability to create programs, it would be very interesting to repeat an experiment similar to that conducted by McCracken et al. (2001). The mathematical nature of the exercises should be changed to better reflect the more general nature of the programming units, but otherwise it would be interesting to evaluate student ability to implement a set specification. Where students were unable to get the program working, it would be interesting to extend the experiment to determine the likely cause of the problem, and the extent of help needed for the student to succeed in implementing the program. This would help improve understanding of the limitation of students at the end of their first year of programming, and likely assistance they would need in implementing programs on their own.

Continued delivery of the introductory programming units discussed in Chapter 5 also offers a range of opportunities to examine the approach in more depth. Consistency of portfolio assessment could be examined to determine if there is variation between grades awarded by different teaching staff. A more formal evaluation of staff workloads could be carried out by examining the time spent providing feedback, assessing portfolios, and supporting students during the teaching period. Results could then be compared with the amount of time allocated to these tasks by the educational institution to determine if workload is comparable to other approaches.

It would also be very interesting to study students within the environment, examining differences between successful and unsuccessful students. This could help identify reasons why students fail in what is considered to be a highly supportive environment, and suggest strategies that could be applied to help students succeed. It is expected that intrinsic motivation plays a significant role in this.

Longitudinal studies could examine how students from portfolio units progressed through later units. Interviewing students at the end of their degree programme could provide a deeper understanding of what is working, and what can be improved with the proposed approach. This could examine aspects such as:

- Do the introductory programming units help students succeed at later programming units?
- Do students feel they learnt general skills they could apply to a wider range of units, or was the learning entirely focused on programming knowledge?
- How do portfolio assessed units relate to other units, from the students perspective?
- What differences do student identify between the different delivery strategies?
- Would students like to see more portfolio units?

In the broader context, future work is needed to trial the approach in other fields and educational institutions. It would be interesting to see how well the approach can be adapted to non-technical units, both within Information Technology and beyond. Where the principles from Chapter 3 can be adopted, it is believed that similar positive learning outcomes will be achieved.

While Section 8.4 outlined some of the challenges facing wider adoption of this approach, a more systematic analysis of peoples responses to the approach would also be enlightening. The approach is considered to be genuinely better than alternatives, but discussions with other teaching staff have been met with general resistance. Better understanding people's doubts about the approach could help adapt strategies to more effectively encourage people to consider its use.

If wider adoption has been achieved, it would be interesting to compare learning outcomes from units using constructive alignment with portfolio assessment, from those using more traditional forms of assessment. By examining a range of units, in different contexts, it should be possible to gain some comparative statistics to support the qualitative findings from this work.

References

Abran, A., Moore, J. W., Dupuis, R., Dupuis, R. L. & Tripp, L. L. (2001), *Guide to the Software Engineering Body of Knowledge - SWEBOK*, IEEE.

ACM/IEEE-CS Joint Task Force (2001), Computing curricula 2001, Technical report, New York, NY, USA.

ACM/IEEE-CS Joint Task Force (2008), Computer science curriculum 2008: An interim revision of CS 2001, Technical report.

ACM/IEEE-CS Joint Task Force (2012), Computer science curricula 2013: Ironman draft (version 0.8), Technical report.

Ala-Mutka, K. M. (2007), 'A survey of automated assessment approaches for programming assignments', *Computer Science Education* **15**(2), 83–102.

Allan, J. (1996), 'Learning outcomes in higher education', *Studies in Higher Education* **21**(1), March 1996.

Anderson, J. R., Reder, L. M., Simon, H. A., Ericsson, K. A. & Glaser, R. (1998), 'Radical constructivism and cognitive psychology', *Brookings papers on education policy* (1), 227–278.

Andrews, S. (2011), Aligning the teaching of FCA with existing module learning outcomes, *in* 'Conceptual Structures for Discovering Knowledge, 19th International Conference on Conceptual Structures', Lecture Notes in Computer Science, Springer, pp. 394–401.

Anik, Z. & Baykoç, O. F. (2011), 'Comparison of the most popular object-oriented software languages and criterions for introductory programming courses with analytic network process: A pilot study', *Computer Applications in Engineering Education* **19**(1), 89–96.

Argyris, C. (1976), 'Theories of action that inhibit individual learning', *American Psychologist* **31**(9), 638:654.

Armarego, J. (2009), Constructive Alignment in SE education: aligning to what?, *in* H. Ellis, S. Demurjian & Naveda, eds, 'Software Engineering: effective teaching and learning approaches and practices', ACM, pp. 15–37.

Astrachan, O., Bruce, K., Koffman, E., Kölling, M. & Reges, S. (2005), 'Resolved: objects early has failed', *ACM SIGCSE Bulletin* **37**(1), 451–452.

Atkinson, C. (2007), *Beyond Bullets Points: using Microsoft® Office PowerPoint® 2007 to create presentations that inform, motivate, and inspire*, Microsoft Press.

Australian Qualifications Framework Council (2013), *Australian Qualifications Framework*, 2nd edn, Australian Qualifications Framework Council, South Australia.

Baird, L. L. (1985), 'Do grades and tests predict adult accomplishment?', *Research in Higher Education* **23**(1), 3–85.

Baker, J. W. (2000), The classroom flip. using web course management tools to become the guide on the side, *in* '11th international Conference on College Teaching and Learning, Jacksonville, FL'.

Bayliss, J. D. & Strout, S. (2006), 'Games as a "flavor" of CS1', *SIGCSE Bulletin* **38**(1), 500–504.

Beck, K. (2000), *Extreme Programming Explained: Embrace change*, Addison-Wesley.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A. & Jeffries, R. (2001), 'Manifesto for agile software development', *The Agile Alliance* pp. 2002–04.

Becker, K. (2002), 'Back to Pascal: retro but not backwards', *Journal of Computing in Small Colleges* **18**, 17–27.

Ben-Ari, M. (1998), Constructivism in computer science education, *in* 'Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education', ACM, New York, NY, USA, pp. 257–261.

Ben-Ari, M. (2001), 'Constructivism in computer science education', *Journal of Computers in Mathematics and Science Teaching* **20**(1), 45–73.

Bennedsen, J. & Caspersen, M. E. (2004), 'Programming in context: a model-first approach to CS1', *SIGCSE Bulletin* **36**, 477–481.

Bennedsen, J. & Caspersen, M. E. (2006), Assessing process and product - a practical lab exam for an introductory programming course, *in* 'Proceedings of the 36th Annual Frontiers in Education Conference', IEEE, pp. 16–21.

Biggs, J. (1996), 'Enhancing teaching through constructive alignment', *Higher Education* **32**, 347–364.

Biggs, J. (1999), 'What the student does', *Higher Education Research and Development* **18**(1), 57–75.

Biggs, J. B. (1987), *Student Approaches to Learning and Studying. Research Monograph*, Australian Council for Educational Research Ltd., Radford House, Frederick St., Hawthorn 3122, Australia.

Biggs, J. B. & Collis, K. F. (1982), *Evaluating the Quality of Learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*, Academic Press New York.

Biggs, J. B. & Tang, C. (2007), *Teaching for Quality Learning at University*, 3rd edn, Open University Press.

Biggs, J., Kember, D. & Leung, D. Y. P. (2001), 'The revised two-factor study process questionnaire: R-SPQ-2F', *British Journal of Educational Psychology* **71**(1), 133–149.

Biggs, J. & Tang, C. (1997), 'Assessment by portfolio: Constructing learning and designing teaching', *Research and Development in Higher Education* pp. 79–87.

Bishop, W. & Freeman, G. (2006), The use of C# as a First Programming Language, *in* 'Proceedings of the 2006 International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS '06)', pp. 97–103.

Black, P. & Wiliam, D. (1998), 'Assessment and classroom learning', *Assessment in Education* **5**(1), 7–74.

Bloom, B. S. (1969), 'Some theoretical issues relating to educational evaluation', *Educational Evaluation: New Roles, New Means (National Society for the Study of Evaluation Yearbook, Part II)* **68**, 26–50.

Börstler, J. & Schulte, C. (2005), 'Teaching object oriented modelling with crc cards and roleplaying games', *Proceedings IFIP World Conference on Computers in Education* .

Böszörményi, L. (1998), 'Why Java is not my favorite first-course language', *Software-Concepts & Tools* **19**(3), 141–145.

Boyer, E. L. (1990), *Scholarship Reconsidered: Priorities of the Professoriate*, Carnegie Foundation for the Advancement of Teaching, Princeton, N.J.

Brabrand, C. (2008), 'Constructive alignment for teaching model-based design for concurrency', *Transactions on Petri Nets and Other Models of Concurrency I* pp. 1–18.

Brabrand, C. & Dahl, B. (2007), Constructive alignment and the SOLO taxonomy: a comparative study of university competences in computer science vs. mathematics, *in* 'Proc. Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007), Koli National Park, Finland. CRPIT', Vol. 88, pp. 3–17.

Brabrand, C. & Dahl, B. (2009), 'Using the SOLO taxonomy to analyze competence progression of university science curricula', *Higher Education* **58**(4), 531–549.

Braun, V. & Clarke, V. (2006), 'Using thematic analysis in psychology', *Qualitative Research in Psychology* **3**(2), 77–101.

Braz, L. M. (1990), Visual syntax diagrams for programming language statements, *in* 'Proceedings of the 8th Annual International Conference on Systems Documentation', SIGDOC '90, ACM, New York, NY, USA, pp. 23–27.

