View Specification of Multi-View Visual Environments

By

Yafei Xiang

A thesis submitted in fulfilment of the requirements for the degree of Master of Science in

Computer Science, The University of Auckland, 2006

Department of Computer Science

University of Auckland

New Zealand

September 2006

ABSTRACT

Multi-view visual environments are popular tools in a wide variety of application domains, including software design, circuit design, architecture design, and process management. They provide advanced information management that is accessed through multiple views, typically represented by a mixture of graphical and textual notations that in turn describe certain concepts or views of the system.

Developing such visual environments – even just a small tool applying a simple visual language – would require considerable effort to overcome various difficulties. The most common problem encountered is the need to manage consistency and inconsistency between representations, and between multiple views of a single representation.

In this thesis, we present our ideas on the provision of view specification support for multi-view visual environments, culminating in the presentation of a language, VSL, to facilitate the view specification effort in developing multi-view visual environments. We also present a prototype VSL tool that we created and its incorporation into Pounamu, a meta-tool for building diverse multi-view visual environments.

PREFACE

Acknowledgements

First and foremost I would like to express my sincere gratitude to my two supervisors, John Hosking and John Grundy, whose insightful ideas, enthusiasm and support for this project have been invaluable. Without their help my work would have been much the poorer.

Heartfelt thanks go to my entire family. You are always supportive, encouraging and loving, even when I am not. I have been blessed in many ways, but none more so than to have you as my family.

Finally, special thanks to Quincy who helped me to realize happiness is the most important thing in my life.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Introduction

As software and physical systems have become increasingly complex, the need for separation of concerns in software engineering has emerged as a crucial element. One of the most effective means to satisfy this requirement is to access information through multiple views, each of which visualises certain pieces of information that interest specific readers (e.g. users, designers, programmers, or managers). As a result, multi-view visual environments have been adopted in a wide variety of application domains including software design, circuit design, architecture design, and process management. Developing such visual environments – even just a small tool applying a simple visual language – would require considerable effort to overcome various difficulties. Fortunately, many frameworks, toolkits, and meta-tools have been created to facilitate the development of such visual environments. These include MetaEdit+ [Kelly et al., 1996], Metabuilder [Ferguson et al., 2000], JView [Grundy et al., 1998a], Vampire [McIntyre, 1995], and DiaGen [Minas and Viehstaedt, 1995]. In particular, Pounamu [Zhu et al., 2004] is both a practical and handy meta-tool for specification and generation of multi-view, multi-notational visual tools.

Pounamu can be used to rapidly design, prototype, and evolve tools supporting a wide range of visual notations. A tool is defined as a meta-specification in Pounamu through its five dedicated components: a shape designer, a connector designer, a meta-model designer, an event handler designer, and a view designer. After specified, the tool is automatically generated and can be used for modelling immediately. Changes to the tool meta-description can be made at any time – even when the tool is in use. Most importantly, Pounamu permits its tools to be easily extended by addition of both built-in and custom event handlers, integration with other tools via a web services-based API, or model transformation (e.g. code generation) using standard XML techniques (e.g. DOM and XSLT). Having theses capabilities is certainly important to the development of multi-view visual tools, but they are relatively lacking in other existing approaches that attempt to address the same issue.

However, specifying views in Pounamu can be quite difficult; this is especially true when the target-modelling domain requires complex mappings that maintain consistency between multiple views and the underlying information model (or repository). For example, a visual notational element may be mapped to a single or multiple repository entities, a visual aspect (e.g. containment) perceived among such elements may actually convey certain physical relationships existed in the repository, or the ability of using external values may be necessary in some circumstances. At present, Pounamu lacks high-level support for such cases that have been introduced above. Neither the view designer nor the event handler designer can be easily used to achieve advanced view specification needs. The view designer adopts a form-based metaphor, using dropdown lists and buttons, which is in some areas hard to use and is limited to the specification of one-to-one mappings. Although the event handler designer can be used to implement complex mappings, much detailed programming knowledge is usually compulsory, providing high barriers to easy use.

Therefore, introducing high-level view specification support for Pounamu and similar development systems is extremely beneficial. In this thesis, we present a high-level, declarative approach to Pounamu view specification and generation. In doing so, we develop a visual modelling language called VSL for view specification, and implement a prototype tool supporting VSL for view generation.

1.2 Thesis Outline

The following outlines the contents of each of the subsequent chapters of this thesis:

Chapter Two: Background and Related Research – provides an overview of multi-view visual environments, introduces Pounamu, and reviews previous research that is related to our work.

Chapter Three: Preliminary VSL Designs – presents the processes that we used to design a visual modelling language for view specification, and our recommendations for the final language design.

Chapter Four: VSL Notation Specification – gives a deep technical view about the View Specification Language (VSL), which is developed based on the experience noted in the last chapter.

Chapter Five: VSL Examples – contains typical examples of view specifications expressed using VSL.

Chapter Six: VSL Tool Implementation – shows how a tool for VSL could be built and how we managed to do our own prove-of-concept implementation.

Chapter Seven: Evaluation – evaluates VSL and the prototype VSL tool we developed, and discusses the strengths and weaknesses of the system.

Chapter Eight: Conclusion and Future Work – concludes the thesis by summarising the major contributions of the work and providing suggestions and possible directions for future work.

CHAPTER 2

BACKGROUND AND RELATED RESEARCH

2.1 Introduction

This chapter presents the background information and related research for this thesis. We start by introducing multi-view visual environments as a background to the problem we are trying to solve and where said problem arises. We then briefly describe an existing meta-tool, Pounamu, which is used as an example of a system to study our research problem. Following this, we address some of the related research that is similar to or provides some basis for the work in this thesis. Finally, we summarize our findings and introduce our plan of solution.

2.2 Multi-view Visual Environments

Software engineering and design environments (or tools) are large, complex pieces of software. They provide advanced information management that is accessed through multiple views, typically represented by a mixture of graphical and textual notations that in turn describe certain concepts or views of the system. For example, a software engineering CASE tool may provide editors for object-oriented analysis notations, such as entity-relationship diagrams, user interface specification diagrams, and those (e.g. Use Case diagrams, Class diagrams, and etc) of the Unified Modelling Language (UML) [UML, 2000].

One of the main advantages of using a multi-view visual environment is its ability to support multiple representations (views) of a single information model. Each of these views provides a partial representation of that model with a certain degree of abstraction, e.g. grouping of similar concepts or constructs at different conceptual levels. The information expressed this way is certainly easy for the user to read, comprehend, and organize. More importantly, this ability is essential for many software design tools, which allow developers to work with software components at different levels of abstraction, using different representations [Meyers, 1991].

The most common problem encountered in the development of such environments and tools is the need to manage consistency and inconsistency between representations, and between multiple views of a single representation. That is, when one view is changed, the other views sharing the affected data must be updated to reflect the change as well. This problem is generally known as view consistency management.

To build such complex tools, a common architecture (adopted by many existing approaches such as [Reiss, 1985], [Meyers, 1991] and [Grundy et al., 2000]) is to have an underlying information model (or repository) that represents the data structures embodied in the tool, with mappings to and from tool (i.e. editor) based representations and views that maintain consistency between various models. Figure 2-1 illustrates the structure of this architecture.

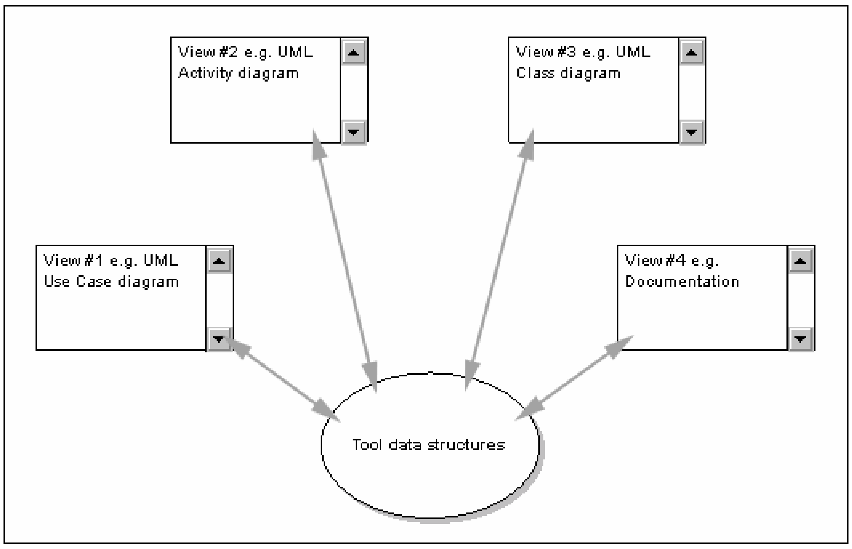


Figure 2-1 Common architecture of multi-view visual environments

[Grundy and Hosking, 2001] provides a broad discussion about software tools.

2.3 Pounamu – an Example of a System to Improve

Pounamu is a meta-tool, developed here at the University of Auckland [Zhu et al., 2004], for specification and generation of multi-view, multi-notational visual tools and environments. The main features of Pounamu are listed as follows:

- It provides an integrated environment (IDE) for developing other tools, such as CASE tools and domain-specific visual languages (DSVLs).
- It supports round-trip engineering and live, evolutionary development so that tools can be designed, prototyped, and evolved rapidly.
- It supplies a set of high-level designers – most of which are visual programming tools with relatively simple appearance and semantics – for specifying the meta-specification of a tool.
- It automatically generates a tool as specified, and allows changes to be made to the meta-tool specification at any time – even when the tool is in use.

- It provides tools with built-in multiple view support, which allows multiple representations (views) of the same underlying information model.
- It has an automatic view consistency mechanism (based on CPRGs [Grundy et al., 1998b]) to keep multiple views of the same information consistent.
- It allows its tools to be extended by addition of both bespoke and custom event handlers, integration with other tools via a web services-based API, or model transformation (e.g. code generation) using standard XML techniques (e.g. DOM and XSLT).
- It allows non-experts and even non-programmers to easily develop exploratory tools.

A wide variety of tools have been developed using Pounamu. These include an electrical circuit designer (Figure 2-2, a), a software process modelling (b), and a web services composition tool (c).

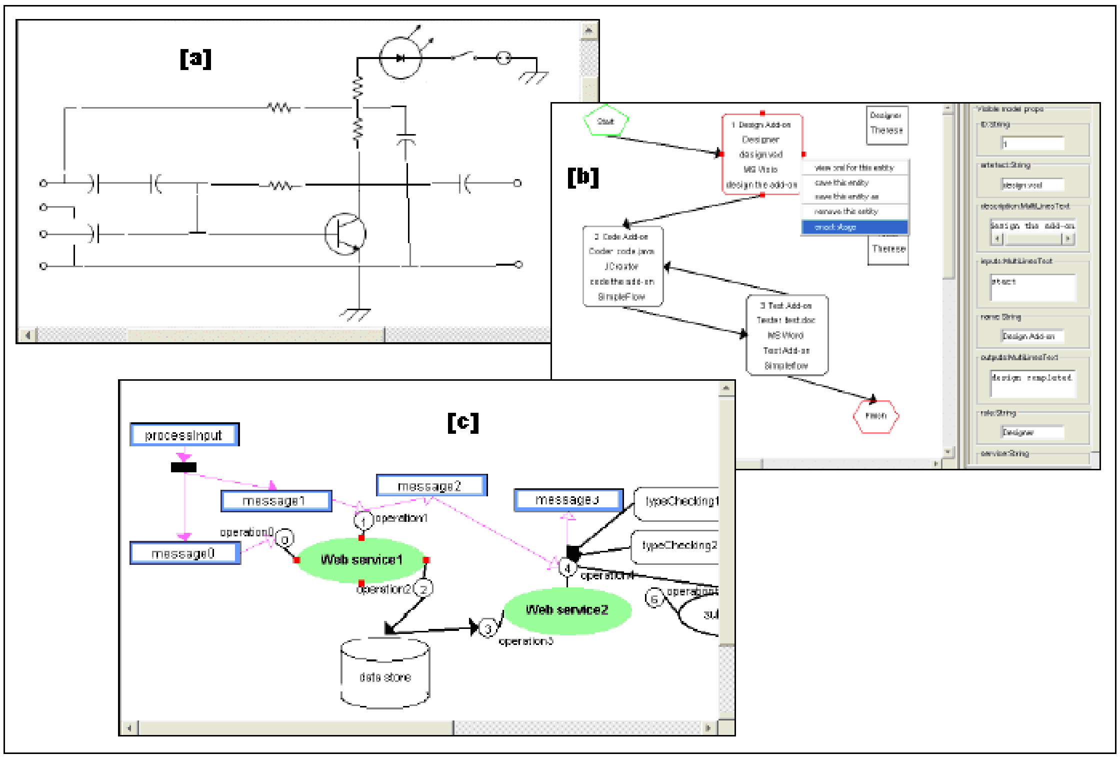


Figure 2-2 Examples of Pounamu applications, from [Zhu et al., 2004]

Figure 2-3 shows the main components of the Pounamu meta-tool. A desired tool (such as one of those shown in Figure 2-2) is initially defined as a meta-tool specification in Pounamu. There are five specification tools: the shape/connector designer permits specification of visual language notation components (e.g. shapes and connections); similarly, the meta-model designer permits specification of the tool's underlying information model components (e.g. entities and associations); the event handler designer allows definition of event handlers to define behaviour semantics; and the view designer allows definition of views (view types) for graphical display and editing of information. Individual tool specifications are grouped into tool projects.

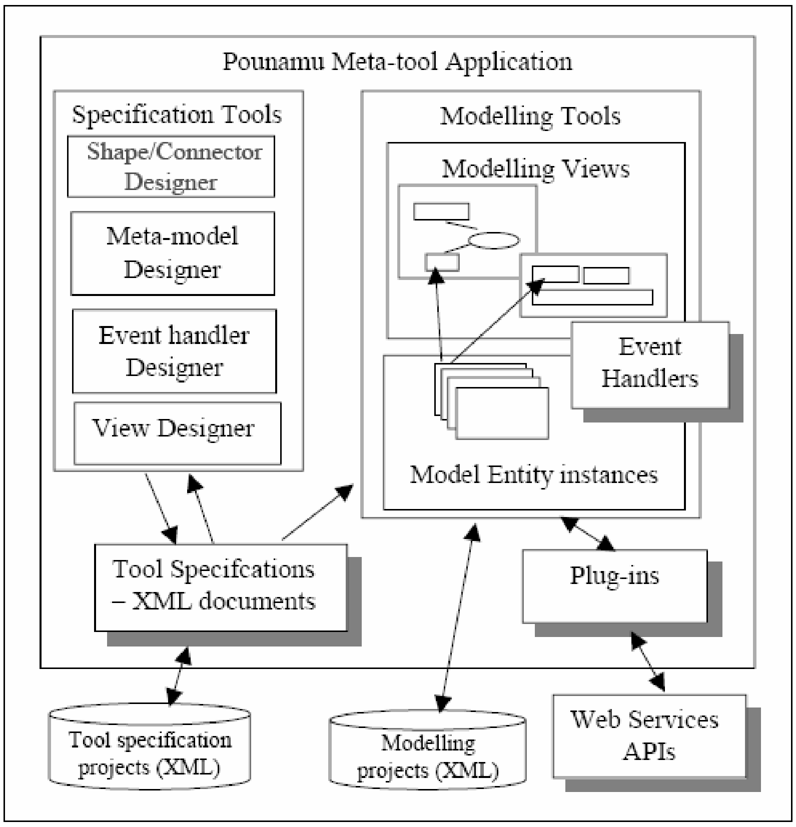
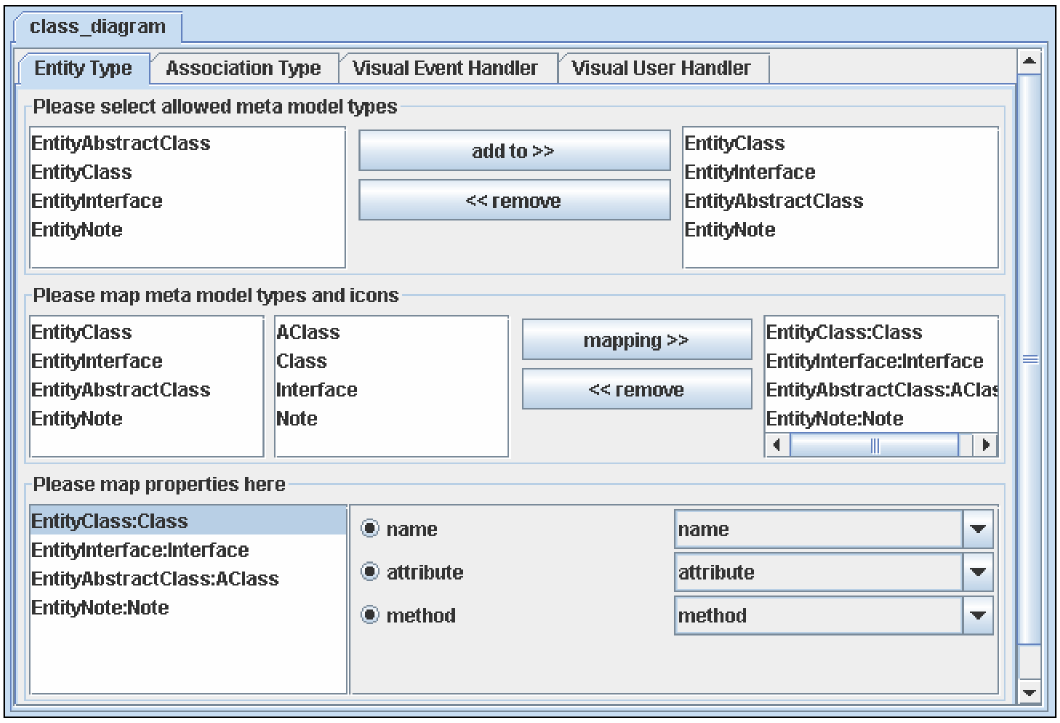


Figure 2-3 The Pounamu approach, from [Zhu et al., 2004]

With a tool project specification, multiple project models associated with that tool can be created. Modelling tools are used to create modelling projects and views, to edit view shapes and connections, and to update model entities and associations.