Brilliant, S. S. & Wiseman, T. R. (1996), 'The first programming paradigm and language dilemma', *SIGCSE Bulletin* **28**, 338–342.

Brown, N., Smyth, K. & Mainka, C. (2006), Looking for evidence of deep learning in constructively aligned online discussions, *in* 'Networked Learning Conference', pp. 10–12.

Brown, S. (2004), 'Assessment for learning', *Learning and Teaching in Higher Education* **1**(1), 81–89.

Bruce, C. S., McMahon, C. A., Buckingham, L. I., Hynd, J. R. & Roggenkamp, M. G. (2003), 'Ways of experiencing the act of learning to program: A phenomenographic study of introductory programming students at university', *Journal of Information Technology Education* **3**, 143–160.

Bruner, J. S. (1961), 'The act of discovery', *Harvard Educational Review* **31**, 21–32.

Cain, A. (2013*a*), Developing assessment criteria for portfolio assessed introductory programming, *in* 'Proceedings of the 2nd IEEE International Conference on Teaching, Assessment and Learning for Engineering', IEEE, pp. 55–60.

Cain, A. (2013*b*), *Programming Arcana*, Swinburne University of Technology.
URL: *http://mercury.it.swin.edu.au/acain/programming-arcana.pdf*

Cain, A. & Woodward, C. J. (2012), Toward constructive alignment with portfolio assessment for introductory programming, *in* 'Proceedings of the first IEEE International Conference on Teaching, Assessment and Learning for Engineering', IEEE, pp. 345–350.

Cain, A. & Woodward, C. J. (2013), Examining student reflections from a constructively aligned introductory programming unit, *in* 'Proceedings of the 15th Australasian Computer Education Conference', Vol. 136, pp. 127–136.

Cain, A., Woodward, C. J. & Pace, S. (2013), Examining student progress in portfolio assessed introductory programming, *in* 'Proceedings of the 2nd IEEE International Conference on Teaching, Assessment and Learning for Engineering', IEEE, pp. 67–72.

Cassel, L., Clements, A., Davies, G., Guzdial, M., McCauley, R., McGettrick, A., Sloan, R., Snyder, L., Tymann, P. & Weide, B. (2008), 'Computer science curriculum 2008: An interim revision of CS 2001'.

Chansarkar, B. A. & Raut-Roy, U. (1987), 'Student performance under different assessment situations', *Assessment and evaluation in Higher Education* **12**(2), 115–122.

Chickering, A. W., Gamson, Z. F. & Poulsen, S. J. (1987), 'Seven principles for good practice in undergraduate education', *American Association for Higher Education Bulletin* pp. 3–7.

Cliburn, D. (2006), The effectiveness of games as assignments in an introductory programming course, *in* 'Frontiers in Education, 36th Annual Conference', IEEE, pp. 6–10.

Cobb, P. (1994), 'Where is the mind? constructivist and sociocultural perspectives on mathematical development', *Educational Researcher* **23**(7), 13–20.

Cockburn, A. (2002), *Agile Software Development*, Addison-Wesley Professional.

Coffield, F., Moseley, D., Hall, E. & Ecclestone, K. (2004), 'Learning styles and pedagogy in post-16 learning: A systematic and critical review'.

Cohen, S. A. (1987), 'Instructional alignment: Searching for a magic bullet', *Educational Researcher* **16**(8), 16–20.

Conway, M. A., Cohen, G. & Stanhope, N. (1992), 'Comment: Why is it that university grades do not predict very-long-term retention?", *Journal of Experimental Psychology: General* **121**(3), 382–384.

Cooper, S., Dann, W. & Pausch, R. (2003), Teaching objects-first in introductory computer science, *in* 'Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education', SIGCSE '03, ACM, New York, NY, USA, pp. 191–195.

Creswell, J. W. (2008), *Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research*, Pearson/Merrill Prentice Hall, Upper Saddle River, N.J.

D3.js (2013), 'D3.js: Data-driven documents'. Accessed 2013-07-09.
URL: *http://d3js.org*

Davey, A. K. & Bond, C. (2002), 'Learning about clinical pharmacokinetics: A case study', *Pharmacy Education* **2**(2), 83–92.

de Raadt, M., Hamilton, M., Lister, R., Tutty, J., Baker, B., Box, I., Cutts, Q., Fincher, S., Hamer, J. & Haden, P. (2005), Approaches to learning in computer programming students and their effect on success, *in* 'Proceedings of the 28th HERDSA Annual Conference: Higher Eduation in a Changing World (HERDSA 2005)', Higher Education Research and Development Society of Australasia (HERDSA), pp. 407–414.

Denning, P. J. (1989), 'A debate on teaching computing science', *Communications of the ACM* **32**, 1397–1414.

Deterding, S., Dixon, D., Khaled, R. & Nacke, L. (2011), From game design elements to gamefulness: defining gamification, *in* 'Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments', ACM, pp. 9–15.

Dewey, J. (1933), *How We Think: A Restatement of the Relation of Reflective Thinking to the Educational Process*, D. C. Heath, Boston.

Dijkstra, E. W. (1989), 'On the cruelty of really teaching computing science', *Communications of the ACM* **32**(12), 1398–1404.

Donnison, S. & Edwards, D. (2011), Re-designing a first year teacher education community service-learning subject using constructive alignment, *in* 'ATEA Conference 2011: Valuing Teacher Education: Policy, Perspectives and Partnerships'.

Douce, C., Livingstone, D. & Orwell, J. (2005), 'Automatic test-based assessment of programming: A review', *Journal of Educational Resources in Computing* **5**.

Driessen, E., van der Vleuten, C., Schuwirth, L., Van Tartwijk, J. & Vermunt, J. (2005), 'The use of qualitative research criteria for portfolio assessment as an alternative to reliability evaluation: a case study', *Medical Education* **39**(2), 214–220.

DuBoulay, B. (1986), 'Some difficulties of learning to program', *Journal of Educational Computing Research* **2**(1), 57–73.

Duffy, T. M. & Cunningham, D. J. (1996), Constructivism: Implications for the design and delivery of instruction, *in* D. J. Jonassen, ed., 'Handbook of research for educational communications and technology', pp. 170–198.

Duffy, T. M. & Jonassen, D. H. (1992), *Constructivism and the Technology of Instruction: A Conversation*, Lawrence Erlbaum.

Eckerdal, A., Thuné, M. & Berglund, A. (2005), What does it take to learn 'programming thinking'?, *in* 'Proceedings of the First International Workshop on Computing Education Research', ICER '05, ACM, New York, NY, USA, pp. 135–142.

Ehlert, A. & Schulte, C. (2009), Empirical comparison of objects-first and objects-later, *in* 'Proceedings of the Fifth International Workshop on Computing Education Research', ICER '09, ACM, pp. 15–26.

Ehlert, A. & Schulte, C. (2010), Comparison of OOP first and OOP later: first results regarding the role of comfort level, *in* 'Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education', ACM, pp. 108–112.

Entwistle, N. J. . J. (1991), 'Approaches to learning and perceptions of the learning environment', *Higher Education* **22**(3), 201–204.

Entwistle, N. & Tait, H. (1990), 'Approaches to learning, evaluations of teaching, and preferences for contrasting academic environments', *Higher Education* **19**(2), 169–194.

Farrell, T. (2007), *Reflective Language Teaching: From Research to Practice*, Continuum Press, London.

Farrell, T. (2008), 'Reflective practice in the professional development of teachers of adult english language learners', *CAELA Network* .

Feldgen, M. & Clua, O. (2004), Games as a motivation for freshman students to learn programming, *in* '34th Annual Frontiers in Education', IEEE, pp. 1079–1084.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2004), 'The TeachScheme! project: Computing and programming for every student', *Computer Science Education* **14**(1), 55–77.

Field, J. (2006), *Lifelong Learning and the New Educational Order*, Trentham Books Limited.

Forte, A. & Guzdial, M. (2005), 'Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses', *Education, IEEE Transactions on* **48**(2), 248–253.

Foundation, T. S. (2011), *SFIA 5 Framework Reference: Skills Defined in Categories and Subcategories*.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2001), *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, Springer.

Gaspar, A. & Langevin, S. (2007), Restoring coding with intention in introductory programming courses, *in* 'Proceedings of the 8th ACM SIGITE Conference on Information Technology Education', ACM, pp. 91–98.

Gaspar, A. & Langevin, S. (2012), 'An experience report on improving constructive alignment in an introduction to programming', *Journal of Computing Sciences in Colleges* **28**(2), 132–140.

Gibbs, G. & Lucas, L. (1997), 'Coursework assessment, class size and student performance: 1984-94', *Journal of further and higher education* **21**(2), 183–192.

Gibbs, G. & Simpson, C. (2004), 'Conditions under which assessment supports students learning', *Learning and Teaching in Higher Education* **1**(1), 3–31.

Glasersfeld, E. (1989), 'Cognition, construction of knowledge, and teaching', *Synthese* **80**, 121–140.

Goldman, K. J. (2004), 'A concepts-first introduction to computer science', *SIGCSE Bulletin* **36**, 432–436.

Green, T. R. G. (2000), Instructions and descriptions: some cognitive aspects of programming and similar activities, *in* 'Proceedings of the Working Conference on Advanced Visual Interfaces', AVI '00, ACM, New York, NY, USA, pp. 21–28.

Green, T. R. G. & Petre, M. (1996), 'Usability analysis of visual programming environments: A 'cognitive dimensions' framework', *Journal of Visual Languages and Computing* **7**(2), 131–174.

Gregor, S., von Konsky, B. R., Hart, R. & Wilson, D. (2008), *The ICT profession and the ICT body of knowledge (Version 5)*, Australian Computer Society, Sydney, Australia.

Gries, D. (1974), 'What should we teach in an introductory programming course?', *SIGCSE Bulletin* **6**, 81–89.

Gross, P. & Powers, K. (2005), Evaluating assessments of novice programming environments, *in* 'Proceedings of the First International Workshop on Computing Education Research', ICER '05, ACM, New York, NY, USA, pp. 99–110.

Gupta, D. (2004), 'What is a good first programming language?', *Crossroads* **10**(4), 7–7.

Guzdial, M. (2003), A media computation course for non-majors, *in* 'ACM SIGCSE Bulletin', Vol. 35, ACM, pp. 104–108.

Guzdial, M. & Forte, A. (2005), Design process for a non-majors computing course, *in* 'Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education', ACM, pp. 361–365.

Guzdial, M. & Soloway, E. (2002), 'Teaching the nintendo generation to program', *Communications of the ACM* **45**(4), 17–21.

Haigh, M. (2013), 'Writing successfully for the journal of geography in higher education', *Journal of Geography in Higher Education* **37**(1), 117–135.

Hartfield, P. J. (2010), 'Reinforcing constructivist teaching in advanced level biochemistry through the introduction of case-based learning activities', *Journal of Learning Design* **3**(3), 12.

Hedges, M. R. & Pacheco, G. A. (2012), Constructive alignment, engagement and exam performance: It's (still) ability that matters, *in* 'New Zealand Association of Economists Conference'.

Henderson, F. (2006), Enriching the learning for offshore students in a 1st year management subject, *in* 'Conference Proceedings of the 17th ISANA International Education Conference', ISANA International Education Association.

Hendry, G. D., Frommer, M. & Walker, R. A. (1999), 'Constructivism and problem-based learning', *Journal of Further and Higher Education* **23**(3), 369–371.

Hill, R. (2009), '"Why should i do this?" Making the information systems curriculum relevant to strategic learners', *ITALICS: Innovations in Teaching and Learning in Information and Computer Sciences* **8**(2), 14–23.

Hoare, C. A. R. (1969), 'An axiomatic basis for computer programming', *Communications of the ACM* **12**, 576–580.

Hoc, J. M. & Nguyen-Xuan, A. (1990), 'Language semantics, mental models and analogy', *Psychology of Programming* **10**, 139–156.

Hoddinott, J. (2000), Biggs constructive alignment: Evaluation of a pedagogical model applied to a web course, *in* 'Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2000', AACE, Chesapeake, VA, pp. 1666–1667.

Houghton, W. (2004), *Engineering Subject Centre Guide: Learning and Teaching Theory for Engineering Academics*, Higher Education Academy Engineering Subject Centre, Loughborough University.

Howe, E., Thornton, M. & Weide, B. W. (2004), Components-first approaches to CS1/CS2: principles and practice, *in* 'Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education', SIGCSE '04, ACM, New York, NY, USA, pp. 291–295.

Howell, K. (2003), 'First computer languages', *Journal of Computing Sciences in Colleges* **18**(4), 317–331.

Israel, N., Pitman, M. & Greyling, M. (2007), 'Engaging critical thinking: Lessons from the RDA tutorials and projects', *South African Journal of Psychology* **37**(2), 375–382.

James, D. & Fleming, S. (2004), 'Agreement in student performance in assessment', *Learning and Teaching in Higher Education* **1**(1), 32–50.

Jenkins, T. (2001), 'The motivation of students of programming', *SIGCSE Bulletin* **33**, 53–56.

Jenkins, T. (2002), On the difficulty of learning to program, *in* 'Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences', pp. 53–58.

Jonassen, D. H. (1991*a*), 'Context is everything', *Educational Technology* **31**(6), 35–37.

Jonassen, D. H. (1991*b*), 'Objectivism versus constructivism: Do we need a new philosophical paradigm?', *Educational Technology Research and Development* **39**(3), 5–14.

Jonassen, D. H. (1992), 'Evaluating constructivistic learning', *Constructivism and the Technology of Instruction: A Conversation* pp. 137–148.

Jones, M. (2010), An extended case study on the introductory teaching of programming, PhD thesis.

Jones, P. (2007), When a wiki is the way: Exploring the use of a wiki in a constructively aligned learning design, *in* 'ICT: Providing choices for learners and learning. Proceedings ASCILITE Singapore 2007', Centre for Educational Development, Nanyang Technological University Singapore.

Joseph, S. & Juwah, C. (2012), 'Using constructive alignment theory to develop nursing skills curricula', *Nurse Education in Practice* **12**(1), 52 – 59.

jQuery (2013), 'jQuery: write less, do more'. Accessed 2013-07-09.
URL: *http://jquery.com/*

Junit (n.d.).

Kelleher, C. & Pausch, R. (2005), 'Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers', *ACM Computing Surveys* **37**, 83–137.

Kember, D. & Leung, D. Y. P. (2007), 'Characterising a teaching and learning environment conducive to making demands on students while not making their workload excessive', *Studies in Higher Education* **31**(2), 185–198.

Kenney, J. L. (2012), 'Getting results: small changes, big cohorts and technology', *Higher Education Research and Development* pp. 873–889.

Kirschner, P. A., Sweller, J. & Clark, R. E. (2006), 'Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching', *Educational psychologist* **41**(2), 75–86.

Kitchenham, B. (2007), Guidelines for performing systematic literature reviews in software engineering, Technical report, Keele University.

Klem, A. M. & Connell, J. P. (2004), 'Relationships matter: Linking teacher support to student engagement and achievement', *Journal of School Health* **74**(7), 262–273.

Kniveton, B. H. (1996), 'Student perceptions of assessment methods', *Assessment & Evaluation in Higher Education* **21**(3), 229–237.

Koffman, E. B. (1988*a*), *Pascal: Problem Solving and Program Design*, Addison-Wesley Longman Publishing Co., Inc.

Koffman, E. B. (1988*b*), 'The case for Modula-2 in CS1 and CS2', *SIGCSE Bulletin* **20**, 49–53.

Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. (2003), 'The BlueJ System and its Pedagogy', *Computer Science Education* **13**(4), 249–268.

Kuhn, K.-A. L. . A. L. (2009), 'Curriculum alignment: Exploring student perception of learning achievement measures', *International Journal of Teaching and Learning in Higher Education* **21**(3), 351–361.

Lage, M. J. & Platt, G. (2000), 'The internet and the inverted classroom', *The Journal of Economic Education* **31**(1), 11–11.

Lahtinen, E., Ala-Mutka, K. & Järvinen, H. M. (2005), 'A study of the difficulties of novice programmers', *ACM SIGCSE Bulletin* **37**(3), 14–18.

Lakoff, G. (1987), 'Women, fire, and dangerous things: What categories reveal about the mind'.

Lethbridge, T., LeBlanc Jr, R., Sobel, A. & Hilburn, T. (2006), 'SE2004: Recommendations for undergraduate software engineering curricula', *IEEE SOFTWARE* pp. 19–25.

Leutenegger, S. & Edgington, J. (2007), 'A games first approach to teaching introductory programming', *ACM SIGCSE Bulletin* **39**(1), 115–118.

Likert, R. (1932), 'A technique for the measurement of attitudes', *Archives of psychology* .

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B. & Thomas, L. (2004), 'A multinational study of reading and tracing skills in novice programmers', *SIGCSE Bulletin* **36**(4), 119–150.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C. & Whalley, J. L. (2006), Research perspectives on the objects-early debate, *in* 'Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education', ITiCSE-WGR '06, ACM, New York, NY, USA, pp. 146–165.

Lizzio, A., Wilson, K. & Simons, R. (2002), 'University students' perceptions of the learning environment and academic outcomes: Implications for theory and practice', *Studies in Higher Education* **27**(1), 27–52.

Maloney, J., Resnick, M., Rusk, N., Silverman, B. & Eastmond, E. (2010), 'The Scratch Programming Language and Environment', *Transactions on Computing Education* **10**(4), 16:1–16:15.

Mannila, L. & De Raadt, M. (2006), An objective comparison of languages for teaching introductory programming, *in* 'Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006', ACM, pp. 32–37.

Mannila, L., Peltomaki, M. & Salakoski, T. (2006), 'What About a Simple Language? Analyzing the Difficulties in Learning to Program', *Computer Science Education* **16**(3), 17.

Manns, M. L. & Nelson, J. (1993), An exploration of schema development in procedure-oriented programmers learning object-oriented technology, *in* 'Proceedings of the International Conference on Information Systems', SOCIETY FOR INFORMATION MANAGEMENT, pp. 385–385.

Marion, W. (1999), 'CS1: what should we be teaching?', *SIGCSE Bulletin* **31**, 35–38.

Markwell, J. (2004), 'The human side of science education: Using McGregor's theory Y as a framework for improving student motivation', *Biochemistry and Molecular Biology Education* **32**(5), 323–325.

Martin, E., Prosser, M., Trigwell, K., Ramsden, P. & Benjamin, J. (2000), 'What university teachers teach and how they teach it', *Instructional Science* **28**, 387–412.

Martin, R. C. (2003), *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall PTR.