Pounamu uses view-to-model mappings (element mappings between view instance and information model instance) for view consistency management. Simple one-to-one mappings (from view shape/connection types to model entity/association types) are supported in the form-based view designer. More complex mappings (e.g. one-to-many and many-to-many) can be specified using event handlers (pieces of particular Java code). Examples of the view designer and the event handler designer are given as following.

Figure 2-4 shows an example of the view designer, which is used to define a visual editor and its mapping to the underlying information model. Each view type defined contains the shape and connection types that are allowed in corresponding view instances, along with a mapping from each of them to a matching (model) entity/association type. As a form-based designer with limited facilities, this tool only supports simple 1:1 view-to-model element mappings, e.g. the Class shape type maps to the EntityClass entity type, as shown in Figure 2-4.



Please select allowed meta model types

EntityAbstractClass	EntityClass
EntityClass	EntityInterface
EntityInterface	EntityAbstractClass
EntityNote	EntityNote

Please map meta model types and icons

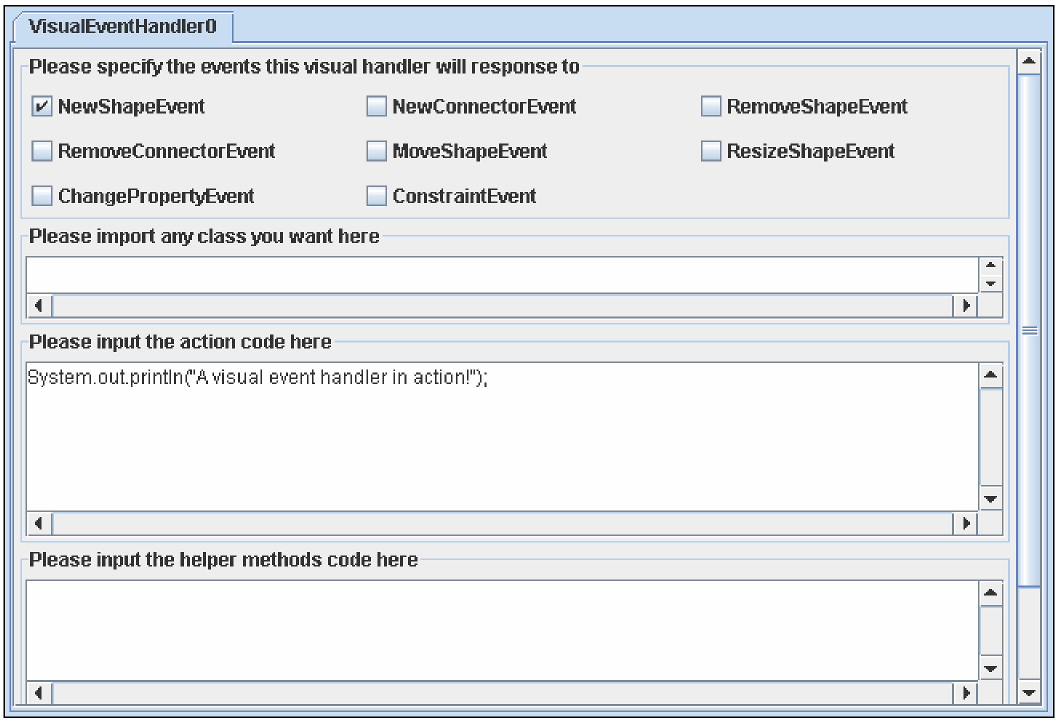
EntityClass	AClass	EntityClass:Class
EntityInterface	Class	EntityInterface:Interface
EntityAbstractClass	Interface	EntityAbstractClass:AClas
EntityNote	Note	EntityNote:Note

Please map properties here

EntityClass:Class	● name	name
EntityInterface:Interface	● attribute	attribute
EntityAbstractClass:AClass	● method	method
EntityNote:Note		

Figure 2-4 Example of the view designer

Figure 2-5 shows an example of the event handler designer, which is used to specify event handlers. An event handler is basically a piece of Java code that detects specific events (e.g. shape/connection addition) fired in the system and responds to those events with predefined actions (via Pounamu's API). In general, event handlers are used to add various constraints, complex mappings, and back end support (e.g. code generation).


VisualEventHandler0

Please specify the events this visual handler will response to

[✓ NewShapeEvent] NewConnectorEvent RemoveShapeEvent RemoveConnectorEvent MoveShapeEvent ResizeShapeEvent ChangePropertyEvent ConstraintEvent

Please import any class you want here

Please input the action code here

```
System.out.println("A visual event handler in action!");
```

Please input the helper methods code here

Figure 2-5 Example of the event handler designer

More detailed information about Pounamu can be found in [Zhu et al., 2004]. In the research presented in this thesis, we use Pounamu view specification as the motivation and exemplar application of our ideas, but these ideas have broader applicability.

2.4 Related Research

Many approaches have been proposed for the kinds of view consistency management described in the previous section:

High-level programming paradigms provide a software architecture or design pattern for building quite generic applications. These include Model-View-Controller (MVC) [Krasner and Pope, 1988] and Abstraction-Link-View (ALV) [Hill et al., 1994]. MVC supports the concept of views of data model, with the ability to propagate objects describing model object changes to observing objects. Similarly, ALV uses flexible inter-object constraints allowing changes made to repository or view specification objects to maintain view consistency. These paradigms typically lack abstractions specific to multi-view visual environments, causing the implementation of desired view consistency control logic to be time-consuming.

Special frameworks, such as Meta-MOOSE [Ferguson et al., 1999], JViews [Grundy et al., 1998], and Escalante [McWhirter and Nutt, 1994], provide low-level yet very powerful sets of reusable facilities for building visual language environments. Many such frameworks adopt an event propagation approach, in which an event travels through from the source to all potential listeners. From this perspective, they are similar to the high-level programming paradigms, but with specified and sophisticated view consistency mechanisms. However, these frameworks require detailed programming and class framework knowledge, limiting their ease of use.

Some CASE tools, meta-CASE tools, and meta-tools provide high-level support for developing multi-view, multi-notational visual tools. These include DiaGen [Minas and Viehstaedt, 1995], JComposer [Grundy et al., 1998], Kaitiaki [Liu et al., 2005], MetaEdit+ [Kelly et al., 1996], GME [Ledeczi et al., 2001], IPSEN [Klein and Schürr, 1997], and Pounamu. Some of them, such as DiaGen, JComposer, and Kaitiaki, adopt a code generation approach from a specification model, suffering from problems similar to those low-level frameworks, as custom coding work is usually required (especially for complex cases). Some of them, such as MetaEdit+, GEM, and IPSEN, use constraints (e.g. OCL [Warmer and Kleppe, 1998]) and formalisms (e.g. graph grammars and graph rewriting rules [Glinert, 1990]) to maintain view consistency, but have a steep initial learning curve and tend to be more difficult to master. Others, such as Pounamu, have a more flexible architecture design, with the notion of data mapping relating repository to related visual views, but support only limited mapping types (e.g. one-to-one) at the high level.

In addition, there are a number of mapping tools that provide insight to us by solving the related *mapping* problem:

Domain-specific mapping tools, such as VML [Amor, 1997], RVM [Grundy et al., 2001], FBM [Li et al., 2002], and TSF [Peltier et al., 2000], use a high-level, declarative language for specifying data transformation which is compiled into a lower, "concrete" level for implementation. There are several characteristics found in these tools. Firstly, with a declarative nature much detailed mapping implementation is hidden behind the scenes; thus, making it easier for users to capture a high-level view of their mapping specifications. Secondly, most of these tools (e.g. VML, RVM, and FBM) adopt domain-specific yet easy-to-understand visual notations, providing a low barrier for the users to start with. Finally, these tools can be easily altered to support a number of different underlying implementation techniques, including programming code (e.g. VML, RVM, and FBM), XSLT (e.g. TSF), and many others as appropriate.

In the following sub-sections, we first briefly elaborate on the most influential research that is similar to or provides some basis for the work in this thesis, and then summarize our findings by comparing typical examples of the related approaches that have been noted above.

2.4.1 JComposer (Grundy et al)

[Grundy et al., 1998a] developed a meta-CASE tool, JComposer, which can be used to develop complex CASE tools and design environments based on an existing framework, JViews (in fact, JComposer itself is developed using this framework). In doing so, JComposer provides several visual notations for specifying JViews-based tools, and generates JViews implementations (Java classes) of these tools. JComposer specifications are used to specify both the repository and view level components of an environment (structure), the structural relationships between them (semantics), and the changes propagated along the relationships (event-handling). Two specification examples are given in Figure 2-6.

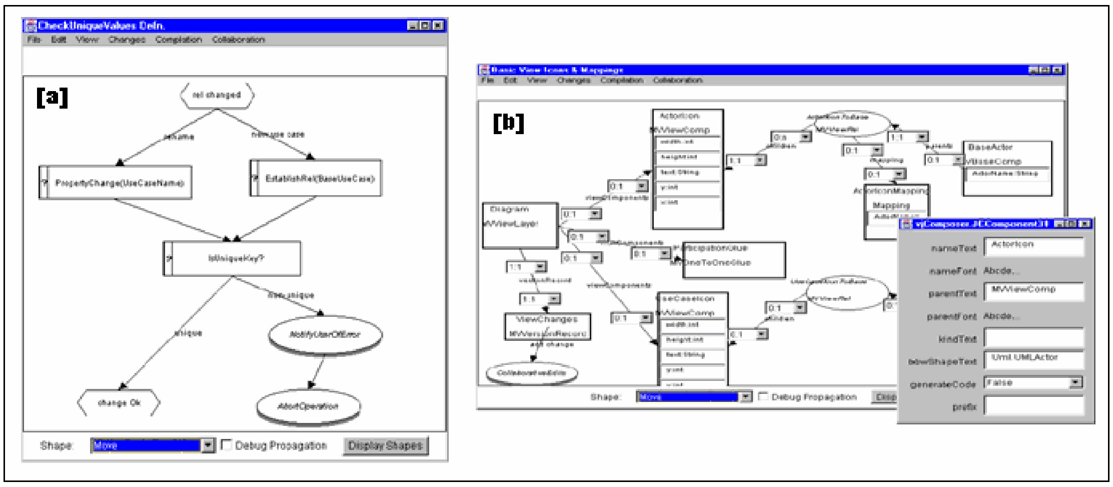


Figure 2-6 Examples of JComposer specifications, from [Grundy et al., 1998a]

Figure 2-6 (a) shows an event handler specification. The notation used is based on a visual event-flow programming style (procedural). Actions (shaded ovals) and Filters (rectangles) can be attached, by Event Flows (arrowed lines), to components and relationships in order to detect and react to selected events (change descriptions).

Figure 2-6 (b) shows a view model and view mapping specification. The notation used has a visual form containing elements in common with UML Class diagrams (declarative). Each view Component (rectangle) can be connected by a Relationship (oval) to a corresponding repository Component (rectangle), with mapping Components (rectangles) to specify the mapping of view component changes to the repository component and vice versa.

2.4.2 Kaitiaki (Liu et al)

[Liu et al., 2005] describe another flow-based approach to event-handling specification. The approach uses a visual Event-Query-Action-Filter language, Kaitiaki, to provide end users ways of expressing event-handling mechanisms via visual specifications. An example is given in Figure 2-7.

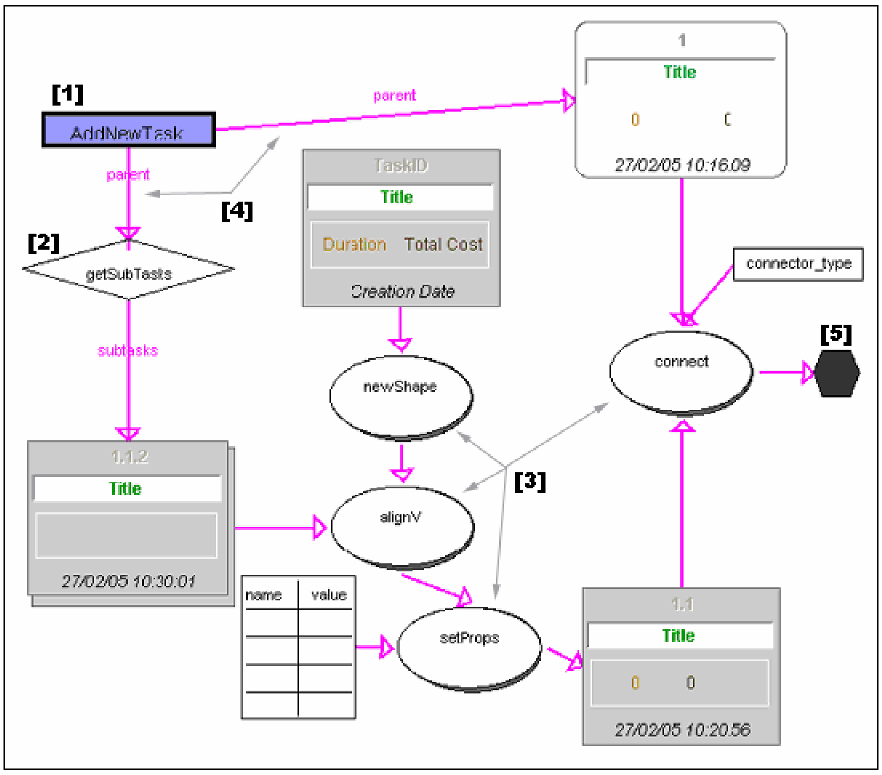
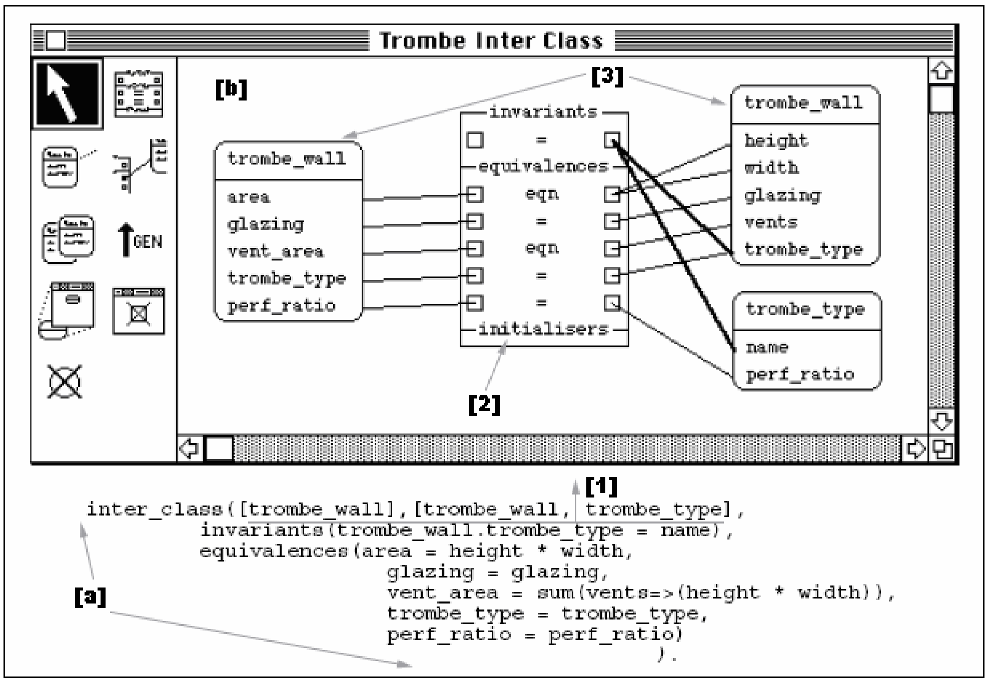


Figure 2-7 Example of a Kaitiaki event specification, from [Liu et al., 2005]

As shown in Figure 2-7, an Event (e.g. 1) can propagate various data flows (e.g. affected objects and property values changed). Queries (e.g. 2), Actions (e.g. 3), and Filters are parameterized with data propagated through incoming Data Propagation Links (e.g. 4). A Query retrieves elements and outputs them; a Filter selects elements from its input; and an Action applies operations to elements passed in. After going through all other elements, an event specification ends with an outgoing Port (e.g. 5).

2.4.3 VML (Amor)

[Amor, 1997] developed a high-level, declarative and bidirectional language, VML, for transferring one design tool view to another in the architecture, engineering, and construction (AEC) domains. The main goal of this language is to enable the developers to capture the essence of the mapping required between representations and to specify detailed correspondences between them [Grundy et al., 2004]. To complement the textual notation of VML, the author also provided a graphical notation called VML-G, which describes a subset of the full VML language and is aimed at high-level views of the mapping specification. An Example is given in Figure 2-8.



```
inter_class([trombe_wall],[trombe_wall, trombe_type],
        invariants(trombe_wall.trombe_type = name),
        equivalences(area = height * width,
                     glazing = glazing,
                     vent_area = sum(vents=>(height * width)),
                     trombe_type = trombe_type,
                     perf_ratio = perf_ratio)
                                  ).
```

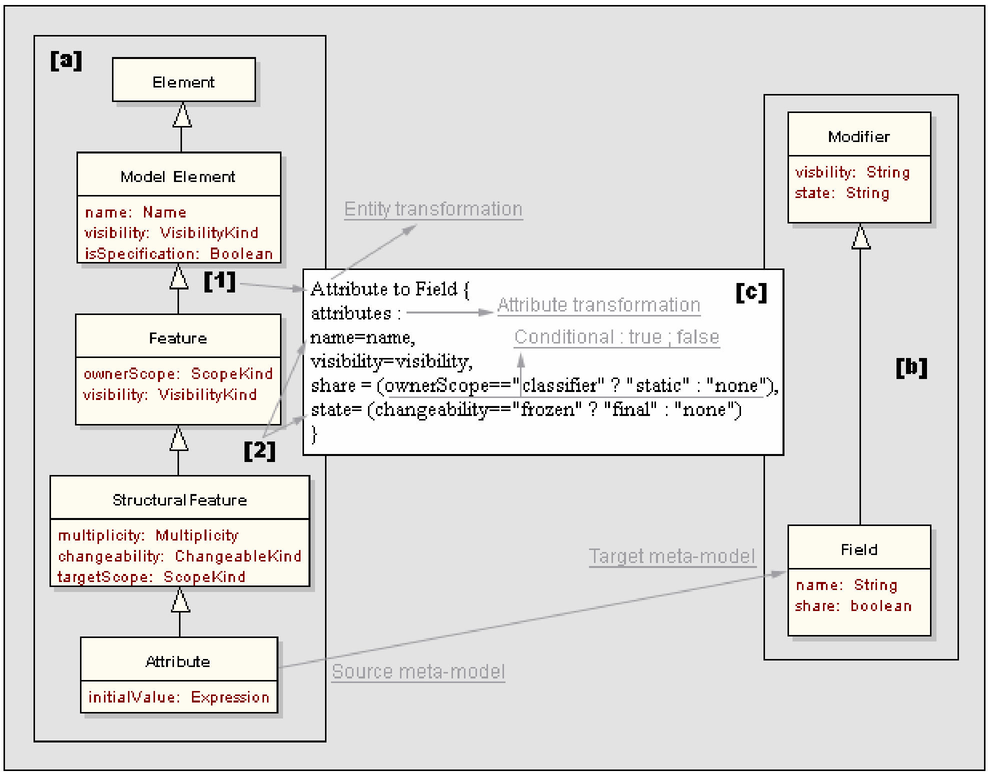
Figure 2-8 (a) A VML mapping and (b) its VML-G representation, from [Grundy et al., 2004]

A VML mapping is made up of a specification of the schemas to be mapped between, and a set of correspondences between entities and attributes, called inter_class specifications, describing how the mapping is to be achieved [Grundy et al., 2004]. An inter_class definition (e.g. Figure 2-8, a) generally contains four parts: the first part (e.g. 1) indicates the entities that are involved in the mapping being specified; the second part (optional) describes the conditions which must hold to use the mapping (invariants); the third part specifies correspondences between the attributes of entities involved (equivalences); and the last part (also optional) defines initial values for attributes when an object is created (initialisers).

In VML-G, an inter_class definition is represented by a single icon (e.g. Figure 2-8, 2). This icon has three sections (invariants, equivalences, and initialisers) corresponding to the three sections of an inter_class definition. The other icon type (e.g. 3) denotes an entity taking part in the mapping with the inter_class. Each section of an inter_class icon often contains one or more rows, each of which has a symbol (in the centre) representing the type of mapping (e.g. equation, function, or procedure) being defined and a box (at each end of the row) to which the attributes and entities involved in the mapping are connected. Wiring from an attribute or entity to a box connects it into that equation.

2.4.4 TSF (Peltier et al)

[Peltier et al., 2000] from the University of Nantes described a model-driven approach, called Two-space Framework (TSF), to XML model transformation. The approach uses an abstract level for specifying transformation that is compiled into a lower, “concrete” XSLT-based level for implementation. The abstract language is declarative and textual. An example is given in Figure 2-9.



```
Attribute to Field {
attributes :
name=name,
visibility=visibility,
share = (ownerScope=="classifier" ? "static" : "none"),
state= (changeability=="frozen" ? "final" : "none")
}
```

Figure 2-9 (a) Part of a UML meta-model, (b) part of a Java meta-model and (c) their transformation

In Figure 2-9, there are two meta-models. The Attribute entity, contained in the source meta-model (a), corresponds to the Field entity, contained in the target meta-model. The transformation rules from Attribute to Field (in fact, from the source meta-model to the target meta-model; note that Attribute and Field are leaves of the inheritance trees they belong to) are expressed as an entity transformation, shown in (c). In general, an entity transformation details entities from the meta-models that take part in the transformation (e.g. 1) and correspondences between attributes in entities (e.g. 2).

2.4.5 Comparison of Related Approaches

Table 2-1 provides a comparison of typical examples of the related approaches, which have been noted previously.

Framework/ Tool	**Paradigm**	**Mapping Specification**	**Behaviour Specification**
MVC/ALV	Textual code	Powerful, but lacks abstractions specific to multi-view visual environments so that implementation of desired view consistency control logic is time-consuming	
JViews	Textual code	Powerful, but still requires detailed programming work and class framework knowledge	
JComposer	Visual notation-based, declarative (Mapping)/ flow-based (Behaviour)	(Mapping) Not obvious, limited mapping types	Less powerful than code but easier, still has high entry barrier for novices and non-experts
Pounamu	Visual form-based	limited, support simple 1:1 mappings only	Event handlers (code)
Kaitiaki	Visual notation-based, flow-based	Not applicable	Obvious, easier, (however) limited power and may be still hard to novices and non-experts
XSLT	Textual , declarative rule-based, Batch-oriented	Hard to use, limited to XML documents	Not applicable
TSF	Textual , declarative rule-based, Batch-oriented	Slightly easy to use than XSLT, also limited to XML documents	

VML	Visual notation-based, declarative, incremental	Easy to use, supports complex mapping types but limited to schemas	Not applicable

Table 2-1 Comparison of typical examples of the related approaches

From this comparison, we can see that:

- Textual-based approaches (e.g. MVC, ALV, JViews, XSLT, and TSF) are powerful (in a "do-it-yourself" sense) but suffer a certain degree of difficulty of use, depending on their domain specialisation. Some general-purpose approaches, such as MVC, ALV, and XSLT, tend to be more difficult to use than other special-purpose approaches, e.g. JViews and TSF. On the whole, there is a huge step from these approaches to visual-based approaches in terms of usability.
- Visual-based approaches (e.g. JComposer, Pounamu, Kaitiaki, and VML) generally provide easy-to-learn user interface for their domain tasks. Almost all of them use a generation approach from a specification model; many of them (e.g. JComposer, Kaitiaki, and VML) adopt domain-specific yet easy-to-understand visual notations for their specification tasks, providing even more easy-to-use facilities. However, these approaches typically limit the range of functions that they can provide to software developers. Thus, some of them (e.g. JComposer and Pounamu) also provide low-level facilities for complex cases.
- Flow-based approaches (e.g. the event specification of JComposer and Kaitiaki) use an event-flow metaphor for behaviour specification. These approaches are suitable for end users who have certain programming background, and provide quite flexible visual specification mechanisms when compared to declarative-based approaches. However, as view consistency control logic is generally complex, a simple behaviour specification created this way may require much more effort than it seems.
- Declarative-based approaches (e.g. view mapping specification of JComposer, XSLT, TSF, and VML) adopt a declarative nature for various specifications. Unlike procedural-based approaches, they try to hide as many implementation details as

possible, and are suitable for a large range of end users. However, these approaches typically limit the range of functions supported.

2.5 Summary

In this chapter, we provided an overview of the technologies involved in this thesis. First, we introduced various multi-view visual environments, such as software engineering and design tools, and their problem of view consistency. We then briefly described Pounamu, a meta-tool for building such visual environments, with a focus on its view consistency management (view-to-model mapping specification and behaviour specification). Finally, we addressed some of the related research that is similar to or provides some basis for the work in this thesis.

Our motivation is obvious. We want to provide sophisticated, easy-to-use view consistency mechanisms for the development of multi-view visual environments, using Pounamu as a prime example of such a development system.

Our solution plan for improving the view consistency management of Pounamu is as follows:

- Provide a high-level (visual), declarative approach to (Pounamu) view specification and generation
- Support a two-phase view implementation, the specification of view type models and the generation of view specification artefacts
- Design an appropriate visual notation for our view specification modelling
- Implement a modelling tool supporting that visual notation

The following chapters develop a visual notation and a corresponding tool for our view specification modelling.

CHAPTER 3

PRELIMINARY VSL DESIGNS

3.1 Introduction

In this chapter, we describe and illustrate the processes that we used to design a visual language for view specification. We first introduce our conceptual design of the language and highlight the issues that are likely encountered in practice. We then clearly explain our approach to addressing those issues, leading to a concrete design, which is, however, still in a preliminary state. Finally, we summarize our design experience, including both positive and negative issues, and our recommendations for the final design (which is presented in the next chapter).

3.2 Language Design Issues

It is obvious that the kind of view specification language that we are seeking would include certain graphical elements, which are used to represent meta-model entities and associations; view shapes and connections; mapping relationships relating view shapes/connections to meta-model entities/associations; and event handler-like components for complex view editing control and model constraint implementation. Figure 3-1 shows our preliminary conceptual design of this language.

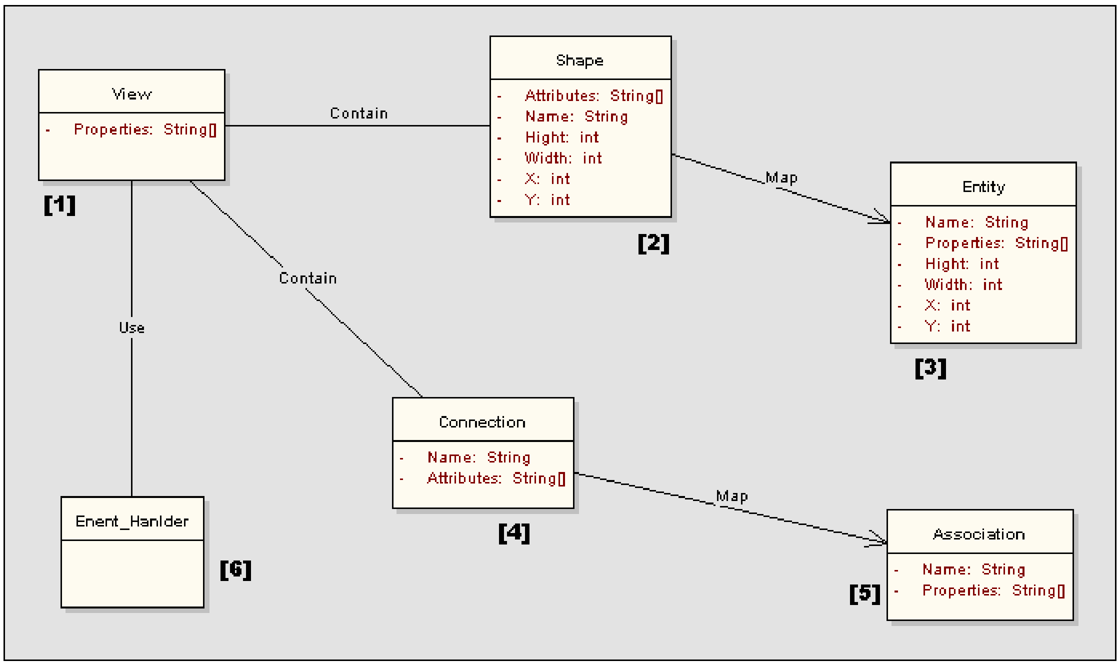


Figure 3-1 The conceptual design

In Figure 3-1, The View element (1) represents a view type being modelled; the Shape element (2) depicts a shape type, stating the kind of shapes allowed to use in the diagram; similarly, the Connection element (4) represents a connection type; the Entity element (3) depicts an entity type, defining the kind of data objects generated and maintained in the common repository; Similar to the Entity element, the Association element (5) represents an association type; and the Event Handler element (6) depicts a handler-like component.

Also shown in Figure 3-1, The View element (1) has “Contain” relationships with the Shape element (2) and the Connection element (4), which indicates that the view type represented by the View element can contain specific shape/connection types. This View element also has a “Use” relationship with the Event Handler element (6), which similarly states that the view type makes use of the handler component. The remaining are mapping relationships between the Shape/Connection and the Entity/Association. Such relationships denote how a data object (e.g. entities and associations) should be created and managed in the common repository according to a graphical object in the diagram.

While this design is a viable candidate, it leads to several issues in practice, especially for large and complex cases. These issues are described as follows:

(1) The language likely produces overly large diagrams, which contain too many graphical symbols and thus take too much space. The reason for this is that each shape/connection type is generally modelled using a pair of elements: a Shape/Connection with an Entity/Association. Try to imagine how many elements would be used in a single diagram for specifying a relatively small view type (say, containing ten different types of shapes/connections). Although multiple diagrams can be used (if the running environment support such an ability), the number of elements needed is still rather high per diagram with the addition of shape/connection types.

(2) This language uses an overly simplistic mechanism for specifying mappings between Shapes/Connections and Entities/Associations. Simple lines (with only two ends) might be enough to cope with one-to-one mappings but are impractical at representing many-to-many mappings or even one-to-many mappings.

(3) In addition, this approach also hides too much mapping information, especially at the property level. When a Shape/Connection is partially mapped to an Entity/Association (meaning that part of the Shape/Connection attributes correspond to part of the Entity/Association properties), the user can not tell the difference with when the Shape/Connection is fully mapped to the Entity/Association.

(4) As a high-level language (aiming at ease of view specification), it would need to incorporate certain advanced features, such as specifying visual relationships (geographic aspects) between two different shapes (e.g. one shape is restricted to visually contain the other).

(5) On the whole, this language must be easy for the modellers to work with for various specification tasks. Useful techniques may include: hiding and displaying specific graphic symbols in a diagram; completing certain tasks automatically; and making the language constructs rapidly understandable by readers.

The next section presents our approach to refining the initial conceptual model to deal with these issues.

3.3 Our Experiment

Much of the design effort focuses on the visual language design rather than the underlying implementation techniques (which are presented in Chapter 6). In the following subsections, we first introduce the preliminary design, an experiment to see whether Mapping Consoles can mitigate some of the problems noted early in this chapter. We then continue describing this design in practice using a typical view specification example.

The following section presents our judgment on and recommendations for this design.

3.3.1 The Preliminary Design

Figure 3-2 shows the notational elements that are created based on the conceptual design, which has been noted in the previous section.

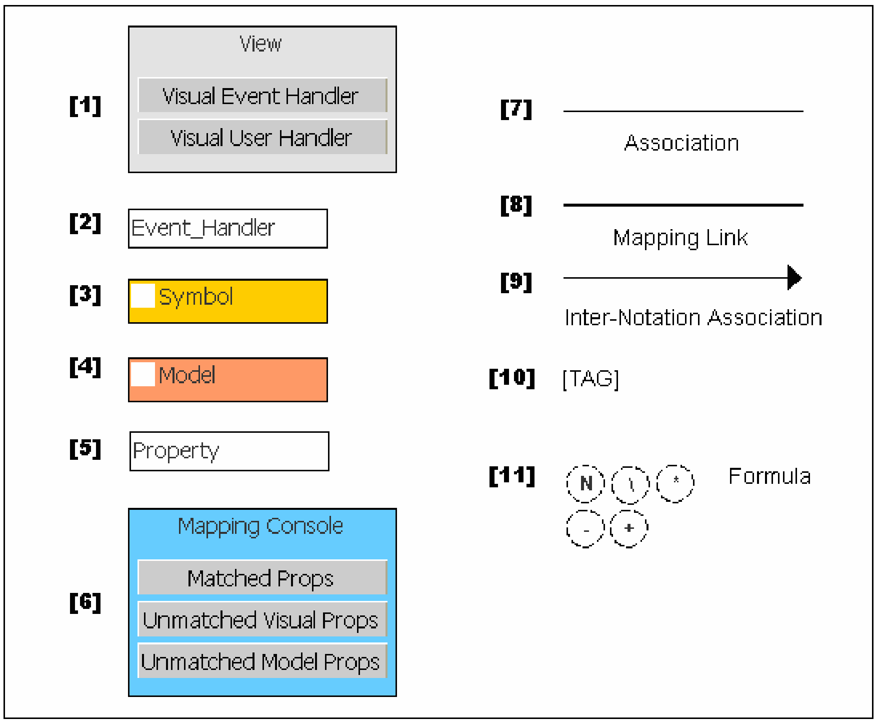


Figure 3-2 The preliminary notational elements

In Figure 3-2, the View element (1) depicts a view type being modelled; the Event Handler element (2) represents a handler-like component; the Symbol element (3) depicts a shape/connection type; the Model element (4) represents an entity/association type; the Property element (5) depicts an attribute/property (of a Symbol/Model); the Mapping Console element (6) is a view-to-model mapping coordinator, which is used between a Symbol and a Model; the Association element (7) represents a general-purpose relationship between two elements (e.g. a View with a Symbol, a Symbol with a Mapping Console, and a Model with a Mapping Console); the Mapping Link element (8) depicts a mapping relationship between two Properties; the Inter-notation Association element (9) represents a visual relationship (e.g. containment) between two Mapping Consoles; the Tag element (10) is a short piece of text (enclosed in a pair of square braces), which is used to attach an additional semantic to elements, such as for Symbols and Models (typical examples of Tags are shown in Table 1); and the Formula element (11) depicts a logic calculation, which is used between two Mapping Links for complex inter-property mappings (typical examples of Formulae are shown in Table 2).

Tag	Example	Remark
[TYPE]	[TYPE: Connection]	To further define a Model as a Connection type
[Multiplicity]	[Multiplicity: 0..*]	To set a restriction on the number of instances of a shape/connection types in the diagram
[Default]	[Default: “Computer Science”]	To set a default value of a property
[Null]	[Null: false]	To indicate whether or not a property can contain null value
[GoTo]	[GoTo: Class View]	To link a shape type to a specific view type

Table 2 Example of typical tags

Formula	Operation	Remark
⊕	Addition/ Catination	To sum up (incoming) numeric values or link string values together
⊖	Subtraction	To subtract one numeric value from another
⊛	Multiplication	To multiply one numeric value by another
⦸	Division	To divide one numeric value by another
Ⓝ	Null Detector	To detect whether or not a value is null

Table 3 Example of typical Formulae

In addition, some elements such as View, Symbol and Model can be used as containers enclosing other elements such as Event Handler and Property. Figure 3-3 shows such examples.

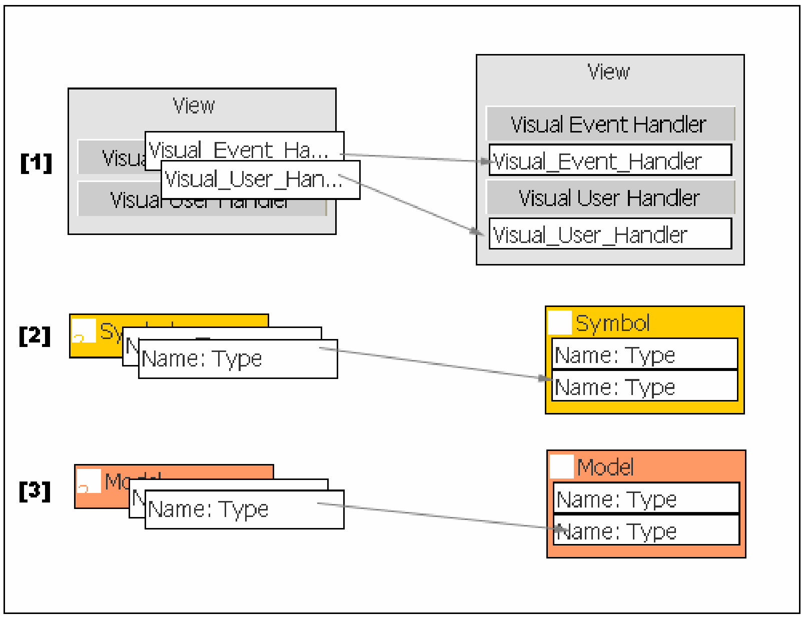


Figure 3-3 Example of the three container elements

As is shown in Figure 3-3, a View has two sections, “Visual Event Handler” and “Visual User Hander”. Each section can contain multiple Event Handlers of a specific type defined by the section name (1). Similarly, a Symbol (2) and a Model (3) can contain multiple Properties. (Arranging two symbols like this is an alternative way to state a “has a” relationship, in addition to using a connection.)