Marton, F. & Booth, S. (1997), *Learning and awareness*, Lawrence Erlbaum.

Marton, F. & Säljö, R. (1976*a*), 'On qualitative differences in learning- II outcome as a function of the learner's conception of the task', *British Journal of Educational Psychology* **46**(2), 115–127.

Marton, F. & Säljö, R. (1976*b*), 'On qualitative differences in learning: I - outcome and process', *British Journal of Educational Psychology* **46**(1), 4–11.

Marton, F. & Säljö, R. (2005), Approaches to learning, *in* F. Marton, D. Hounsell & N. Entwistle, eds, 'The Experience of Learning: Implications for teaching and studying in higher education', 3rd internet edn, Edinburgh: University of Edinburgh, Centre for Teaching, Learning and Assessment, pp. 39–58.

Mason, R. & Cooper, G. (2013), Distractions in programming environments, *in* 'Proceedings Fifteenth Australasian Computing Education Conference (ACE2013)', Adelaide, SA.

Mayer, R. E. (2004), 'Should there be a three-strikes rule against pure discovery learning?', *American Psychologist* **59**(1), 14.

Mayer, R. E. (2005), *The Cambridge Handbook of Multimedia Learning*, Cambridge University Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001), 'A multi-national, multi-institutional study of assessment of programming skills of first-year CS students', *SIGCSE Bulletin* **33**(4), 125–180.

McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G. & Mander, K. (2005), 'Grand challenges in computing: Education - a summary', *The Computer Journal* **48**(1), 42–48.

McGregor, D. (1960), *The human side of enterprise*, McGraw-Hill.

Meyer, J. H. F. & Muller, M. W. (1990), 'Evaluating the quality of student learning. An unfolding analysis of the association between perceptions of learning context and approaches to studying at an individual level', *Studies in Higher Education* **15**(2), 131–154.

Mills, G. E. (2010), *Action Research: A Guide for the Teacher Researcher*, 4th edn, Pearson.

Mody, R. P. (1991), 'C in education and software engineering', *SIGCSE Bulletin* **23**, 45–56.

Morton, J. (2008), 'Learning to be a sport and exercise scientist: evaluations and reflections on laboratory-based learning and assessment', *The Journal of Hospitality Leisure Sport and Tourism* **7**(2), 93–100.

MySQL (2013), 'MySQL: The world's most popular open source database'. Accessed 2013-07-09.
URL: *https://www.mysql.com*

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. & Velázquez-Iturbide, J. A. (2002), Exploring the role of visualization and engagement in computer science education, *in* 'Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education', ITiCSE-WGR '02, ACM, New York, NY, USA, pp. 131–152.

Norton, L. (2004), 'Using assessment criteria as learning criteria: a case study in psychology', *Assessment & Evaluation in Higher Education* **29**(6), 687–702.

NVD3 (2013), 'NVD3: Re-usable charts for d3.js'. Accessed 2013-07-09.
URL: *http://nvd3.org*

OMG (2011), *OMG Unified Modeling Language (OMG UML), Superstructure*, OMG.
URL: *http://www.omg.org/spec/UML/2.4.1/Superstructure*

Palincsar, A. S. (1998), 'Social constructivist perspectives on teaching and learning', *Annual Review of Psychology* **49**(1), 345–375.

Palumbo, D. B. (1990), 'Programming language/problem-solving research: A review of relevant issues', *Review of Educational Research* **60**(1), 65.

Pardede, E. & Lyons, J. (2012), 'Redesigning the assessment of an entrepreneurship course in an information technology degree program: Embedding assessment for learning practices', *IEEE Transactions on Education* **55**(4), 1.

Pattis, R. E. (1990), A philosophy and example of CS-1 programming projects, *in* 'Proceedings of the 21st SIGCSE Technical Symposium on Computer Science Education', SIGCSE '90, ACM, New York, NY, USA, pp. 34–39.

Pattis, R. E. (1993), The procedures early approach in CS 1: a heresy, *in* 'Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education', SIGCSE '93, ACM, New York, NY, USA, pp. 122–126.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. & Paterson, J. (2007), 'A survey of literature on the teaching of introductory programming', *ACM SIGCSE Bulletin* **39**(4), 204–223.

Peixoto, C. E. L., Audy, J. L. N. & Prikladnicki, R. (2010), Effort estimation in global software development projects: Preliminary results from a survey, *in* 'Global Software Engineering (ICGSE), 2010 5th IEEE International Conference on', IEEE, pp. 123–127.

Penaluna, A., Media, D., , University, S. M., Swansea, Penaluna, K. & Wales (2009), 'Assessing creativity: drawing from the experience of the uk's creative design educators', *Education + Training* **51**(8), 718–732.

Pendergast, M. O. (2006), 'Teaching introductory programming to IS students: Java problems and pitfalls', *Journal of Information Technology Education* **5**, 491–515.

Petticrew, M. & Roberts, H. (2008), *Systematic Reviews in the Social Sciences: A Practical Guide*, Wiley-Blackwell.

Phillips, D. C. (1995), 'The good, the bad, and the ugly: The many faces of constructivism', *Educational Researcher* **24**(7), 5–12.

Phillips, R. (2005), 'Challenging the primacy of lectures: The dissonance between theory and practice in university teaching', *Journal of University Teaching & Learning Practice* **2**(1).

Piaget, J. (1950), *Psychology of Intelligence*, Routledge & Kegan Paul, London.

Plimmer, B. (2000), A case study of portfolio assessment in a computer programming course, *in* 'Proceedings of the 13th Annual Conference of the National Advisory Committee on Computing Qualifications', pp. 279–284.

Prosser, M. & Millar, R. (1989), 'The how and what of learning physics', *European Journal of Psychology of Education* **4**(4), 513–528.

Qiao, A., Sun, L. & Wang, N. (2009), The design of web-based learning activities a case study on learning activities design from mainland China, *in* '2nd IEEE International Conference on Computer Science and Information Technology', IEEE, pp. 126–129.

Raeburn, P., Muldoon, N. & Bookallil, C. (2009), Blended spaces, work based learning and constructive alignment: Impacts on student engagement, *in* 'Same Places, Different Spaces: Proceedings ASCILITE 2009', Auckland, pp. 820–831.

Ragonis, N. & Ben-Ari, M. (2007), 'A long-term investigation of the comprehension of OOP concepts by novices', *Computer Science Education* **15**(3), 203–221.

Rajaravivarma, R. (2005), 'A games-based approach for teaching the introductory programming course', *SIGCSE Bulletin* **37**(4), 98–102.

Raman, A. (2008), 'The need for a philosophical grounding in higher degree science research programmes', *Current Science* **95**(5), 590–593.

Ramsden, P. (1991), 'A performance indicator of teaching quality in higher education: The course experience questionnaire', *Studies in Higher Education* **16**(2), 129–150.

Ramsden, P. (1992), *Learning to Teach in Higher Education*, Psychology Press.

Ramsden, P. & Entwistle, N. J. (1983), *Understanding Student Learning*, Croom Helm.

Reges, S. (2006), Back to basics in CS1 and CS2, *in* 'Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education', SIGCSE '06, ACM, New York, NY, USA, pp. 293–297.

Renumol, V. G., Janakiram, D. & Jayaprakash, S. (2010), 'Identification of cognitive processes of effective and ineffective students during computer programming', *Transactions on Computing Education* **10**(3), 10:1–10:21.

Richards, J. C. & Lockhart, C. (1994), *Reflective Teaching in Second Language Classrooms*, Cambridge University Press.

Richardson, L. & Ruby, S. (2007), *RESTful Web Services*, O'Reilly Media.

Rist, R. (2004), Learning to program: Schema creation, application, and evaluation, *in* S. Fincher & M. Petre, eds, 'Computer Science Education Research', Routledge Falmer.

Rist, R. S. (1996), System structure and design, *in* 'Empirical Studies of Programmers, Sixth Workshop', pp. 163–194.

Ritchie, D. M., Johnson, S. C., Lesk, M. E. & Kernighan, B. W. (1978), 'The C programming language', *The Bell System Technical Journal* **57**(6), 1991–2019.

Roberts, E. S. (1993), 'Using C in CS1: Evaluating the Stanford experience', *SIGCSE Bulletin* **25**, 117–121.

Roberts, E. S. (1995), 'A C-based graphics library for CS1', *SIGCSE Bulletin* **27**, 163–167.

Robins, A., Rountree, J. & Rountree, N. (2003), 'Learning and teaching programming: A review and discussion', *Computer Science Education* **13**(2), 137–172.

Ross, J. (2010), Extending constructive alignment beyond unit content: a critical thinking and writing skills improvement project, *in* 'RMIT Accounting Educators' Conference', Melbourne Victoria.

Rountree, N., Rountree, J. & Robins, A. (2002), 'Predictors of success and failure in a CS1 course', *ACM SIGCSE Bulletin* **34**(4), 121–124.

Rowntree, D. (1977), *Assessing Students: How Shall We Know Them?*, Taylor & Francis.

Royce, W. W. (1970), Managing the development of large software systems, *in* 'Proceedings of IEEE WESCON', Vol. 26, Los Angeles.

Rubin, M. J. (2013), The effectiveness of live-coding to teach introductory programming, *in* 'Proceeding of the 44th ACM technical symposium on Computer science education', SIGCSE '13, ACM, New York, NY, USA, pp. 651–656.