In fact, a Mapping Console can also act as a container, but its usage is a bit different to those described above. Mapping Consoles can not directly obtain Properties. They are used as coordinators between Symbols and Models in view-to-model mappings. Figure 3-4 shows a Mapping Console in use.

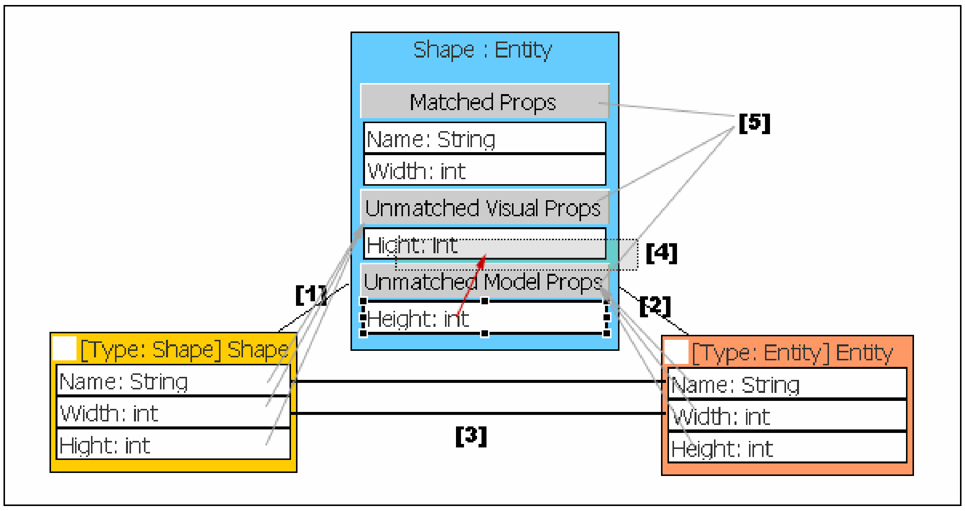


Figure 3-4 Example of using a Mapping Console

As is shown in Figure 3-4, after a Symbol/Model is connected to a Mapping Console through an Association (1) (2), the Mapping Console will copy the Properties of this Symbol/Model. Properties copied by the Mapping Console are categorized into different sections (5). There are two mechanisms to specify an inter-property mapping. One mechanism is to use a Mapping Link to connect one Property to another (3), whereas the other mechanism is to drag one Property and drop it on top of another Property (4) (at which point a Mapping Link will be generated and connected to proper Properties automatically). After two properties are matched, they will be eliminated from their current sections and a new Property will appear in the "Matched Props" section. (Note that the Tag "[Type: Shape]", which here is used to state that the Symbol represents a shape type, not a connection type.)

In the following subsection, we continue describing these elements in which they are used to specify a simple visual language.

3.3.2 Specifying a Visual Language

The visual language, as used in this example, is meant to provide a visual representation of a simple school model, which involves entities such as schools, classes, and students, and their relationships. This language defines two different but related view types: School View and Class View. Figure 3-5a shows a School diagram in use.

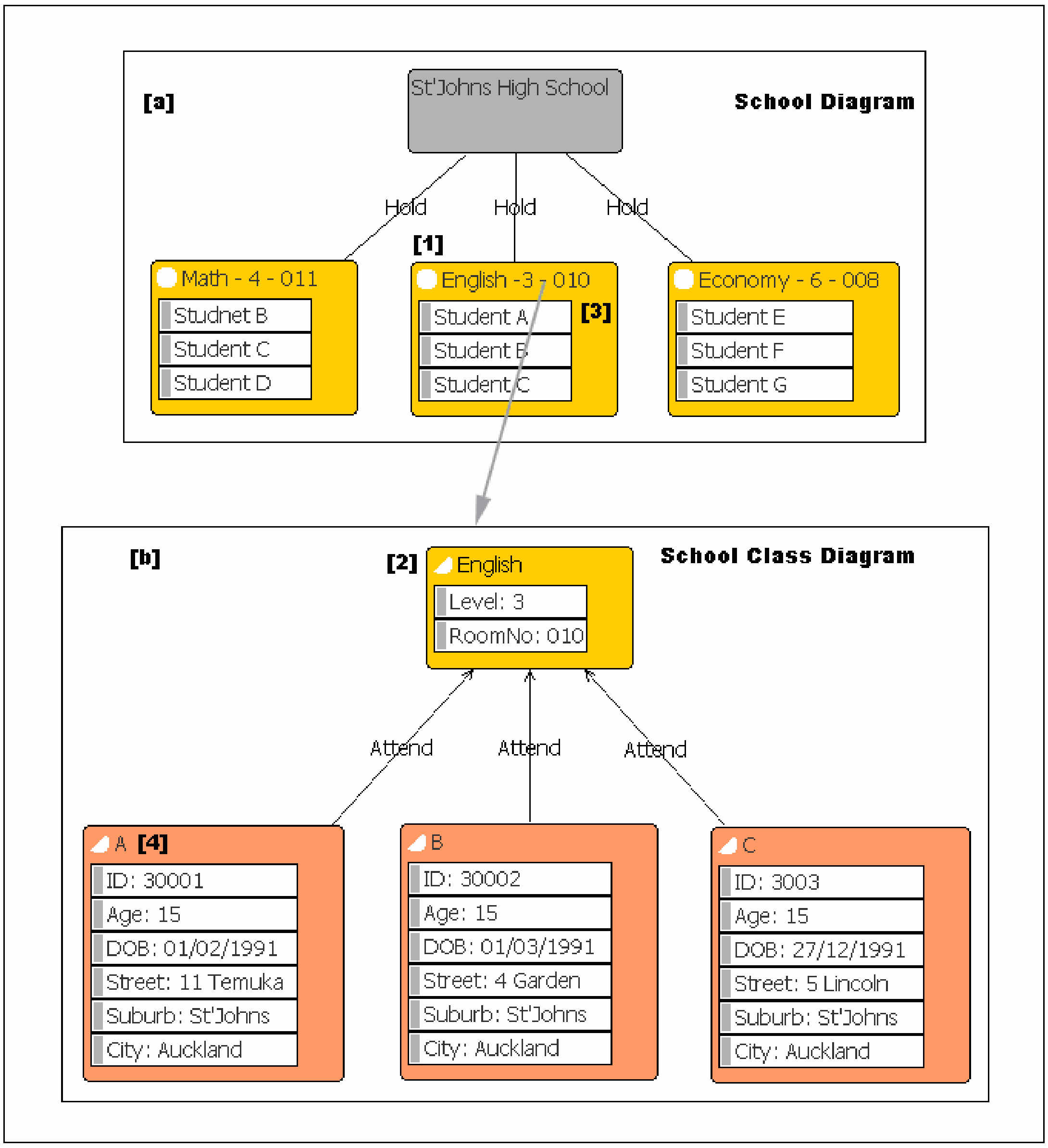


Figure 3-5 (a) A School diagram and (b) a related Class diagram

In Figure 3-5a, the school (St'Johns High School) is represented by the School element (top), which has "Hold" relationships with the three Class elements (Math, English, and Economy). There are several Student elements (e.g. Student A, Student B and Student C) enclosed inside their Class elements. In particular, the English class (1) is further illustrated in the (School) Class diagram that is shown in Figure 3-5b. Compared to Figure 3-5a, the Class element (2) here does not contain any Student elements but properties (e.g. Level and RoomNo). In fact, the Student elements (e.g. 3) that were contained in (1) become stand-alone elements (e.g. 4), which themselves can contain properties (e.g. ID, Age, DOB, or etc) in a Class diagram.

In the following, we illustrate how to specify this language using the notational elements presented previously. We first give a general description of the meta-model specification of this language, and we then demonstrate the School View specification and the Class View Specification, respectively.

The meta-model specification: Figure 3-6 shows the meta-model elements of this language. The Model attached with a type Tag of "E" (1) depicts a School entity type, which has a string property Name; the Model assigned with a type Tag of "A" (2) depicts an Association association type, which also has a string property Name; and (3)(4) are similar to (1).

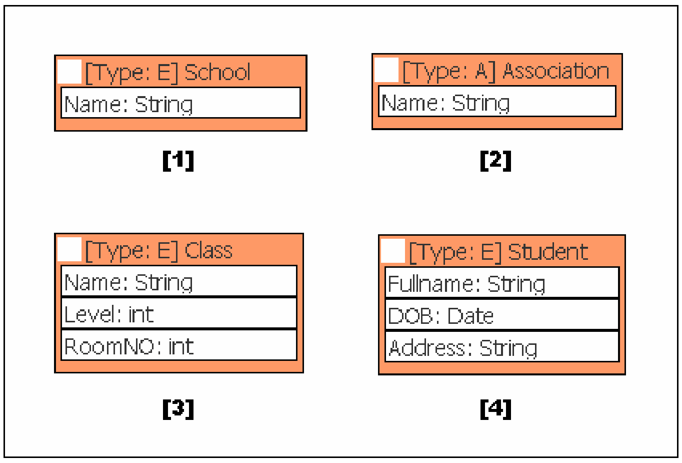


Figure 3-6 The meta-model specification

The School View specification: Figure 3-7 shows the School view type specification of this language.

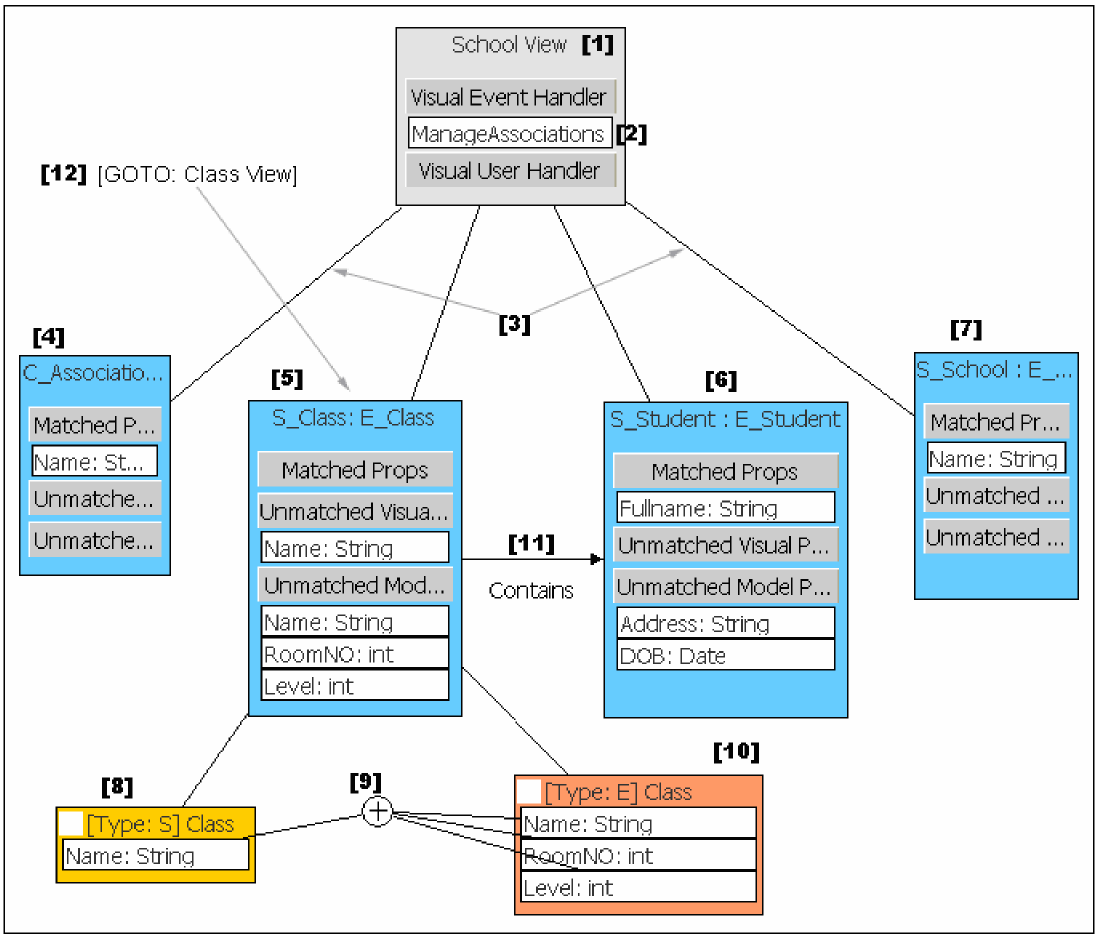


Figure 3-7 The School View specification

In Figure 3-7, the School view type itself is represented by the View (1), which contains the Event Handler (2) and is connected to several Mapping Consoles via the Associations (3). There are four Mapping Consoles, each of which is connected to a Symbol and a Model via Associations (e.g. 5). Once a Mapping Console is connected like this, the related Symbol and Model can be hidden in the background manually (e.g. 4, 6, and 7). After that, inter-property mappings can be specified – even when related Symbols and Models are invisible (e.g. 4, 6, and 7). In particular, (5) is connected to (6) via the Inter-notation Association (11), which states that the Class Symbol visually contains the Student Symbol (e.g. as shown in Figure 3-5a). In addition, (5) is also assigned with a

GOTO Tag of "Class View" (12), which specifies the Class symbol can be further illustrated in (navigated to) a Class diagram (e.g. as shown in Figure 3-5b).

The Class View specification: Figure 3-8 shows the Class view type specification of this language.

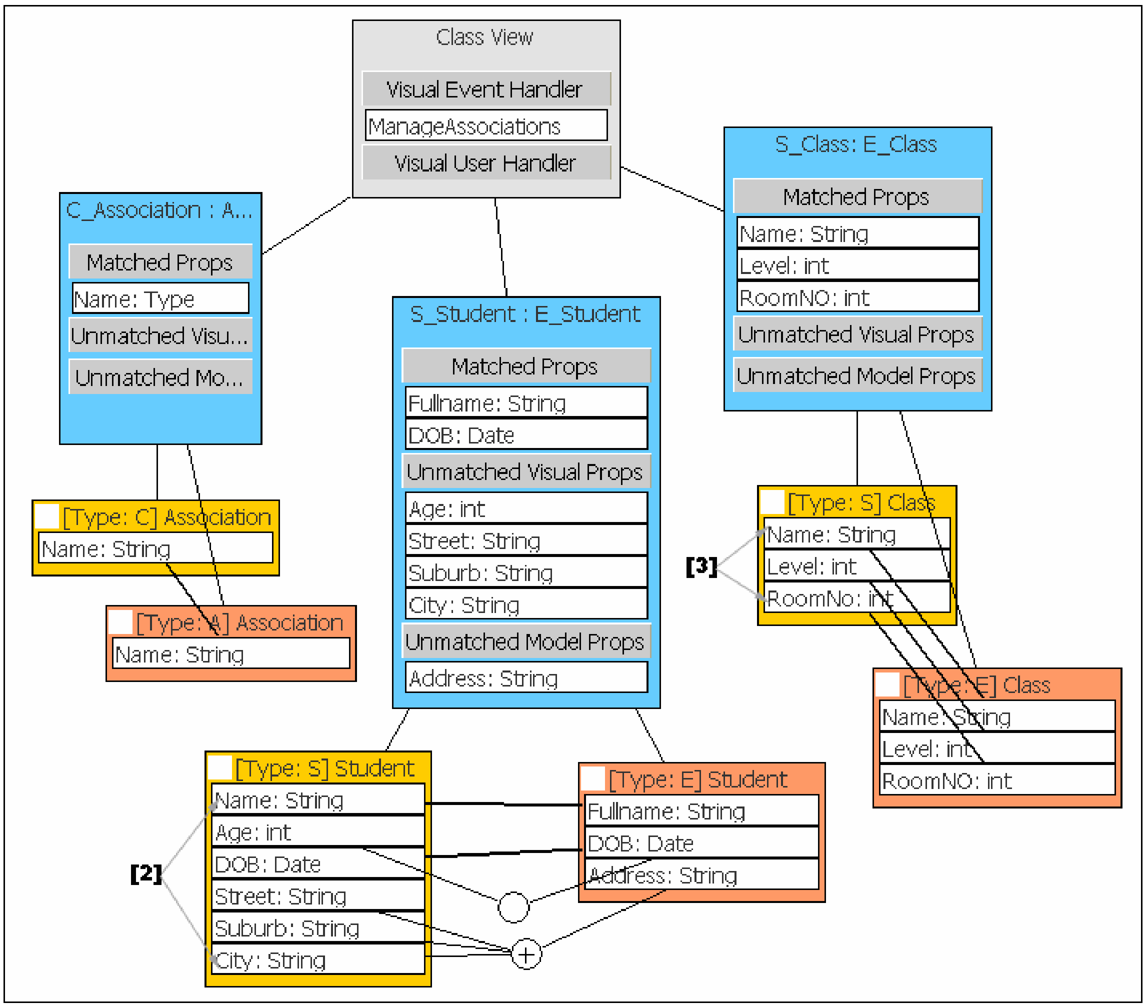


Figure 3-8 The Class View specification

Compared to Figure 3-7, there are only three Mapping Consoles, and we have chosen to show their related Symbols and Models (rather than eliding them) in Figure 3-8. In addition, The Student Symbol (2) and the Class Symbol (3) contain more visual properties, which are matched with corresponding model properties. This enables entities, such as classes and students, to be rendered differently between School diagrams and related Class diagrams (as you can see by comparing Figure 3-5a to Figure 3-5b). The remaining elements are the same as is presented in Figure 3-7.

3.4 Post-design Notes – Experience Learned

Our notes are separated into three subsections. We first present positive attributes of the preliminary design, and follow this by negative attributes. After that, we finish the discussion by expressing some improvement ideas for the design. These are used as a basis for the final design described in the next chapter.

3.4.1 Positive Attributes

Mapping Consoles can be used to visually divide view specification diagrams into localized areas, each of which presents a view-to-model mapping involving related Symbols and Models (e.g. as shown in Figure 3-4 and 3-8). This enables the modeller to focus on a small area and to elide unnecessary Symbols and Models in the diagram as needed (e.g. Figure 3-7). Mapping Consoles also make inter-property mappings easier alone with drag and drop.

Mapping Links are intuitive to modellers. The specification of mapping at the property level makes view specifications more obvious. This is because that a modeller can clearly see which properties are matched and how these properties are mapped each other (e.g. Figure 3-4, 3-7, and 3-8). In addition, wiring with Formulae enables the specification of complex inter-property mappings (e.g. Figure 3-7 and 3-8).

Inter-notation Associations give modellers high-level support to enrich their language design with easy-to-use representation power (e.g. Figure 3-5 and 3-7). They are reasonable easy to use. More importantly, they hide many specification details modellers would otherwise have to do on them own.

Tags are quite handy for various small specification purposes. They are useful when the use of graphic symbols is ineffective (e.g. making things obvious). As pieces of textual information, they are highly applicable, extendable and biddable; they can be attached to any object as simple sting fields, each of which itself is able to contain multiple Tags.

3.4.2 Negative Attributes

Mapping Consoles sometimes make diagrams more complicated than they need to be (e.g. Figure 3-8); they double up certain parts of the information presented in the diagram (e.g. Figure 3-7 and 3-8) and are often too big to be used frequently (the more compact the diagram is, the easier it is for the modeller to draw, maintain, and comprehend). More importantly, the use of Mapping Consoles does not cope with complex (e.g. many-to-one, one-to-many, and many-to-many) view-to-model mappings: in these cases, hiding related Symbols and Models also screen some significant information, in which the Property holders (e.g. Symbols and Models) are lost (if additional information is provided, Mapping Consoles become even bigger). In fact, Mapping Consoles are non-structural constructs, which do not contribute to the structure of view type models; therefore, they are optional (learning how to use Mapping Consoles may take some time on top of others).

Mapping Links (as lines) are too simple to represent complex inter-property mappings, especially the mappings that involve multiple Properties on either or both sides. More importantly, too many Mapping Links blur other types of connections in the diagram (e.g. Figure 3-7 and 3-8). The use of Formulae makes the diagram even harder to understand, as more lines and circles are used graphically. In addition, although drag and drop is useful sometimes, it is ineffective in complex cases (e.g. many-to-many mappings) and takes too much effort to implement.

Inter-notation Associations are inexpressive e.g. what the “Contains” in Figure 3-7 implies can not be perceived. They also lacks at dealing with more than two Mapping Consoles, as they are just simple lines with two arrow heads. More importantly, they are not customizable; therefore, it is impossible for the modeller to make their own variations (after all, the requirements are often vary from case to case).

Combining many Tags together in a single field would unexpectedly enlarge the size of the symbol (e.g. Figure 3-4, 4-7, and 4-8). Most importantly, too many Tags used in a diagram would cause the graphical symbols contained to be less obvious (this is highly against the principle of making a good visual language).

3.4.3 Future Improvements

Based on the positive and negative attributes discussed above, we propose to make the following improvements to the final design:

First, our experimental introduction of Mapping Consoles has shown them to be an ineffective mechanism. For complex view-to-model mappings, a simple and intuitive mechanism is desired. In addition, while practicing with this initial design, we also realized that many-to-many view-to-model mappings are particularly hard to deal with (and almost impossible to achieve for implementation). Therefore, an effective mechanism needs to break such complex mappings into simple alternatives as well.

Second, the abstractions used, such as symbols and models, led to some difficulties for users. A diagram containing a large number of similar symbols differentiated by textual information is certainly hard for readers to understand (e.g. it is difficult to distinguish between a shape type and a connection type, between an entity type and an association type, and between an Event Handler and a Property). Therefore, clear visual distinctions need to be made among those similar elements.

Third, the use of too many Mapping Links would similarly make the diagram difficult to understand. This difficulty would be even worse when many Formulae are also used. Thus, we need a revised mechanism to indicate the mapping directionality, to hide and display Mapping Links as needed, and, more importantly, to better support advanced inter-property mappings (e.g. many-to-one and many-to-many).

And so forth, we feel that Inter-notation Associations are too simple and thus inexpressive. What is important is that they are not customizable and extendable. These serious issues need to be addressed along with the underlying implementation techniques.

On the whole, we feel that some other minor improvements (e.g. polishing the language) would need to be made, so that the language is reasonably terse, easier to understand, and convenient to use.

3.5 Summary

This chapter has presented our language design processes. It started by introducing the conceptual design of this language and highlighting its issues of use. Following this, it provided the concrete design that takes into account of those issues, with a typical view specification example showing this concrete design in practice. Finally, it focused on the experience that we learned during the whole design course, leading to the improvements that can be made in the final design.

In the next chapter, we present our final design of the View Specification Language (VSL), which builds from the lessons gained in the design experiments undertaken in this Chapter.

CHAPTER 4

VSL NOTATION SPECIFICATION

4.1 Introduction

In this chapter, we provide a deep technical view about the View Specification Language (VSL). We start by introducing VSL, briefly explaining what the language attempts to achieve, how it can be used, and what kind of notation it defines. Following this, we will in turn discuss *Structural Elements*, *Mapping Elements*, *Constraint Elements*, and *Artefact Elements* that are contained in the VSL notation.

In order to help you clearly understand the abstract concepts presented in this chapter, we use the specification of a simple Use Case view type (similar to the one defined in UML, Unified Modelling Language [UML, 2000]) through all the sections to illustrate the basic features of the language.

4.2 Overview

The **V**iew **S**pecification **L**anguage (VSL) is a high-level, declarative visual language designed to facilitate the effort of view specification and generation in developing multi-view visual environments. In doing so, it defines a View Specification Diagram (VSD), which is based on a box-and-line diagramming technique tailored for creating graphical models of view types (definitions). A view specification model, then, is a network of graphical objects, which are constructs (e.g. *Shapes*, *Connections*, *Entities*, and *Associations*), the mappings that define their data relations, and the constraints that define their syntactical and semantical behaviour.

When specifying a view type, a user of VSL initially defines a structural specification denoting visual elements (e.g. *Shapes* and *Connections*) and model elements (e.g. *Entities* and *Associations*). The user then maps the visual elements with their corresponding model elements. Finally, the user applies constraints to necessary elements and generates a set of view specification artefacts (e.g. files, code, or both), which can then be used in a target running system for creating view instances (diagrams) as specified.

VSL provides a visual notation containing various graphical elements that describe a view (diagram) type. These elements are distinguishable from each other and familiar to many end users (using a common iconic form). Some elements are similar in shape but are further differentiated by secondary factors such as colour, icon or both (similar elements often have comparable uses in modelling). Almost all the elements have their own properties: some properties are visually displayed on the screen, whereas others are hidden behind the scenes (accessible through platform-specific property sheets). Based on their use in the different stages mentioned above, all the elements are divided into four functional categories. These are as follows:

- Structural Elements
- Mapping Elements
- Constraint Elements
- Artefact Elements

In the following sections, specification of a Use Case diagram type is used to describe and illustrate the process of designing a view specification using VSL.

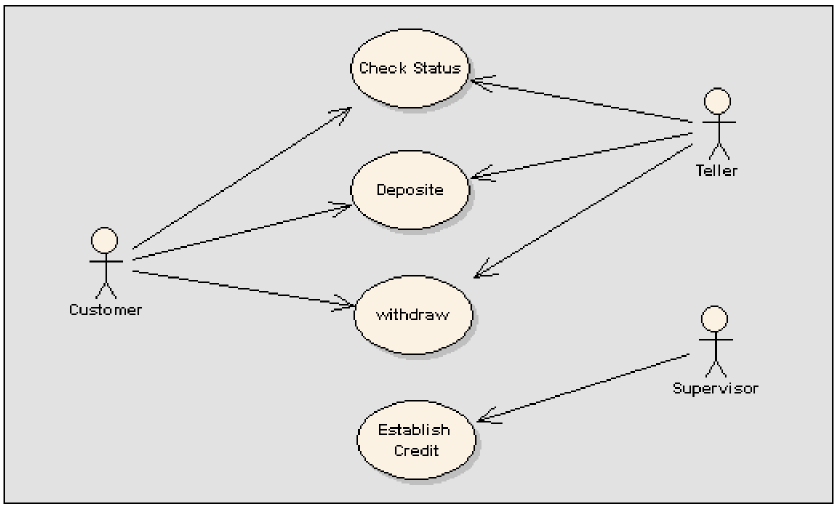


Figure 4-1 A Use Case diagram

Figure 4-1 shows the Use Case diagram in use. Use Case diagrams contain actors (e.g. Customer, Teller, and Supervisor) and use cases (e.g. Check Status, Deposit, Withdraw, and Establish Credit). Actors employ use cases according to their specific roles. Although this Use Case example is simple, showing only three notational elements, it serves to illustrate the basic uses of VSL.

4.3 Structural Elements

In VSL, *Structural Elements* can be used to define a structural specification of a view type being modelled. Such a specification generally describes various visual elements (e.g. shape/connection types), model (data) elements (e.g. entity/association types) and handler-like (programming) components (e.g. visual handlers).

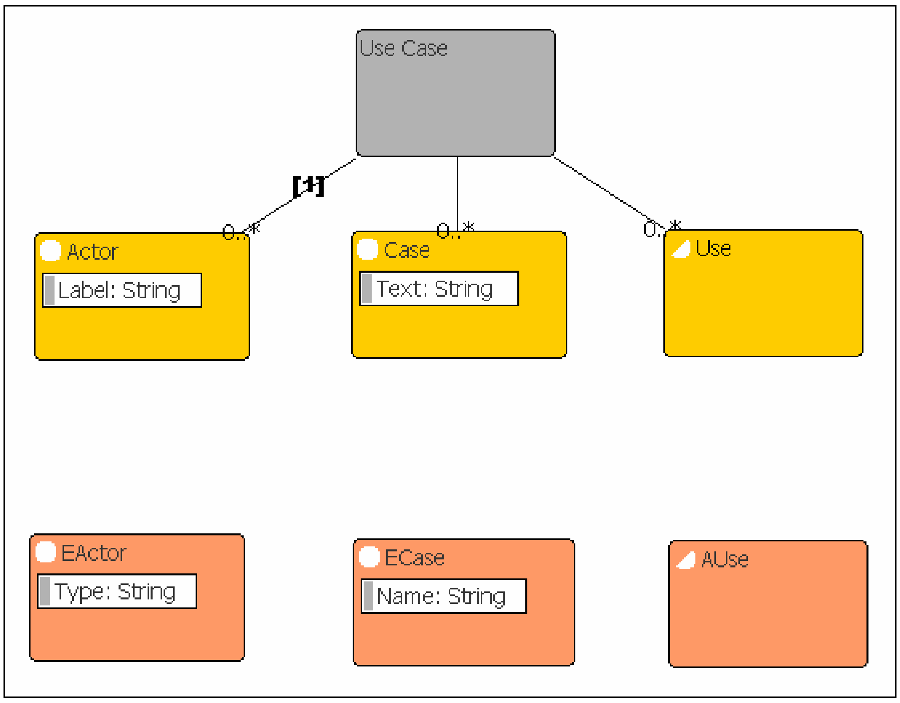


Figure 4-2 Structural specification of the Use Case view type

Figure 4-2 shows a structural specification of the Use Case view type that defines diagrams, such as the one shown in Figure 4-1. The view type itself is represented by the *View* (Use Case), which has one-to-many relationships to the *Shapes* (Actor and Case) and the *Connection* (Use) through the *General Associations*, such as (1). Also showing are the *Entities* (EActor and ECase) and the *Association* (AUse), and they are unconnected to any others at this stage. The remaining are the *Properties* (Label, Text, Type, and Name) contained in their corresponding containers.

The detailed specifications of the six *Structural Elements* are as follows:

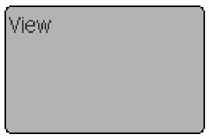
View

A *View* represents a view type being modelled; it can be connected to or from a *Shape*/*Connection* through a *General Association*, which indicates that the *View* has the *Shape*/*Connection*; it can act as a container that visually contains multiple *Visual handlers*, which indicates that the *View* has the *Visual handlers*.

(There are two mechanisms used in VSL to indicate a "has a" relationship between two shapes. One is to connect a shape to another through a proper connection. The other is to drop an item shape on top of a container shape, at which point the system will automatically put the item shape in place.)

A *View* is represented as a rounded-corner rectangle with a label indicating the *View* name on the top (see the right-hand figure below). The properties of *View* are listed as follows:

- Key
- Name
- Description



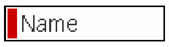
For a *View*, the first property, "Key", denotes the *View* identification, which is usually generated by the system automatically. The second property, "Name", states the *View* name, which is used as the name of the view type that this *View* represents. The last property, "Description", contains the *View* comments, which are optional and may be used as tool tips in running systems.

Visual Handler

A *Visual Handler* represents a handler-like component that is used by a view type; it can act as an item visually contained in a *View*, which indicates that the *Visual handler* is a part of the *View.*

A *Visual Handler* is represented as a narrow rectangle with a quadrate red icon and a label indicating the *Visual Handler* name on the left (see the right-hand figure below). The properties of *Visual Handler* are listed as follows:

- Key



- Name
- Type

For a *Visual Handler*, the first property, “Key”, denotes the *Visual Handler* identification, which is usually generated by the system automatically. The second property, “Name”, states the *Visual Handler* name, which is used as the name of the class that implements this *Visual Handler*. The last property, “Type”, indicates the *Visual Handler* type, which can be “User” or “Event”. A user *Visual Handler* has to be triggered manually, whereas an event *Visual Handler* is activated automatically after the monitored events have been fired.

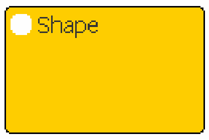
Shape

A *Shape* represents a shape type that is contained in a view type; it can be connected to or from a *View* through a *General Association*, which indicates that the *Shape* is a part of the *View*; it can act as a container that visually contains multiple *Properties*, which states that the *Shape* has the *Properties*.

(You may question our intention behind representing *Properties* as stand-alone shapes, because this would be generally achieved differently in many other visual modelling languages such as UML. We will later describe this intention when addressing *Property*.)

A *Shape* is represented as a rounded-corner rectangle with a round white icon and a label indicating the *Shape* name on the top (see the right-hand figure below). The properties of *Shape* are as follows:

- Key
- Name



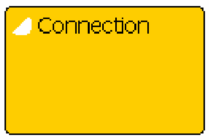
For a *Shape*, the first property, “Key”, denotes the *Shape* identification, which is usually generated by the system automatically. The last property, “Name”, states the *Shape* name, which is also the name of the shape type that this *Shape* represents.

Connection

A *Connection* represents a connection type that is contained in a view type; it can be connected to or from a *View* through a *General Association*, which indicates that the *Connection* is a part of the *View*; it can act as a container that visually contains multiple *Properties*, which states that the *Connection* has the *Properties*.

A *Connection* is represented as a rounded-corner rectangle with a white semi-circle icon and a label indicating the *Connection* name on the top (see the right-hand figure below). The properties of *Connection* are as follows:

- Key
- Name



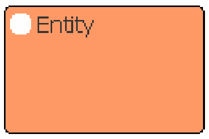
For a *Connection*, the first property, “Key”, denotes the *Connection* identification, which is usually generated by the system automatically. The last property, “Name”, states the *Connection* name, which is also the name of the connection type that this *Connection* represents.

Entity

An *Entity* represents an entity type that usually corresponds to a shape type behind the scenes; it can act as a container that visually contains multiple *Properties*, which indicates that the *Entity* has the *Properties*.

An *Entity* is represented as a rounded-corner rectangle with a round white icon and a label indicating the *Entity* name on the top (it is distinguished from a *Shape* by having a different interior colour, as shown in the right-hand figure below). The properties of *Entity* are as follows:

- Key
- Name



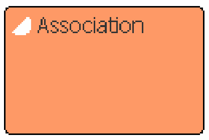
For an *Entity*, the first property, "Key", denotes the *Entity* identification, which is usually generated by the system automatically. The last property, "Name", states the *Entity* name, which is also the name of the entity type that this *Entity* represents.

Association

An *Association* represents an association type that usually corresponds to a connection type behind the scenes; it can act as a container that visually contains multiple *Properties*, which indicates that the *Association* has the *Properties*.

An *Association* is represented as a rounded-corner rectangle with a white semi-circle icon and a label indicating the *Association* name on the top (it is distinguished from a *Connection* by having a different interior colour, see the right-hand figure below). The properties of *Association* are as follows:

- Key
- Name



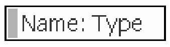
For an *Association*, the first property, "Key", denotes the *Association* identification, which is usually generated by the system automatically. The last property, "Name", indicates the *Association* name, which is also the name of the association type that this *Association* represents.

Property

A *Property* represents a property or attribute of a shape/connection/entity/association type; it can act as an item visually contained inside a *Shape/Connection /Entity/Association*, which indicates that the *Property* belongs to the *Shape/Connection/ Entity/Association*. *Properties* are designed as stand-alone shapes; thus, a connection can be used explicitly between one *Property* and another. (This is needed in VSL for specifying inter-property mappings.)

A *Property* is represented as a narrow rectangle with a quadrate grey icon and a label indicating the property name and type on the left (see the right-hand figure below). The properties of *Property* are as follows:

- Key
- Name
- Type



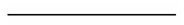
For a *Property*, the first property, "Key", denotes the *Property* identification, which is usually generated by the system automatically. The second property, "Name", states the *Property* name, which is also the name of the property this *Property* represents. The last property, "Type", indicates the data type of the property.

General Association

A *General Association* can be used between a *View* and a *Shape*/*Connection*, which states that the *View* contains the *Shape*/*Connection*.

A *General Association* is represented as a solid line (see the right-hand figure below). The properties of *General Association* are as follows:

- SMultiplicity
- Name
- TMultiplicity



For a *General Association*, the first property, "SMultiplicity", denotes the source multiplicity. The second property, "Name", presents the name of the *General Association*. The last property, "TMultiplicity", indicates the target multiplicity, which can be used to constrain the number of instances of the target shape/connection allowed in a diagram (see an example in Chapter 5, Section 2).