Ruby, S., Thomas, D. & Heinemeier Hansson, D. (2013), *Agile Web Development with Rails 3.2*, The Pragmatic Bookshelf.

Salleh, N., Mendes, E. & Grundy, J. (2011), 'Empirical studies of pair programming for CS/SE teaching in higher education: A Systematic Literature Review', *Software Engineering, IEEE Transactions on* (99), 1–1.

Savery, J. R. & Duffy, T. M. (1995), 'Problem based learning: An instructional model and its constructivist framework', *EDUCATIONAL TECHNOLOGY-SADDLE BROOK NJ-* **35**, 31–31.

Schaefer, D. & Panchal, J. (2009), 'Incorporating research into undergraduate design courses: a patent-centered approach', *The International Journal of Mechanical Engineering Education* **37**(2), 98–110.

Schmidt, H. G., Van der Molen, H. T., Te Winkel, W. W. & Wijnen, W. H. (2009), 'Constructivist, problem-based learning does work: A meta-analysis of curricular comparisons involving a single medical school', *Educational Psychologist* **44**(4), 227–249.

Schön, D. A. (1983), *The Reflective Practitioner: How Professionals Think in Action*, Basic books.

Schulte, C. & Bennedsen, J. (2006), What do teachers teach in introductory programming?, *in* 'Proceedings of the Second International Workshop on Computing Education Research', ICER '06, ACM, pp. 17–28.

Schwaber, K. & Beedle, M. (2002), *Agile Software Development with Scrum*, Prentice Hall.

Schwartz, M. S., Sadler, P. M., Sonnert, G. & Tai, R. H. (2009), 'Depth versus breadth: How content coverage in high school science courses relates to later success in college science coursework', *Science Education* **93**(5), 798–826.

Scott, L. M. & Fortune, C. F. (2009), Promoting student centered learning: Portfolio assessment on an undergraduate construction management program, *in* 'Building Research and Education: ASC2009, Proceedings of the 45th Annual Conference'.

Scriven, M. (1967), The methodology of evaluation, *in* R. W. Tyler, R. M. Gagne & M. Scriven, eds, 'Perspectives of Curriculum Evaluation', Vol. 1, Rand McNally, Chicago, IL, pp. 39–83.

Sermersheim, J. (2006), *Lightweight Directory Access Protocol (LDAP): The Protocol*, The Internet Society.
URL: *http://tools.ietf.org/html/rfc4511*

Shannon, C. E. (1949), 'Communication in the presence of noise', *Proceedings of the IRE* **37**(1), 10–21.

Sheard, J., Carbone, A. & Dick, M. (2003), Determination of factors which impact on it students' propensity to cheat, *in* 'Proceedings of the fifth Australasian conference on Computing Education', Vol. 20 of *ACE '03*, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 119–126.

Sheard, J. & Dick, M. (2011), Computing student practices of cheating and plagiarism: a decade of change, *in* 'Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education', ITiCSE '11, ACM, New York, NY, USA, pp. 233–237.

Sheard, J., Simon, Carbone, A., D'Souza, D. & Hamilton, M. (2013), Assessment of programming: pedagogical foundations of exams, *in* 'Proceedings of the 18th ACM conference on Innovation and Technology in Computer Science Education', ITiCSE '13, ACM, New York, NY, USA, pp. 141–146.

Sheetz, S. D., Irwin, G., Tegarden, D. P., Nelson, H. J. & Monarchi, D. E. (1997), 'Exploring the difficulties of learning object-oriented techniques', *Journal of Management Information Systems* **14**, 103–131.

Shepherd, J. (2005), Weaving a web of consistency: a case study of implementing constructive alignment, *in* 'HERDSA 2005 Conference Proceedings'.

Shoufan, A. & Huss, S. A. (2010), 'A course on reconfigurable processors', *Trans. Comput. Educ.* **10**(2), 7:1–7:20.

Smith, E. & Gorard, S. (2005), ' 'They don't give us our marks': The role of formative feedback in student progress', *Assessment in Education: Principles, Policy & Practice* **12**(1), 21–38.

Smith, K. & Tillema, H. (2001), 'Long-term influences of portfolios on professional development', *Scandinavian Journal of Educational Research* **45**(2), 183–203.

Smith, K. & Tillema, H. (2003), 'Clarifying different types of portfolio use', *Assessment & Evaluation in Higher Education* pp. 625–649.

Soetanto, K. (2003), Proposing the IOC pedagogy for Japanese higher education in the Universalize era, *in* 'Proceedings of the 25th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (IEEE Cat. No.03CH37439)', IEEE, pp. 3521–3524.

Soetanto, K. (2012), Motivational education for science course, *in* 'Proceedings of IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE) 2012', IEEE, pp. W2A–5–W2A–7.

Soloway, E. (1986), 'Learning to program = learning to construct mechanisms and explanations', *Communications of the ACM* **29**, 850–858.

Soloway, E. & Spohrer, J. C. (1988), *Studying the Novice Programmer*, L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Steffe, L. P. & Gale, J. E. (1995), *Constructivism in Education*, Lawrence Erlbaum Associates, NJ.

Steffen, M., May, D. & Deuse, J. (2012), The industrial engineering laboratory, *in* 'Proceedings of the 2012 IEEE Global Engineering Education Conference (EDUCON)', IEEE, pp. 1–10.

Stephenson, J. & Laycock, M. (1993), *Using Learning Contracts in Higher Education*, Routledge.

Sutherland, J. & Schwaber, K. (2007), *The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process*.

SwinGame (2013), 'SwinGame: Build your imagination'. Accessed 2013-07-09.
URL: *http://www.swingame.com*

Szili, G. & Sobels, J. (2011), 'Reflections on the efficacy of a constructivist approach to teaching and learning in a first-year bachelor of environmental management topic', *Journal of Geography in Higher Education* **35**(4), 499–512.

Talay-Ongan, A. (2003), Online teaching as a reflective tool in constructive alignment, *in* 'Proceedings of International Education Research Conference AARE–NZARE, Auckland, New Zealand, Australian Association for Research in Education'.

Tang, C., Lai, P., Arthur, D. & Leung, S. F. (1999), 'How do students prepare for traditional and portfolio assessment in a problem-based learning curriculum', *Themes and Variation in PBL. Newcastle: Australian Problem Based Learning Network* pp. 206–217.

Taxén, G. (2004), 'Teaching computer graphics constructively', *Computers & Graphics* **28**(3), 393 – 399.

Teater, B. A. (2010), 'Maximizing student learning: A case example of applying teaching and learning theory in social work education', *Social Work Education* **30**(5), 571–585.

Terrell, J., Richardson, J. & Hamilton, M. (2011), 'Using Web 2.0 to Teach Web 2.0: A case study in aligning teaching, learning and assessment with professional practice', *Australasian Journal of Educational Technology* p. 17.

Thorpe, M. (2000), 'Encouraging students to reflect as part of the assignment process: Student responses and tutor feedback', *Active Learning in Higher Education* **1**(1), 79–92.

Thota, N. & Whitfield, R. (2010), 'Holistic approach to learning and teaching introductory object-oriented programming', *Computer Science Education* **20**(2), 103–127.

Thramboulidis, K. (2003*a*), 'A constructivism-based approach to teach object-oriented programming', *Journal of Informatics Education and Research* **5**(1), 1–12.

Thramboulidis, K. (2003*b*), Teaching advanced programming concepts in introductory computing courses: a constructivism based approach, *in* 'ICEE International Conference on Engineering Education'.

Thramboulidis, K. C. (2003*c*), 'A sequence of assignments to teach object-oriented programming: a constructivism design-first approach', *Informatics in Education* **2**(1), 103–122.

Tigelaar, D., Dolmans, D., Wolfhagen, I. & Vleuten, C. (2007), 'Quality issues in judging portfolios: implications for organizing teaching portfolio assessment procedures', *Studies in Higher Education* **30**(5), 595–610.

Treleaven, L. & Voola, R. (2008), 'Integrating the development of graduate attributes through constructive alignment', *Journal of Marketing Education* **30**(2), 160–173.

Trewin, D. (2000), *Australian Standard Classification of Education (ASCED)*, Australian Bureau of Statistics.

Trigwell, K. & Prosser, M. (1991), 'Improving the quality of student learning: the influence of learning context and student approaches to learning on learning outcomes', *Higher Education* **22**(3), 251–266.

Trigwell, K., Prosser, M. & Waterhouse, F. (1999), 'Relations between teachers' approaches to teaching and students' approaches to learning', *Higher Education* **37**(1), 57–70.

Twitter Bootstrap (2013), 'Twitter Bootstrap: Sleek, intuitive, and powerful front-end framework for faster and easier web development'. Accessed 2013-07-09.
URL: *http://twitter.github.io/bootstrap/*

Tyler, R. W. (1969), *Basic Principles of Curriculum and Instruction*, University of Chicago Press.

Tynjala, P. (1998), 'Traditional studying for examination versus constructivist learning tasks: Do learning outcomes differ?', *Studies in Higher Education* **23**(2), 173–189.