4.4 Mapping Elements

As noted in chapter two, multiple views map to a common repository in multi-view visual environments. Each of these views provides a partial representation of the repository and contributes to the construction of the repository model. Behind the scenes, this is achieved by specifying how the graphical symbols contained in the view relate to their underlying model data and vice versa. Such mapping relationships are often quite complicated. For example, a symbol may be created with external values passed in, may be associated to multiple different repository entity/association instances, or not be physically related to any such instances at all. Advanced mapping mechanisms are thereby needed.

In VSL, *Mapping Elements* can act on *Structural Elements* to form mapping chains, which are routes that are connected by *Mapping Elements*. A full mapping chain first starts at an *Input*, after which it is passed to a *Shape/Connection*. Then it is transmitted to one or more *Entities*/*Associations* and finally it is either stopped where it is or returned to the *Shape*/*Connection* where it ends. With the different *Mapping Elements* used in a mapping chain, the *Structural Elements* are mapped together as needed.

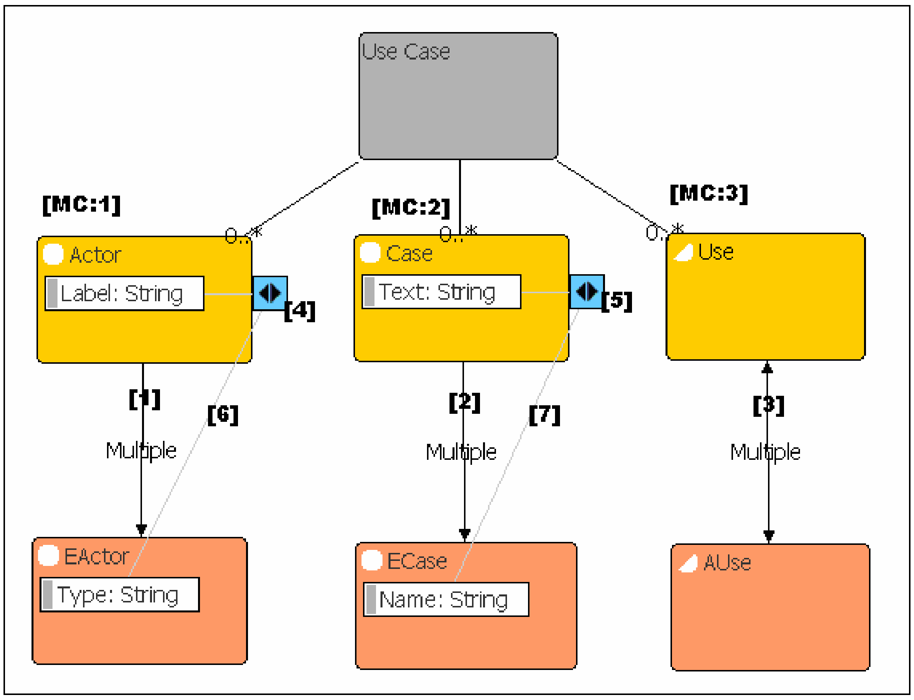


Figure 4-3 Mapping specification of the Use Case view type

Figure 4-3 shows a mapping specification of the Use Case view type on top of the structural specification denoted in Figure 4-2. There are three mapping chains created in Figure 4-3. The first (MC: 1) is initialised at the *Shape* (Actor) and then passed to the *Entity* (EActor) through the "multiple" *Generation Flow* (1), which indicates that an EActor entity is generated in the repository whenever an Actor shape is drawn in the diagram. Similarly, the second (MC: 2) is routed from the *Shape* (Case) to the *Entity* (ECase) via the "multiple" *Generation Flow* (2). Then the last (MC: 3) is started at the *Connection* (Use) and ended with the *Association* (AUse) through the bidirectional "multiple" *Generation Flow* (3), which states that a Use connection is generated in the diagram if an applicable AUse association is created in the repository, and vice versa. Compared to Figure 4-2, the model objects (EActor, ECase, and AUse) in Figure 4-3 have now been mapped to their corresponding visual objects (Actor, Case, and Use).

Also shown in Figure 4-3 are two inter-property mappings. In the first mapping (left), the visual property (Label) is mapped to the model property (Type) via the bidirectional *Property Mapper* (4) connected by two *Mapping Associations*, such as (6), which indicates that changes made to the Label apply to the Type and vice versa. In the second mapping (middle), the visual property (Text) is mapped to the model property (Name) through the bidirectional *Property Mapper* (5) connected by two *Mapping Associations*, such as (7).

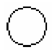
The detailed specifications of the six *Mapping Elements* are as follows:

Input

An *Input* can be used to connect to a *Shape/Connection* through a *Generation Flow,* which indicates the direct consumer of this *Input*. In fact, the parameters defined in the *Input* are not only accessible by its direct consumer, but also the mapping chain that is carried out from that consumer. *Inputs* are needed for using external values *e.g.* creating the objects of a specific class type in UML (the class type has to be known prior to the creation). (See an example in Chapter 5, Section 3.)

An *Input* is represented as a circle (see the right-hand figure below). The properties of *Input* are as follows:

- Key
- Handler
- ParameterProfiles



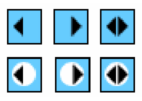
For an *Input*, the first property, “Key”, denotes the *Input* identification, which is usually generated by the system automatically. The second property, “Handler”, indicates the name of the handler class that is responsible for gathering the *Input* values (e.g. such a handler class instance may pop up an interface allowing the diagram user to fill in the parameter values, and then stores them somewhere that can be queried later in the following processes). The last property, "ParameterProfiles", contains the collection of the parameter definitions, which states the parameter names (all the parameter names in VSL begin with "?" to differentiate from the other textual identifications) and their value types.

Property Mapper

A *Property Mapper* can be connected to or from a *Property* through a *Mapping Association*; it can be visually attached to a *Shape*/*Connection*/*Entity*/*Association* to form a visual group, in which all the components contained move as a whole. A *Property Mapper* acts as a coordinator that helps support the visual-to-model property mappings. Because the presence of *Property Mappers* is potentially very frequent in a complex VSD – even in a simple VSD – *Property Mappers* are kept as small as possible to ensure the compactness and cleanliness of VSDs while still presenting useful visual clues.

A *Property Mapper* is represented as a small square with an icon in the centre, and the icon varies depending on the type (arrow head) and formula (circle) of this *Property Mapper* (see the right-hand figure below). The properties of *Property Mapper* are as follows:

- Key
- Type
- Formula



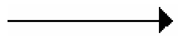
For a *Property Mapper*, the first property, "Key", denotes the *Property Mapper* identification, which is usually generated by the system automatically. The second property, "Type", states the *Property Mapper* type, which can be "V->M", "V<-M", and or "V<->M" (respectively mapped from/to visual properties to/from model properties or bi-directionally between them). The last property, “Formula”, contains the logical calculation (which can be represented by OCR) from and to the visual and model properties that relate to this *Property Mapper*.

Generation Flow

A *Generation Flow* can be used to connect an *Input* to a *Shape/Connection*, which indicates a “generation” relationship between the *Input* and the *Shape/Connection*; it also can be used to connect a *Shape* to an *Entity* and a *Connection* to an *Association*. A *Generation Flow* is a flow object that can pass a signal alone the mapping chain. In a mapping chain, a signal is passed from the start of a flow object to its end and then taken by the other flow objects that follow. Eventually it goes through the mapping chain to accomplish all the mapping specifications. The term “generation” is meant to indicate that the signal passing through a *Generation Flow* creates instances of the shape/connection/entity/association type. (See an example in Chapter 5, Section 3.)

A *Generation Flow* is represented as a solid line with a solid arrowhead (see the right-hand figure below). The properties of *Generation Flow* are as follows:

- Direction
- Type



For a *Generation Flow*, the first property, “Direction,” denotes the *Generation Flow* orientation, which can be “Unidirectional” or “Bidirectional.” A unidirectional *Generation Flow* indicates that the mapping signal is only passed from the start of this *Generation Flow* to its end, whereas a bidirectional *Generation Flow* declares that the mapping signal is also passed from the end to the start. The last property, “Type” states the *Generation Flow* type, which can be “Unique” or “Multiple”. A unique *Generation Flow* implies that only one instance of the target type, to which the *Generation Flow* connects, is generated for all the instances of the source type, whereas a multiple

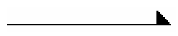
Generation Flow means that an instance of the target type is generated whenever an instance of the source type is created.

Search Flow

A *Search Flow* can be used to connect a *Shape* to an *Entity*, which indicates a "search" relationship between the *Shape* and the *Entity*; it can also be used to connect a *Connection* to an *Association*, but the usage is considered less practical. A *Search Flow* is a flow object, similar to a *Generation Flow*. The term "Search" is meant to indicate that the signal passed through a *Search Flow* results in looking up instances of the entity/association type in the repository. (See an example in Chapter 5, Section 3.)

A *Search Flow* is represented as a solid line with a half solid arrowhead (see the right-hand figure below). The properties of *Search Flow* are as follows:

- AttributeProfiles
- NOI
- AutoCreated



For a *Search Flow*, the first property, "AttributeProfiles", contains the search criteria, which is used to filter all the instances of the target type that this *Search Flow* connects to. The second property, "NOI", indicates the maximum number of the instances that can be found by this *Search Flow*. The last property, "AutoCreated", holds a Boolean value, which indicates whether or not an instance, as specified in the search criteria, should be created if the search has not found any records.

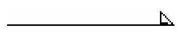
Reference Flow

A *Reference Flow* can be used to connect a *Shape* to an *Entity*, which indicates a "reference" relationship between the *Shape* and the *Entity*; it can also be used to connect a *Connection* to an *Association*, but the usage is considered less practical. A *Reference*

Flow is very similar to a *Search Flow* because they share the same *Structural Element* set that they can connect with. Nevertheless, instead of searching, a *Reference Flow* uses the (external) value that is obtained through its InputParameter. (See an example in Chapter 5, Section 2.)

A *Reference Flow* is represented as a solid line with a half open arrowhead (see the right-hand figure below). The property of *Reference Flow* is as follows:

- InputParameter



For a *Reference Flow*, the property, "InputParameter", states the name of the input parameter that is defined in the *Input* that exists in the same mapping chain as this *Reference Flow*.

Mapping Association

A *Mapping Association* can be used to connect a *Property* to or from a *Property Mapper*, which indicates a "mapping" relationship between the *Property* and the *Property Mapper*. *Mapping Associations* are specifically signified in grey to differentiate them from the other types of connections, and can be hidden or displayed manually (together in a VSD).

A *Mapping Association* is represented as a solid grey line (see the right-hand figure below). There are no properties for a Mapping Association.

4.5 Constraint Elements

Constraints are another kind of handler-like component established specifically in VSL (in addition to *Visual handlers)* to simplify the process of building view specification models. In fact, a *Constraint* can be viewed as a sophisticated *Visual Handler*, which is simple, useful, and re-useable. For example, *Constraints* can be used directly without any programming work, and they address the common needs that are required in the development of various multi-view visual environments.

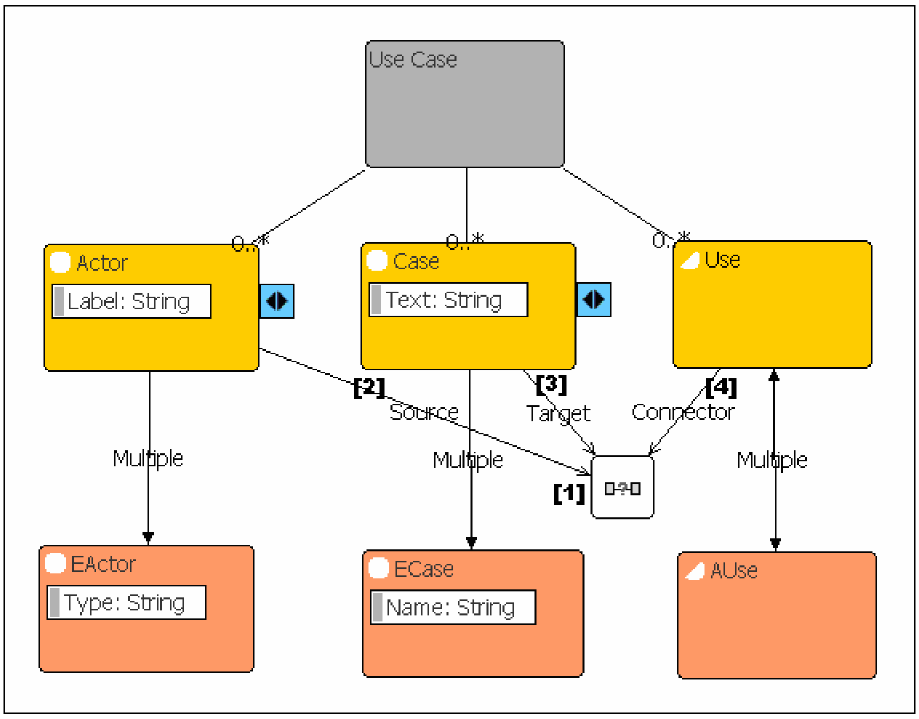


Figure 4-4 Constraint specification of the Use Case view type

Figure 4-4 shows a constraint specification of the Use Case view type on top of the mapping specification illustrated in Figure 4-3. The *Constraint* (1) is joined by the *Shape* (Actor) through the *Participation Link* (2) assigned with a role of “Source”. The *Constraint* (1) is also joined by the *Shape* (Case) through the *Participation Link* (3) assigned with a role of “Target”. In addition, The *Constraint* (1) is joined by the *Connection* (Use) through the *Participation Link* (4) assigned with a role of “Connector”. This indicates that Actor can connect to Case through Use, but not the other way around.

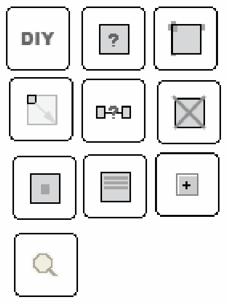
The detailed specifications of two Constraint Elements are as follows:

Constraint

A *Constraint* can be connected to a *Shape/Connection* through a *Participation Link*, which indicates the *Shape/Connection* must obey the force or rule that is defined by the *Constraint*. In general, a force can be as simple as keeping diagram components visually together or as complex as switching representations of those components according to the zooming level.

A *Constraint* is represented as a rounded-corner square with a centre icon varying on the type of the *Constraint* (see the figure below to the right). The properties of *Constraint* are as follows:

- Key
- Type
- Participants
- Parameters
- Extended



For a *Constraint*, the first property, “Key”, denotes the *Constraint* identification, which is usually generated by the system automatically. The second property, “Type”, indicates the *Constraint* type, which is further detailed below. The third property, “Participants”, contains the collection of participants that participate in this *Constraint*. The fourth property, “Parameters”, holds the parameter values, and is used to define this *Constraint*. The last property, “Extended”, is a Boolean value that states whether or not the default (underlying) implementation of the *Constraint* is used (if not, the user has to implement it on his/her own).

In VSL, there are various *Constraints* that can be used by modellers to solve common problems. Based on the functionality intended, all the *Constraints* are mapped to specific types as follows:

Custom: a custom *Constraint* should be used if all other types of *Constraints* are inapplicable in the situation. It is meant to indicate that the user has to implement the *Constraint* on his/her own.

Creatable: a creatable *Constraint* can be used when the user wants to control the creation of a shape. For example, you might specify that only two actors are allowed in the Use Case diagram.

Deletable: a deletable *Constraint* is similar to a creatable *Constraint,* except that it applies to deletion rather than creation.

Connectable: a connectable *Constraint* can be used to ensure that a connection connects a proper source shape to a proper target. For example, an actor can be connected to a use case – but not the other way around.

Attachable: an attachable *Constraint* can be used to make a shape adhesive to another, even when one of them moves (e.g. a *Property Mapper* attached to a *Shape* in the VSD).

Resizable: a resizable *Constraint* can be used to control the size of a shape (e.g. making the size of the shape unchangeable).

Containable: a containable *Constraint* can be used to turn a shape into a container that contains other shapes. When a container moves, all the contained shapes move with it.

Collectable: a collectable *Constraint* can be used to turn a shape into a collector that collects other shapes (e.g. *Properties* inside a *Shape* in the VSD).

Collapsible: a collapsible *Constraint* can be used to make a set of shapes collapse into a single icon or expand into multiple icons.

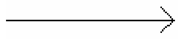
Zoomable: a zoomable *Constraint* can be used to allow a shape to be represented differently according to the zooming levels (e.g. showing a larger version of a shape if the zooming level is higher).

Participation Link

A *Participation Link* can be used to connect a *Shape/Connection* (participant) to a *Constraint* with or without a (participation) role, which indicates the *Shape/Connection* obeys the *Constraint.*

A *Participation Link* is represented as a solid line with a line arrowhead (see the right-hand figure below). The property of *Participation Link* is as follows:

- Role



For a *Participation Link*, the property, "Role", denotes the participation role, which differentiates the participant from others connected to the *Constraint.*

4.6 Artefact Elements

In VSL, *Artefact Elements* are a mechanism for a modeller to provide additional information for the readers. However, the basic structures of the view specification model, as determined by the *Structural Elements*, *Mapping Elements*, and *Constraint Elements*, are not changed with the addition of *Artefact Elements* in the VSD (as you can see by comparing Figure 4-4 to Figure 4-5); *Artefact Elements* simply provide secondary notation for modelling.

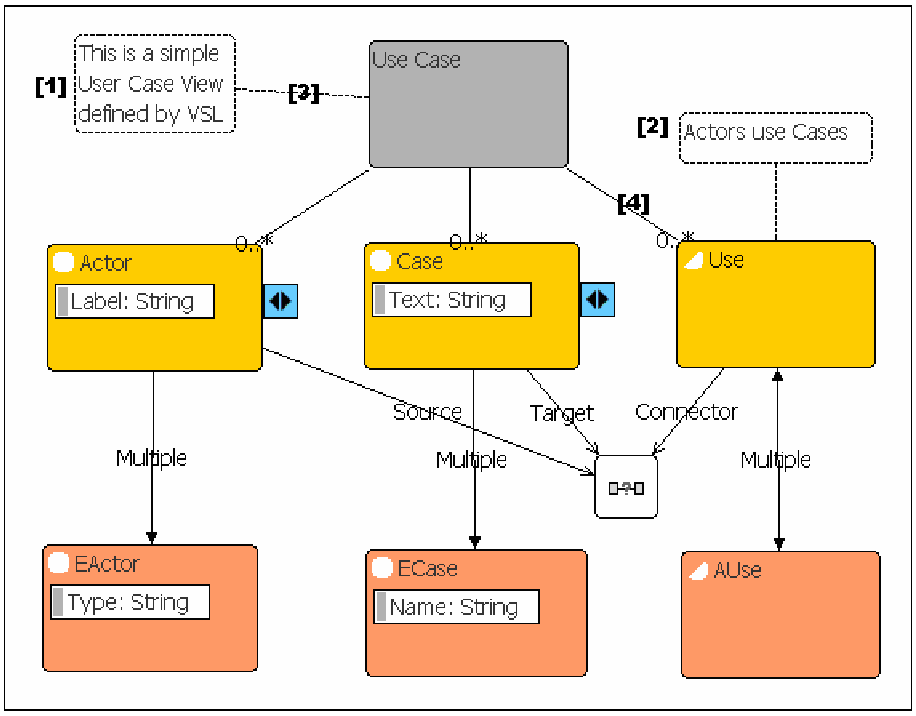


Figure 4-5 Artefact specification of the Use Case view type

Figure 4-5 shows an artefact specification of the Use Case view type on top of the constraint specification presented in Figure 4-4. The *Annotation* (1) is connected to The *View* (Use Case) through the *Artefact Association* (3). Similarly, the *Annotation* (2) is connected to The *Connection* (Use) through the *Artefact Association* (4).

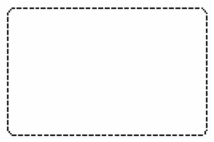
The detailed specifications of the two *Artefact Elements* are as follows:

Annotation

An *Annotation* can be used to add textual information in a VSD; it can stand on its own as a general comment; it can be connected to others such as a *Shape*, which indicates that the *Annotation* applies to that *Shape*.

An *Annotation* is represented by a dotted-line rectangle with a label containing text information in the centre (see the right-hand figure below). The properties of *Annotation* are as follows:

- Key
- Content

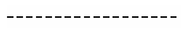


For an Annotation, the first property, “Key”, denotes the Annotation identification, which is usually generated by the system automatically. The last property, “Content”, contains the actual textural information of this Annotation.

Artefact Association

An *Artefact Association* can be used to connect an *Annotation* to others such as an *Entity*, which indicates that the *Annotation* refers to that *Entity.*

An *Artefact Association* is represented as a dashed line (see the right-hand figure below).



4.7 Summary

This chapter has presented VSL, mainly its notation. We first introduced some general concepts of the language. We then described all the notational elements in depth, such as *Structural Elements*, *Mapping Elements*, *Constraint Elements*, and *Artefact Elements*. In addition, we illustrated how to specify a simple Use Case view type using VSL in a series of steps. In the next chapter, we will illustrate the use of VSL with several complex examples.

CHAPTER 5

VSL EXAMPLES

5.1 Introduction

In this chapter, we present a number of examples of view specifications expressed using VSL. We start by introducing a basic example to clearly describe and explain the use of the fundamental elements of VSL. We then demonstrate how to use an *Input* and related elements, how to use a mixture of *Mapping Flows*, and how to use several *Constraints* for complex view specification scenarios. On the whole, the chapter is intended mainly to illustrate the use of VSL.

5.2 A Basic Example

Figure 5-1 shows an example View Specification Diagram (VSD), which defines a view type (called Example View). This view type itself is represented by the *View* (1), which is associated to two *Shapes* and one *Connection* using three *General Associations*. The *Shape* (Shape1) is connected to the *Entity* (Entity1) via a *Generation Flow*; the *Shape* (Shape2) is connected to the *Entity* (Entity1) via a *Search Flow*; and the *Connection* (Connection1) is connected to the *Association* (Association1) via a *Generation Flow*. Also shown are view-to-model property mappings. For example, the view *Property* (Property1), contained in Shape1, is mapped to the model *Property* (Property3) through two *Mapping Associations* inserted with a *Property Mapper*.

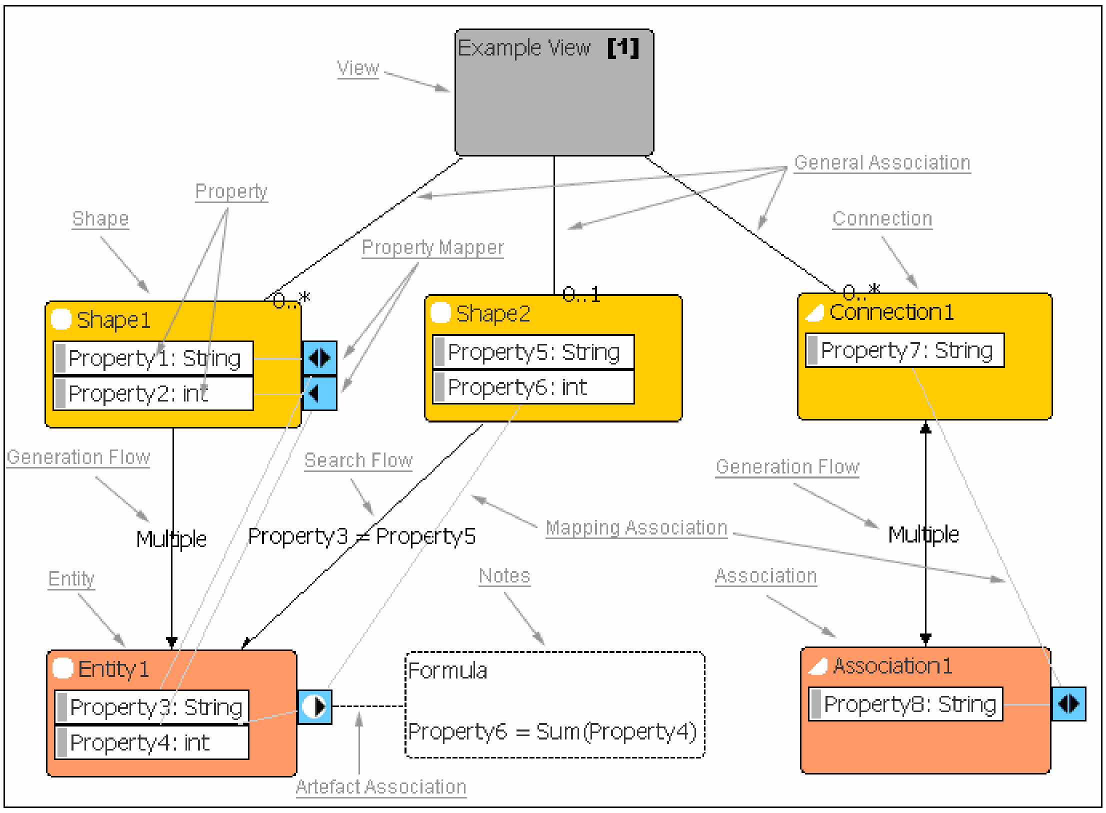


Figure 5-1 An example View Specification Diagram

In the following, we divide the above example into three parts, shown in Figure 5-2, Figure 5-3 and Figure 5-4, respectively. For each part, we describe the elements involved and illustrate the process in detail.

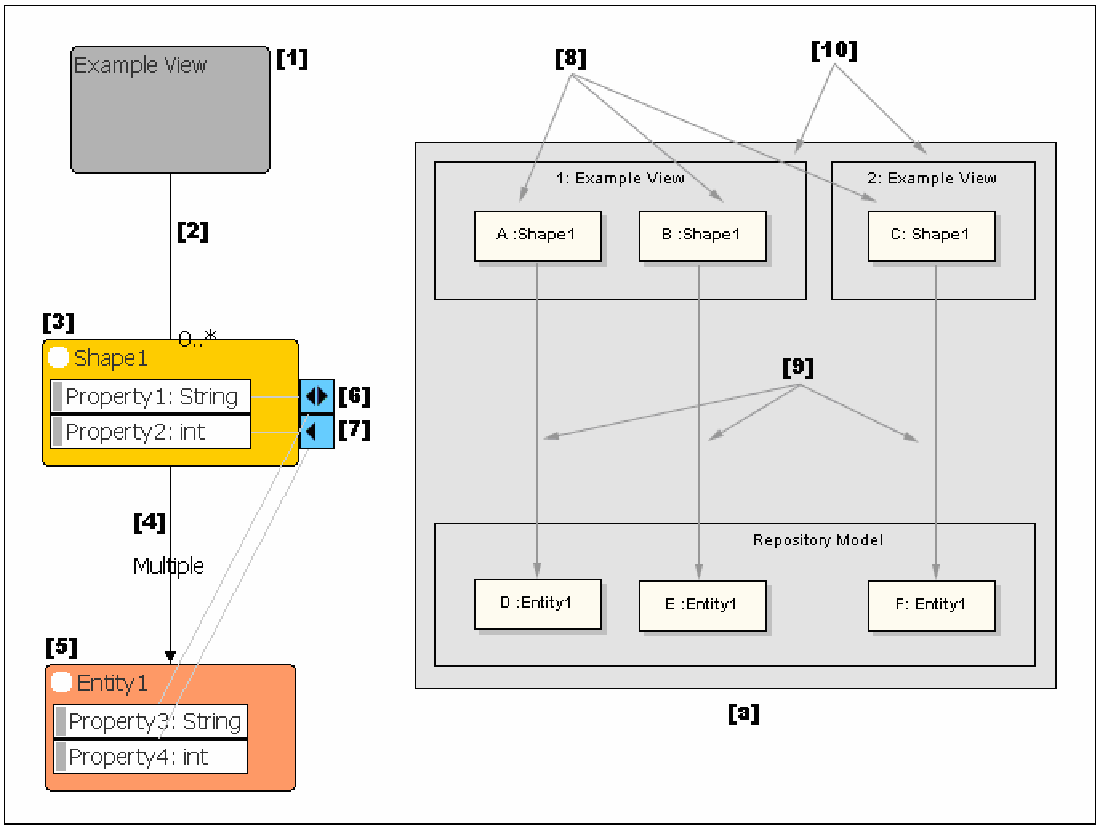


Figure 5-2 Illustration of the part one of the basic example

Figure 5-2 illustrates the first part. The *View* (1) represents the "Example View" view type being modelled; similarly, the *Shape* (3) represents the "Shape1" shape type; and the *Entity* (5) represents the "Entity1" entity type. The *General Association* (2), assigned with a target multiplicity of "0..*", indicates that an Example View diagram (e.g. 10) can contain null to unlimited instances (e.g. 8) of Shape1. The unidirectional *Generation Flow* (4), assigned with a type of "Multiple", indicates that whenever an instance of Shape1 is created in the diagram, a related instance of Entity1 is generated in the common repository (e.g. 9). In addition, the view *Property* (Property1) is mapped to the model *Property* (Property3) using the bidirectional *Property Mapper* (6), meaning that changes made to Property1 (the Property1 of a Shape1 instance) apply to Property3 (the Property3 of a corresponding Entity1 instance), and vice versa. Whereas the view *Property* (Property2) is mapped to the model *Property* (Property4) using the unidirectional model-to-view *Property Mapper* (7), meaning that changes made to Property4 apply to Property2, and not the other way around.

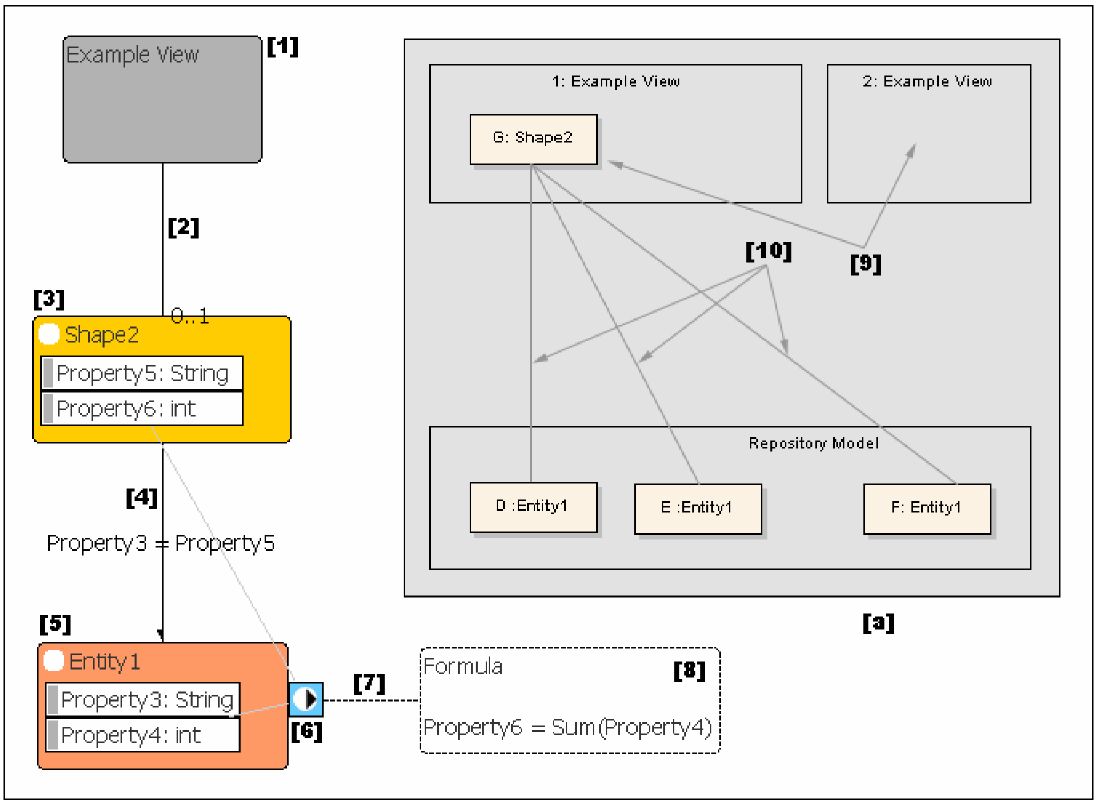


Figure 5-3 Illustration of the part two of the basic example

Figure 5-3 illustrates the second part. The *View* (1) represents the "Example View" view type being modelled; similarly, the *Shape* (3) represents the "Shape2" shape type; and the *Entity* (5) represents the "Entity1" entity type. The *General Association* (2), assigned with a target multiplicity of "0..1", indicates that an Example View diagram can contain null to single instance (e.g. 9) of Shape2. The *Search Flow* (4), assigned with a search criteria of "Property3 = Property5", indicates that an instance of Shape2 is dynamically associated to specific instances of Entity1 based on its Property5 value (e.g. 10). In addition, the view *Property* (Property6) is mapped to the model *Property* (Property4) using the unidirectional model-to-view *Property Mapper* (6) assigned with a formula of "Property6 = Sum(Property4)", meaning that changes made to Property4 apply to Property6 and the Property6 value of a Shape2 instance equals the sum of the Property4 values of related Entity1 instances according to (4). Also shown is the *Annotation* (8), which is associated to the *Property Mapper* (6) via the *Artefact Association* (7).

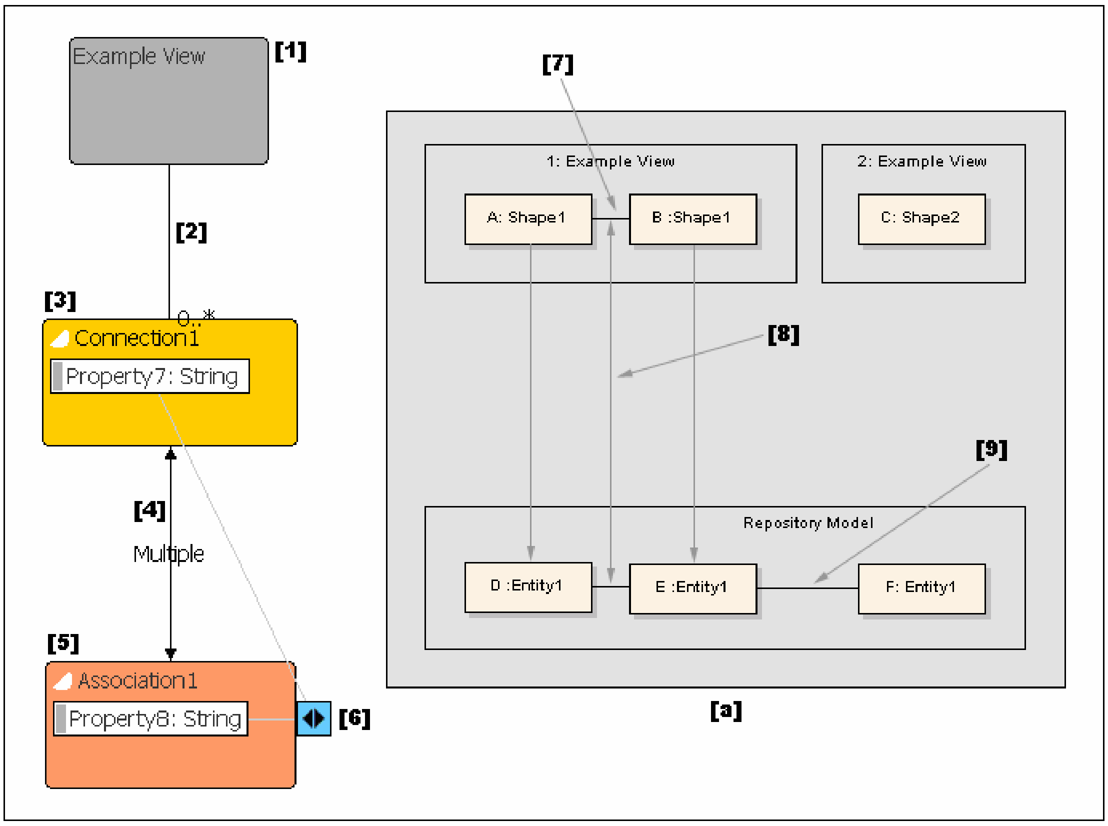


Figure 5-4 Illustration of the part three of the basic example

Figure 5-4 illustrates the last part. The *View* (1) represents the "Example View" view type being modelled; similarly, the *Connection* (3) represents the "Connection1" connection type; and the *Association* (5) represents the "Association1" association type. The *General Association* (2), assigned with a target multiplicity of "0..*", indicates that an Example View diagram can contain null to unlimited instances (e.g. 7) of Connection1. The bidirectional *Generation Flow* (4), assigned with a type of "Multiple", indicates that whenever an instance of Connection1 is created in the diagram, a related instance of Association1 is generated in the common repository, and vice versa if it is applicable (e.g. applicable: 8, inapplicable: 9). In addition, the view *Property* (Property7) is mapped to the model *Property* (Property8) using the bidirectional *Property Mapper* (6), meaning that changes made to Property7 apply to Property8, and vice versa.