Van Canneyt, M. (2013), *Free Pascal: Language Reference Guide*.
URL: *ftp://ftp.freepascal.org/pub/fpc/docs-pdf/ref.pdf*

Van Canneyt, M. & Klämpfl, F. (2011), *Free Pascal User's Guide*.
URL: *ftp://ftp.freepascal.org/pub/fpc/docs-pdf/user.pdf*

Van Gorp, M. & Grissom, S. (2001), 'An empirical evaluation of using constructive classroom activities to teach introductory programming', *Computer Science Education* **11**(3), 247–260.

van Rossum, E. J. . J. & Schenk, S. M. (1984), 'The relationship between learning conception, study strategy and learning outcome', *British Journal of Educational Psychology* **54**(1), 73–83.

Vanfretti, L. & Milano, F. (2012), 'Facilitating constructive alignment in power systems engineering education using free and open-source software', *IEEE Transactions on Education* **55**(3), 309–318.

Vogel, D., Kennedy, D., Kuan, K., Kwok, R. & Lai, J. (2007), Do mobile device applications affect learning?, *in* '40th Annual Hawaii International Conference on System Sciences', IEEE, p. 4.

Vrasidas, C. (2000), 'Constructivism versus objectivism: Implications for interaction, course design, and evaluation in distance education', *International Journal of Educational Telecommunications* **6**(4), 339–362.

Warren, I. (2005), Teaching patterns and software design, *in* 'Proceedings of the 7th Australasian Conference on Computing Education', pp. 39–49.

White, G. & Sivitanides, M. (2005), 'Cognitive differences between procedural programming and object oriented programming', *Information Technology and Management* **6**(4), 333–350.

Wiedenbeck, S. (2005), Factors affecting the success of non-majors in learning to program, *in* 'Proceedings of the First International Workshop on Computing Education Research', ICER '05, ACM, New York, NY, USA, pp. 13–24.

Wiliam, D. (2006), 'Formative assessment: Getting the focus right', *Educational Assessment* **11**(3-4), 283–289.

Winslow, L. E. (1996), 'Programming pedagogy: a psychological overview', *ACM SIGCSE Bulletin* **28**(3), 17–22.

Wirfs-Brock, R. & McKean, A. (2003), *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley Professional.

Wirth, N. (1971), 'The programming language Pascal', *Acta informatica* **1**(1), 35–63.

Woodward, C. J., Cain, A., Pace, S. & Jones, A., F. K. J. (2013), Helping students track learning progress using burn down charts, *in* 'Proceedings of the 2nd IEEE International Conference on Teaching, Assessment and Learning for Engineering', IEEE, pp. 104–109.

Wulf, T. (2005), Constructivist approaches for teaching computer programming, *in* 'Proceedings of the 6th Conference on Information Technology Education', ACM, pp. 245–248.

Yip, W. (2005), Web-based support for constructive alignment, *in* 'Proceedings of the IASTED International Conference: Web-based Education', ACTA Press, Switzerland.

A1

Constructive Alignment Literature Survey Data

This appendix lists the data from the systematic literature review of applications of constructive alignment, as reported in peer reviewed literature. The data is presented in four tables:

1. Table A1.1 provides overview details of each paper, listing its field, the year level of the unit it described, the method of the unit's delivery, and the geographic location of the reported work.
2. Table A1.2 lists the details associated with the evaluation of constructive alignment, including the data sources used, and any reported positive or negative effects.
3. The teaching and learning activities used in the reported units is listed in Table A1.3, along with any method used to incorporate constructive learning theories in the delivery of the unit.
4. Finally, Table A1.4 lists the methods of assessment used and details of how the assessment, teaching methods, and intended learning outcomes were aligned.

A further discussion and analysis of this data is presented in Chapter 2.

A1.1 Paper Overview Details

Table A1.1 lists each paper, its field, year level, method of delivery and geographic location. The following abbreviations are used in the this table.

- Field values include:

  AERS : Agriculture, Environmental And Related Studies

  AB : Architecture And Building

  CA : Creative Arts

  EDU : Education

  ENG : Engineering And Related Technologies

  FHPS : Food, Hospitality And Personal Services

  HEAL : Health

  IT : Information Technology

  MC : Management And Commerce

  MIX : Mixed Field Programmes

  SCI : Natural And Physical Sciences

  SOC : Society And Culture

- Level values include:

  PG : Postgraduate

  UG : Undergraduate

  Yr 1 : First year

  Yr 2 : Second year

  Yr 3 : Third or later years

- Delivery values include:

  F2F : Face to Face

  OL : Online

Table A1.1: Summary details of papers analysed in systematic literature review on constructive alignment.

Author (Year)	Field	Level	Delivery	Location
Tang et al. (1999)	HEAL	UG (Yr 3)	F2F	Hong Kong
Hoddinott (2000)	SCI	UG (Yr 3) and PG	OL	Canada
Davey & Bond (2002)	HEAL	UG (Yr 3)	F2F	New Zealand
Talay-Ongan (2003)	EDU	UG & PG	F2F & OL	Australia
Norton (2004)	SOC	UG (Yr 3)	F2F	United Kingdom
Warren (2005)	IT	UG (Yr 2)	F2F	New Zealand
Shepherd (2005)	MC	UG (Yr 2)	F2F	Australia
Yip (2005)	IT	UG & PG	F2F	Hong Kong
Henderson (2006)	MC	UG (Yr 1)	F2F	Australia
Brown et al. (2006)	AERS	UG & PG	OL	United Kingdom
Israel et al. (2007)	SOC	UG (Yr 3)	F2F	South Africa
Vogel et al. (2007)	MC	UG & PG	F2F	Hong Kong
Jones (2007)	SOC	UG (Yr 3)	F2F & OL	Australia
Brabrand (2008)	IT	UG & PG	F2F	Europe
Treleaven & Voola (2008)	MC	PG	F2F	Australia
Morton (2008)	SOC	UG (Yr 2)	F2F	United Kingdom
Raman (2008)	AERS	UG (Yr 3)	F2F	Australia
Kuhn (2009)	MC	UG (Yr 1)	F2F	Australia
Raeburn et al. (2009)	HEAL	UG & PG	OL	Australia
Hill (2009)	IT	UG (Yr 3)	F2F	United Kingdom
Scott & Fortune (2009)	AB	UG (Yr 1)	F2F & OL	Europe
Schaefer & Panchal (2009)	ENG	UG & PG	F2F	America
Qiao et al. (2009)	EDU	Not Specified	F2F	China
Thota & Whitfield (2010)	IT	UG (Yr 1)	F2F	China
Teater (2010)	SOC	UG (Yr 2)	F2F	United Kingdom
Hartfield (2010)	SCI	UG (Yr 3)	F2F	Australia
Ross (2010)	MC	UG (Yr 3)	F2F	Australia
Shoufan & Huss (2010)	IT	UG & PG	F2F	Europe
Szili & Sobels (2011)	AERS	UG (Yr 1)	F2F	Australia
Andrews (2011)	IT	UG (Yr 3)	F2F	United Kingdom
Terrell et al. (2011)	SOC	PG	OL	Australia
Donnison & Edwards (2011)	EDU	UG (Yr 1)	F2F	Australia
Joseph & Juwah (2012)	HEAL	UG (Yr 3)	F2F	United Kingdom
Pardede & Lyons (2012)	IT	Not Specified	F2F	Australia
Kenney (2012)	MC	UG (Yr 2)	F2F	Australia
Hedges & Pacheco (2012)	MC	UG (Yr 1)	F2F	New Zealand
Vanfretti & Milano (2012)	ENG	UG & PG	F2F	Europe
Steffen et al. (2012)	ENG	PG	F2F	Europe

A1.2 Evaluation Data

Table A1.2 list the evaluation sources, and the reported positive and negative effects of applying constructive alignment. The following abbreviations are used in the this table.

- Evaluation sources include:

  SFT : Survey results from student feedback of teaching surveys.

  IV/FG : Student interviews of focus groups.

Table A1.2: Evaluation details from papers analysed in the systematic literature review of applications of constructive alignment.

	Evaluation Source						**+'ve Effects**				**-'ve Effects**			
Author (Year)	**SFT**	**Results**	**Student Work**	**Online Activity**	**IV/FG**	**Reflections on Teaching**	**Outcomes**	**Satisfaction**	**Engagement**	**Other**	**Staff Workload**	**Student Workload**	**Other**	**No Evaluation**
Tang et al. (1999)	✓												✓	
Hoddinott (2000)	✓						✓					✓		
Davey & Bond (2002)	✓							✓	✓					
Talay-Ongan (2003)	✓			✓				✓						
Norton (2004)	✓		✓						✓				✓	
Warren (2005)	✓	✓					✓	✓			✓	✓		
Shepherd (2005)	✓	✓						✓	✓					
Yip (2005)														✓
Henderson (2006)		✓			✓				✓					
Brown et al. (2006)			✓				✓							
Israel et al. (2007)	✓	✓					✓	✓			✓	✓		
Vogel et al. (2007)	✓	✓			✓					✓				
Jones (2007)														✓
Brabrand (2008)	✓							✓						
Treleaven & Voola (2008)	✓		✓			✓	✓							
Morton (2008)	✓									✓				
Raman (2008)														✓
Kuhn (2009)	✓	✓						✓						
Raeburn et al. (2009)	✓			✓			✓	✓	✓					
Hill (2009)	✓	✓					✓		✓	✓				
Scott & Fortune (2009)	✓				✓	✓	✓	✓		✓				
Schaefer & Panchal (2009)			✓							✓				

Continued on next page

Table A1.2 – *Continued from previous page*

	Evaluation Source						+'ve Effects				-'ve Effects			
Author (Year)	**SFT**	**Results**	**Student Work**	**Online Activity**	**IV/FG**	**Reflections on Teaching**	**Outcomes**	**Satisfaction**	**Engagement**	**Other**	**Staff Workload**	**Student Workload**	**Other**	**No Evaluation**
Qiao et al. (2009)	✓				✓		✓							
Thota & Whitfield (2010)	✓				✓		✓							
Teater (2010)	✓						✓	✓						
Hartfield (2010)	✓	✓	✓		✓		✓	✓						
Ross (2010)	✓	✓	✓			✓		✓						
Shoufan & Huss (2010)	✓	✓					✓	✓						
Szili & Sobels (2011)	✓		✓				✓	✓	✓					
Andrews (2011)		✓	✓				✓							
Terrell et al. (2011)			✓				✓	✓	✓			✓		
Donnison & Edwards (2011)														✓
Joseph & Juwah (2012)	✓		✓		✓		✓							
Pardede & Lyons (2012)	✓	✓					✓					✓		
Kenney (2012)	✓							✓	✓					
Hedges & Pacheco (2012)		✓		✓				✓						
Vanfretti & Milano (2012)					✓					✓				
Steffen et al. (2012)						✓	✓							

A1.3 Teaching and Learning Activities

Table A1.3 lists the teaching and learning activities used, and details on how constructivism was incorporated. This table uses the following abbreviation.

Construct : Method for incorporating constructive learning theories.

Table A1.3: Paper summary details from the systematic literature review on applications of Constructive Alignment.

	Activities			Construct		
Author (Year)	**Lectures**	**Tutorial Classes**	**Other**	**Interactive Classes**	**Group Discussion**	**Problem-based Learning**
Tang et al. (1999)	✓		✓		✓	✓
Hoddinott (2000)						
Davey & Bond (2002)	✓	✓		✓		✓
Talay-Ongan (2003)	✓	✓		✓		✓
Norton (2004)		✓		✓		✓
Warren (2005)	✓	✓		✓		✓
Shepherd (2005)	✓	✓		✓		✓
Yip (2005)	✓	✓				✓
Henderson (2006)						
Brown et al. (2006)						
Israel et al. (2007)	✓	✓			✓	✓
Vogel et al. (2007)						
Jones (2007)	✓	✓			✓	
Brabrand (2008)	✓	✓		✓		✓
Treleaven & Voola (2008)	✓	✓				
Morton (2008)	✓	✓			✓	✓
Raman (2008)						
Kuhn (2009)	✓	✓				
Raeburn et al. (2009)						
Hill (2009)	✓	✓				
Scott & Fortune (2009)	✓			✓	✓	
Schaefer & Panchal (2009)	✓					✓
Qiao et al. (2009)						
Thota & Whitfield (2010)	✓	✓		✓	✓	✓
Teater (2010)		✓		✓	✓	✓
Hartfield (2010)	✓	✓			✓	✓
Ross (2010)	✓	✓			✓	

Continued on next page

Table A1.3 – *Continued from previous page*

	Activities			**Const.**		
Author (Year)	**Lectures**	**Tutorial Classes**	**Other**	**Interactive Classes**	**Group Discussion**	**Problem-based Learning**
Shoufan & Huss (2010)	✓	✓	✓	✓		
Szili & Sobels (2011)	✓		✓			✓
Andrews (2011)	✓	✓				
Terrell et al. (2011)						
Donnison & Edwards (2011)	✓	✓	✓			
Joseph & Juwah (2012)	✓	✓				
Pardede & Lyons (2012)	✓	✓		✓		✓
Kenney (2012)	✓	✓				
Hedges & Pacheco (2012)	✓	✓				
Vanfretti & Milano (2012)						✓
Steffen et al. (2012)		✓				✓

A1.4 Assessment and Alignment

Table A1.4 lists the assessment methods used, and the means of aligning teaching and learning activities and assessment tasks to the reported unit's intended learning outcomes.

Table A1.4: Paper summary details from the systematic literature review on applications of Constructive Alignment.

	Assessment							Aligned By		
Author (Year)	**Exam**	**Tests**	**Assignments**	**Group Work**	**Participation**	**Portfolio**	**Reflective Journal**	**Staff Only**	**Little Details**	**Matrix/Table**
Tang et al. (1999)	✓					✓	✓		✓	
Hoddinott (2000)	✓	✓	✓				✓		✓	
Davey & Bond (2002)	✓			✓				✓	✓	
Talay-Ongan (2003)								✓	✓	
Norton (2004)			✓	✓				✓	✓	
Warren (2005)	✓		✓					✓	✓	
Shepherd (2005)	✓	✓	✓				✓	✓	✓	
Yip (2005)				✓				✓		
Henderson (2006)	✓		✓	✓				✓	✓	
Brown et al. (2006)								✓	✓	
Israel et al. (2007)	✓			✓				✓	✓	
Vogel et al. (2007)								✓	✓	
Jones (2007)			✓	✓				✓		✓
Brabrand (2008)	✓		✓					✓		
Treleaven & Voola (2008)	✓			✓	✓		✓	✓		
Morton (2008)			✓					✓	✓	
Raman (2008)			✓					✓	✓	
Kuhn (2009)	✓		✓	✓				✓		✓
Raeburn et al. (2009)						✓		✓	✓	
Hill (2009)			✓	✓				✓		✓
Scott & Fortune (2009)						✓			✓	
Schaefer & Panchal (2009)	✓		✓				✓	✓		✓
Qiao et al. (2009)								✓	✓	
Thota & Whitfield (2010)	✓		✓	✓			✓	✓		✓
Teater (2010)	✓							✓	✓	
Hartfield (2010)	✓		✓					✓		✓
Ross (2010)								✓	✓	
Shoufan & Huss (2010)	✓							✓		✓
Szili & Sobels (2011)	✓		✓				✓	✓	✓	
Andrews (2011)	✓		✓					✓		
Terrell et al. (2011)					✓		✓	✓		✓

Continued on next page

Table A1.4 – *Continued from previous page*

	Assessment							**Aligned By**		
Author (Year)	**Exam**	**Tests**	**Assignments**	**Group Work**	**Participation**	**Portfolio**	**Reflective Journal**	**Staff Only**	**Little Details**	**Matrix/Table**
Donnison & Edwards (2011)						✓		✓	✓	
Joseph & Juwah (2012)	✓		✓					✓		✓
Pardede & Lyons (2012)	✓		✓	✓				✓	✓	
Kenney (2012)	✓	✓	✓	✓				✓	✓	
Hedges & Pacheco (2012)	✓	✓	✓					✓		
Vanfretti & Milano (2012)	✓		✓	✓				✓	✓	
Steffen et al. (2012)	✓		✓	✓				✓	✓	

A2

Chapters from the Programming Arcana

The following list outlines the chapters from the Programming Arcana, and the concepts covered.

1. **Building Programs**: Introduces students to the tools they require, and shows them a basic, "Hello World", program they can compile to check that their tools are working.
   - **Programs** are introduced as a sequence of instructions that get the computer to perform actions.
   - **Machine and Assembly code** provides some context as to why compilers are necessary. Machine code is presented as the computer's natural language, and Assembly code as a first step toward making this code more human-friendly.
   - **Source code and compilers** are introduced with the idea of third generation languages, and the need for a compiler to convert source code to machine code.
   - The **Terminal** is introduced as a means of running programs, and the steps for using the compiler are presented. This section also introduces the **Bash** shell, along with commands to navigate through the file system.
   - The final concept outlines the code for a **Hello World** program in C and Pascal, together with the steps needed to compile and run this program.
2. **Program Creation**: describes how code can be written to create a *Program*.
   - Introduces the idea that a **Program** can be created in code, and that it has a name and a list of instructions for the computer to perform.
   - **Procedures** are introduced as a named group of instructions that performed a task. These instructions can be run using a **Procedure Call**.
   - The idea that procedures can be distributed in a **Library** was discussed.
   - Programming language terminology is also introduced, including **State-**

ments as the technical term for commands, **Expressions** for calculated values, **Types** to describe different kinds of data, and **Identifiers** as the names for artefacts such as the programs created and the procedures called.
    - **Comments** are discussed as a means of documenting code.
3. **Procedure Declaration**: Introduces the idea that you can create your own procedures to encapsulate the steps of a task.
    - **Procedure declaration** describes how procedures can be created as a sequence of instructions that are run when the procedure is called.
    - The concept of a **Program** is extended to indicate that a program's code can include procedure declarations.
4. **Storing and Using Data**: Makes programs more dynamic using variables and constants to store data, and functions to calculate values.
    - **Variables** are introduced as a means of storing data that changes within the code, while **Constants** are introduced as a means of storing data that does not change.
    - The **assignment statement** is introduced as the means of storing a value in the variable, and the concept of an **expression** is updated to indicate it can read a value from the variable.
    - Programming terminology related to the location of a variable is introduced; **local variables** are declared within a procedure, **global variables** within a program, and **parameters** are a means of enabling data to be passed to a procedure.
    - The different parameter passing options are presented, with **pass-by-value** indicating that the value of the expression in the procedure call was passed, while with **pass-by-reference** the parameter needs to be passed a *variable* to which it will refer.
    - Creating **Functions** to calculate values is also introduced, along with updating what an expressions is to include the use of **function calls**.
    - To realise these concepts, the previous **statement**, **program** and **procedure declaration** concepts are updated.
5. **Control Flow**: Introduces structured programming principles, along with the control flow mechanisms of selection and repetition.
    - **Boolean data** is discussed as a means of directing the control flow statements. This includes the use of **comparisons** to calculate boolean values, as well as the **logical operators** (*and*, *or*, and *not*).
    - Selection is described in terms of **branching**, including the ideas of **if statements** and **case statements**.
    - **Looping** introduces **pre-test loops** that repeated code zero or more times, and **post-test loops** that repeated code one or more times.
    - Other control flow statements are covered in the section on **jumping**. This

includes **break** to jump out of a loop, **continue** to jump to the end of a loop, **exit/return** to jump out of a function or procedure, and the infamous **goto** statement.

- Finally, the idea of grouping statements in a **compound statement** was presented, and explained in terms of providing a sequence of statements within the control flow statements.

6. **Managing Multiple Values**: Presents the use of arrays to make it easier to work with a large amount of data.
   - **Arrays** are shown as a means of managing a number of values in a single variable. **String** is discussed as an example of an array students have already been working with.
   - The importance of **pass-by-reference** is reinforced.
   - **For loops** are introduced as a convenient means of looping over the elements of an array.
   - The **Assignment statement** and **Expression** concepts are updated to indicate how arrays can be used.
7. **Custom Data Types**: Describes how developers can create types to help them organise the data in their programs, much as functions and procedures helped to organise functionality.
   - **Types** are described again in more detail to provide context.
   - **Type declaration** is discussed along with **records/structs**, **enumerated types** and **unions**, as well as what a **Program** can contain.
   - The **Assignment statement** and **Expression** concepts are updated to indicate how the various custom types can be used.
8. **Dynamic Memory Allocation**: Extends programs beyond the confines of the stack, allowing the allocation of data on the heap.
   - The **Stack** and **Heap** are discussed. This highlights the need for values on the stack to have a known size, requiring another “space” for allocating data when its size is not known at compile time.
   - **Pointers** are introduced as a means of referring to space allocated on the Heap.
   - The need for specific actions to **allocate memory**, and to **free** that allocation are presented.
   - Common **issues with pointers** are discussed, including why they are likely to occur and how to address these issues. This includes **access violations**, **memory leaks** and **accessing released memory**.
9. **Input and Output**: Describes how to save and load data from file.
   - The concept of **persisting data** is discussed along with the idea of a process and its memory being freed after a program terminates. This leads to details on saving data from the program’s memory onto persistent storage.

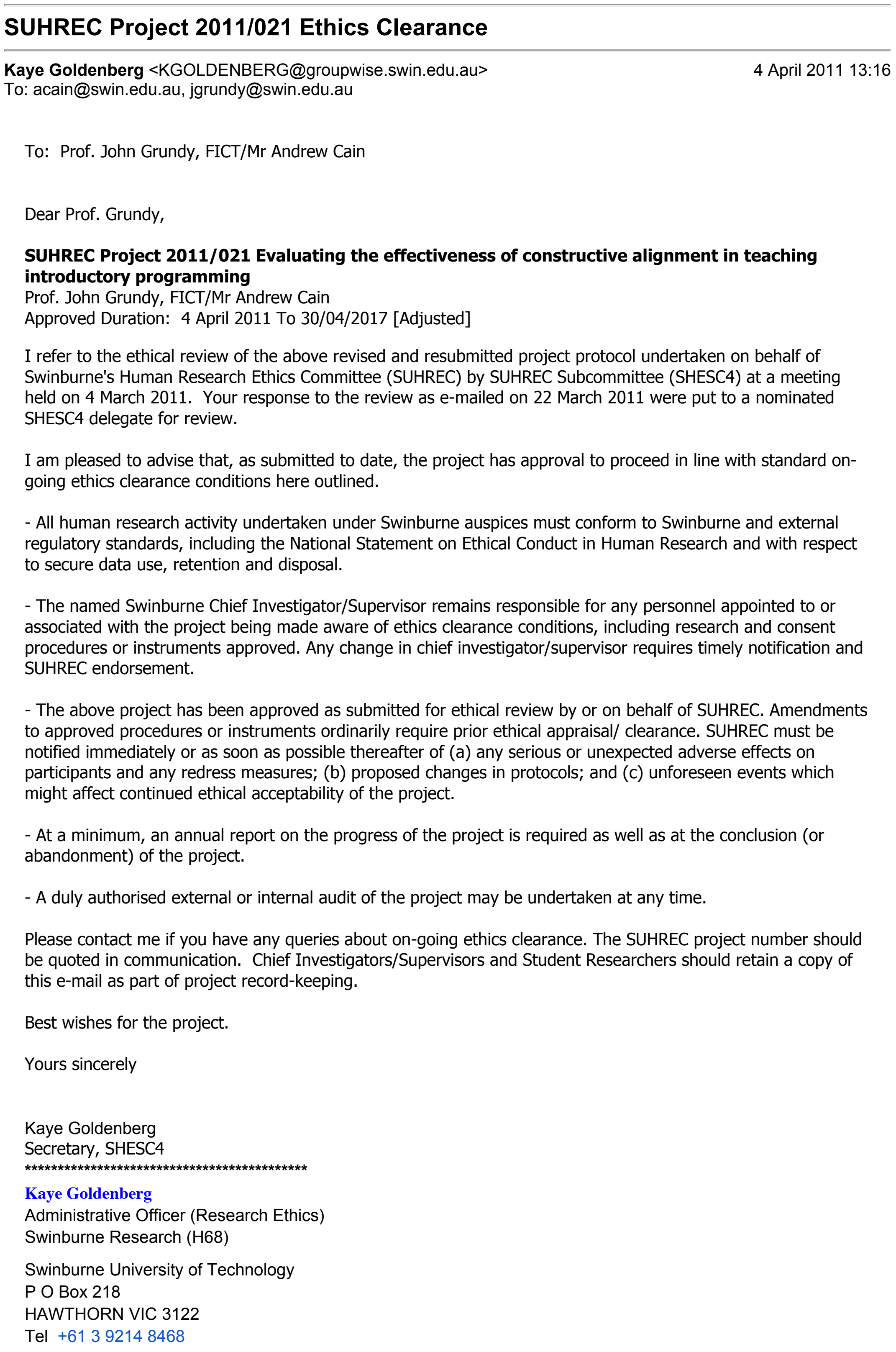
- **Files** and text and binary **file formats** are discussed.
- **Interacting with Files** describes typical input and output operations you likely to perform on files.
- **Other output devices** relates the concepts presented to terminal input/output and the idea that the same concepts apply to sending data across a network connection.

A3

Ethics Approval for Research Protocol

This appendix includes one attachment related to the granted ethics approval for the work reported in this thesis: SUHREC Project 2011/021 Ethics Clearance – email received indicating approval to carry out this research.

All conditions pertaining to this clearance were properly met, and annual reports have been submitted each year as per the required reporting standards.



SUHREC Project 2011/021 Ethics Clearance

Kaye Goldenberg <KGOLDENBERG@groupwise.swin.edu.au> 4 April 2011 13:16
To: acain@swin.edu.au, jgrundy@swin.edu.au

To: Prof. John Grundy, FICT/Mr Andrew Cain

Dear Prof. Grundy,

SUHREC Project 2011/021 Evaluating the effectiveness of constructive alignment in teaching introductory programming
Prof. John Grundy, FICT/Mr Andrew Cain
Approved Duration: 4 April 2011 To 30/04/2017 [Adjusted]

I refer to the ethical review of the above revised and resubmitted project protocol undertaken on behalf of Swinburne's Human Research Ethics Committee (SUHREC) by SUHREC Subcommittee (SHESC4) at a meeting held on 4 March 2011. Your response to the review as e-mailed on 22 March 2011 were put to a nominated SHESC4 delegate for review.

I am pleased to advise that, as submitted to date, the project has approval to proceed in line with standard on-going ethics clearance conditions here outlined.

- All human research activity undertaken under Swinburne auspices must conform to Swinburne and external regulatory standards, including the National Statement on Ethical Conduct in Human Research and with respect to secure data use, retention and disposal.

- The named Swinburne Chief Investigator/Supervisor remains responsible for any personnel appointed to or associated with the project being made aware of ethics clearance conditions, including research and consent procedures or instruments approved. Any change in chief investigator/supervisor requires timely notification and SUHREC endorsement.

- The above project has been approved as submitted for ethical review by or on behalf of SUHREC. Amendments to approved procedures or instruments ordinarily require prior ethical appraisal/ clearance. SUHREC must be notified immediately or as soon as possible thereafter of (a) any serious or unexpected adverse effects on participants and any redress measures; (b) proposed changes in protocols; and (c) unforeseen events which might affect continued ethical acceptability of the project.

- At a minimum, an annual report on the progress of the project is required as well as at the conclusion (or abandonment) of the project.

- A duly authorised external or internal audit of the project may be undertaken at any time.

Please contact me if you have any queries about on-going ethics clearance. The SUHREC project number should be quoted in communication. Chief Investigators/Supervisors and Student Researchers should retain a copy of this e-mail as part of project record-keeping.

Best wishes for the project.

Yours sincerely

Kaye Goldenberg
Secretary, SHESC4
****
Kaye Goldenberg
Administrative Officer (Research Ethics)
Swinburne Research (H68)

Swinburne University of Technology
P O Box 218
HAWTHORN VIC 3122
Tel +61 3 9214 8468