5.3 Using an Input

Figure 5-5 (a) shows a simple UML Class diagram, with two Class elements connected. The Class (1), which represents a student class, is connected to the Class (2), which represents a School class, through an Association (connection) assigned with a name of "Study at". These two classes are instantiated as Object elements in an Object diagram, as shown in Figure 5-5 (b). There are one instance of School, the Object (3), and two instances of Student, the Object (4) and Object (5). These Objects are also connected (as shown) via two Associations.

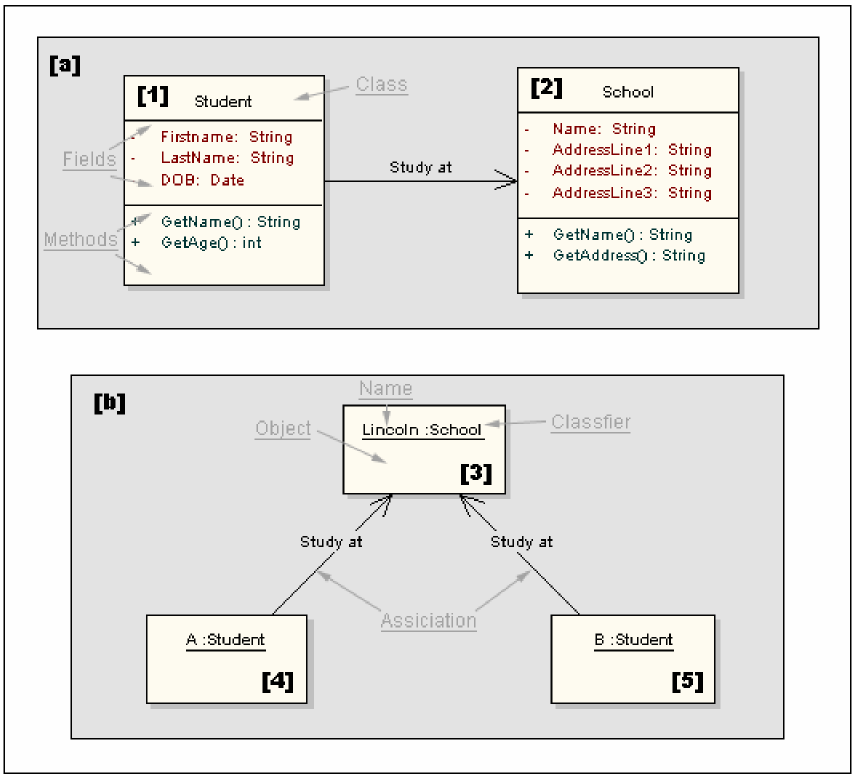


Figure 5-5 (a) A UML Class diagram and (b) a related Object diagram

Although an object is an instance of a class, an Object element is actually different to a Class element. An Object is typically associated to a Class using an Association assigned with a name of "Instance of", and it is generally created with a specific Class known in advance (so that a proper Association can be generated automatically). This is one of the possible situations in which Inputs can be used. Figure 5-6 shows an example of using an *Input*.

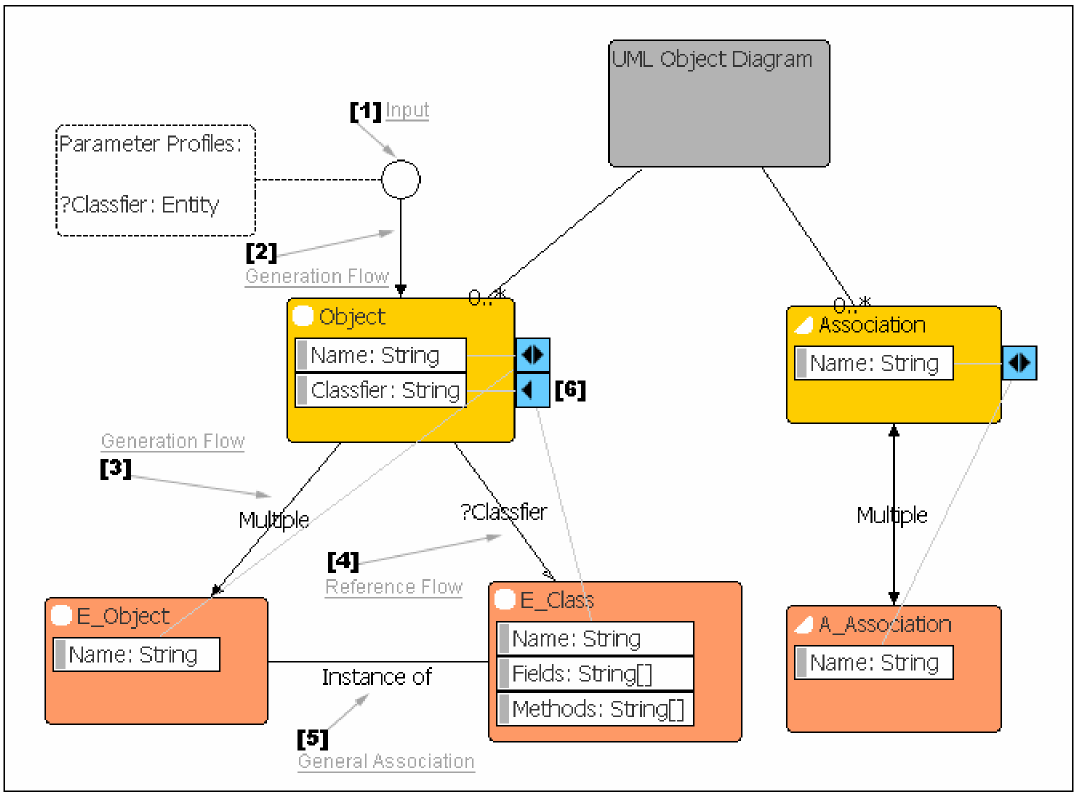


Figure 5-6 Example of using an Input

In Figure 5-6, the *Input* (1), assigned with a parameter profile of "?Classifier: Entity", is connected to the *Shape* (Object) via the unidirectional *Generation Flow* (2), meaning that an external value, defined by the parameter profile, is passed into the creation of an Object instance. The unidirectional *Generation Flow* (3), assigned with a type of "Multiple", indicates that whenever an instance of Object is created in the diagram, a related instance of E_Object is generated in the common repository. The *Reference Flow* (4), assigned with an input parameter of "?Classifier", indicates that the external value (passed in) is treated as a specific instance of E_Class. The *General Association* (5), assigned with a name of "Instance of", indicates that an E_Object instance created is automatically connected to the E_Class instance (passed in) via a new "Instance of" Association. The unidirectional model-to-view *Property Mapper* (6) indicates that changes made to the property Name of an E_Class instance apply to the property Classifier of related Object instances according to (4).

5.4 Using a mixture of Mapping Flows

Figure 5-7 shows a Small Business diagram. An Employee (which represents a person who works for a company) can be connected to a Department (which represents a division of a company) via a "Work at" connection; and a Department can be connected to a Company (which represents a business organization) via a "Belong to" connection. After these elements are connected, some of their properties are changed automatically. The property Department of an Employee presents the name of the Department this Employee works at (e.g. 4); the property NoOfPeople of a Department is the total number of the Employees this Department has (e.g. 3); the property NoOfPeople of a Company is the sum of the NoOfPeople values of the Departments this Company contains (e.g. 2). In particular, there are two identical Companies, each of which takes their own "Belong to" connections, as shown in the diagram (1).

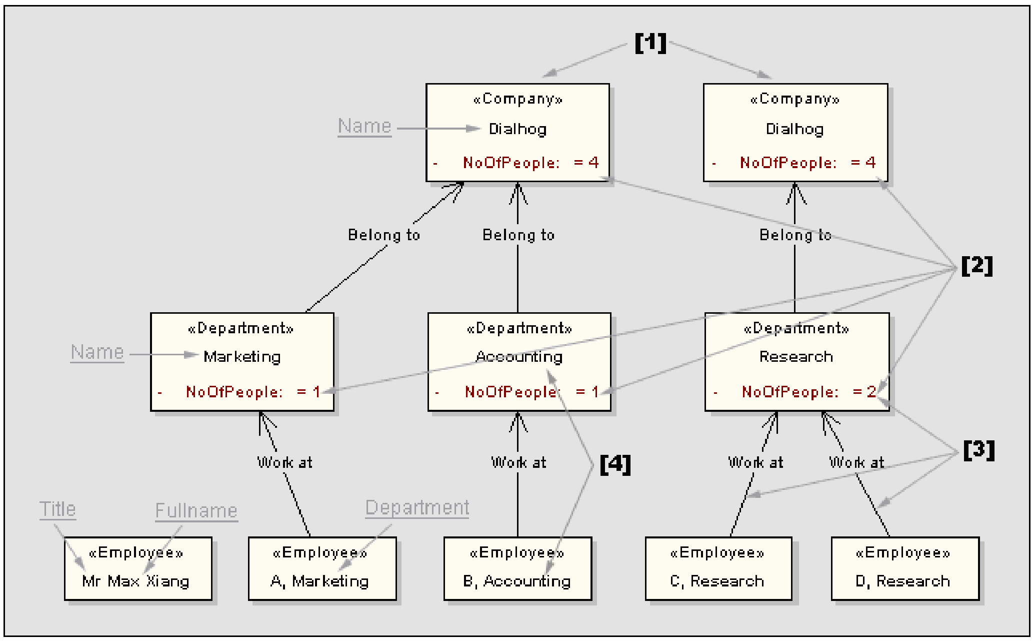


Figure 5-7 A Small Business diagram

In the following, we specify the above view type in three phases, shown in Figure 5-8, Figure 5-9 and Figure 5-10, respectively. For each phase, we describe and explain the important elements, especially *Mapping Flows*, in detail.

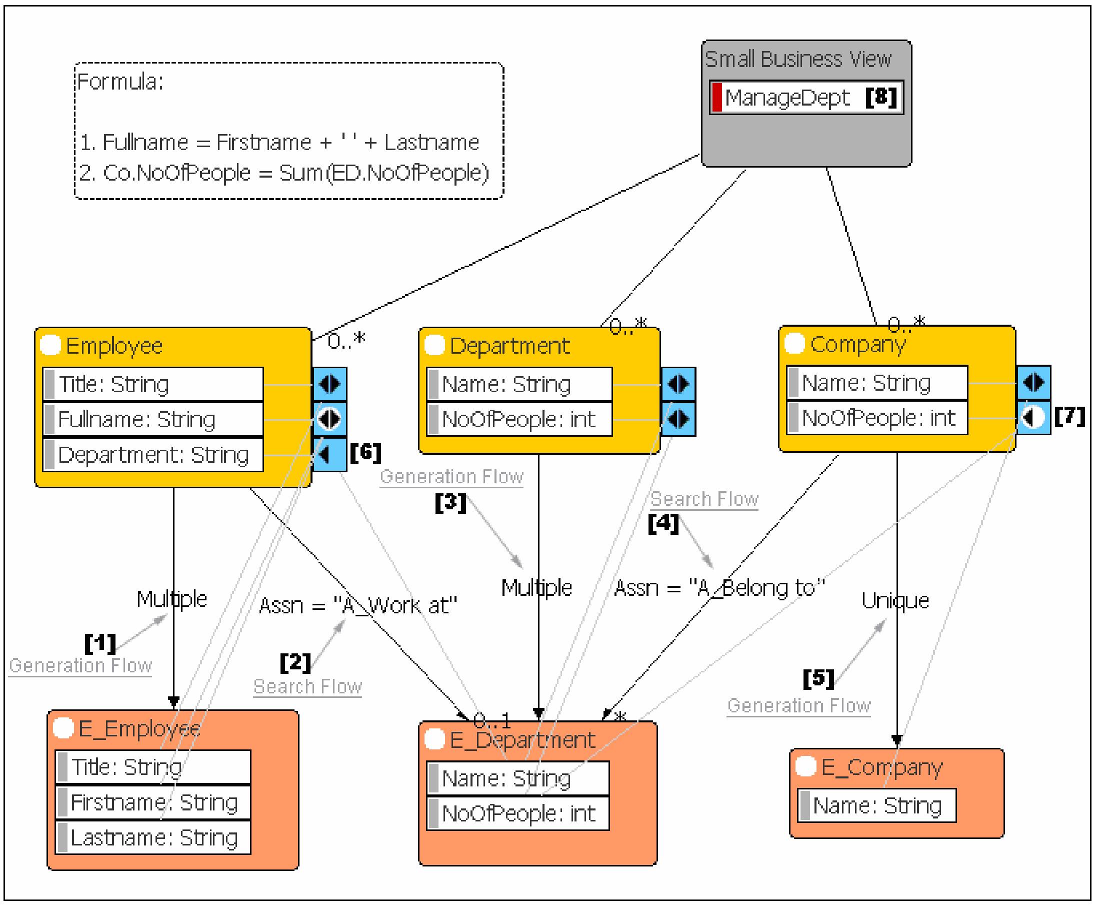


Figure 5-8 Phase one of the Small Business View specification

Figure 5-8 shows the first phase. There are three *Generation Flows*. The unidirectional *Generation Flow* (1), assigned with a type of “Multiple”, indicates that whenever an Employee instance is created in the diagram, a related E_Employee instance is generated in the common repository. The *Generation Flow* (3) is used in a similar way to (1). The unidirectional *Generation Flow* (5), assigned with a type of “Unique”, indicates that no matter how many instances of Company are created in the diagram (or related diagrams), there is only one instance of E_Company in the common repository, and this instance corresponds to all the instances of Company (e.g. Figure 5-7[1]).

Also shown in Figure 5-8, there are two *Search Flows*. The *Search Flow* (2), assigned with a search criteria of “Assn = A_Work at” and a number-of-instance of “0..1”, indicates that an instance of Employee is associated to zero or one specific instance of E_Department based on the search criteria. The *Search Flow* (4), assigned with a search criteria of “Assn = A_Belong to” and a number-of-instance of “*”, indicates that a Company instance is associated to an arbitrary number of E_Department instances based on the search criteria.

In addition, there are two *Property Mappers* related to (2) and (4). The unidirectional model-to-view *Property Mapper* (6) indicates that changes made to the property Name of an E_Department instance apply to the property Department of related Employee instances according to (2) (e.g. Figure 5-7^4). The unidirectional model-to-view *Property Mapper* (7), assigned with a formula of “Compnay.NoOfPeople = Sum(E_Department.NoOfPeople)”, indicates that changes made to the property NoOfPeople of an E_Department instance apply to the property NoOfPeople of related Company instances, and the NoOfPeople value of an Company instance equals the sum of the NoOfPeople values of all related E_Department instances according to (4) (e.g. Figure 5-7^2). (The Visual Handler, 8, is used to increase the NOOfPeople value of Department instances automatically.)

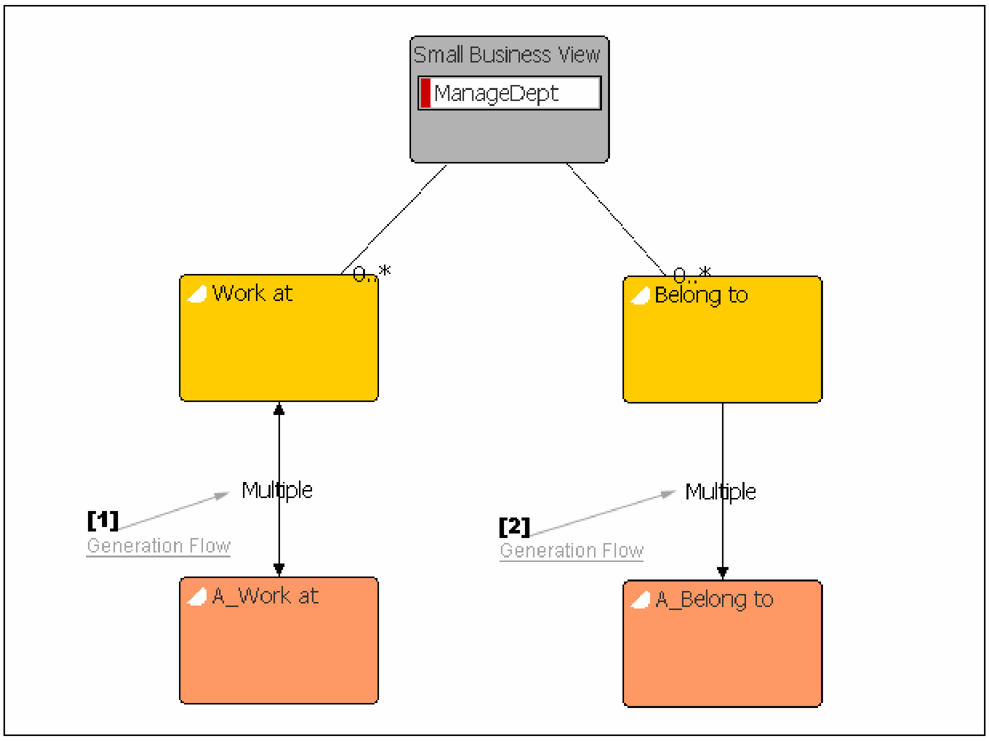


Figure 5-9 Phase two of the Small Business View specification

Figure 5-9 shows the second phase. There are two *Generation Flows*. The bidirectional *Generation Flow* (1), assigned with a type of "Multiple", indicates that whenever an instance of "Work at" is created in the diagram, a related instance of "A_Work at" is generated in the common repository, and vice versa if it is applicable (e.g. Figure 5-4[8, 9]). The unidirectional *Generation Flow* (2), assigned with a type of "Multiple", indicates that whenever an instance of "Belong to" is created in the diagram, a related instance of "A_Belong to" is generated in the common repository, and not the other way around (this is necessary because multiple identical Company instances, sharing the same E_Company instance, can appear in the same diagram e.g. Figure 5-7[1]).

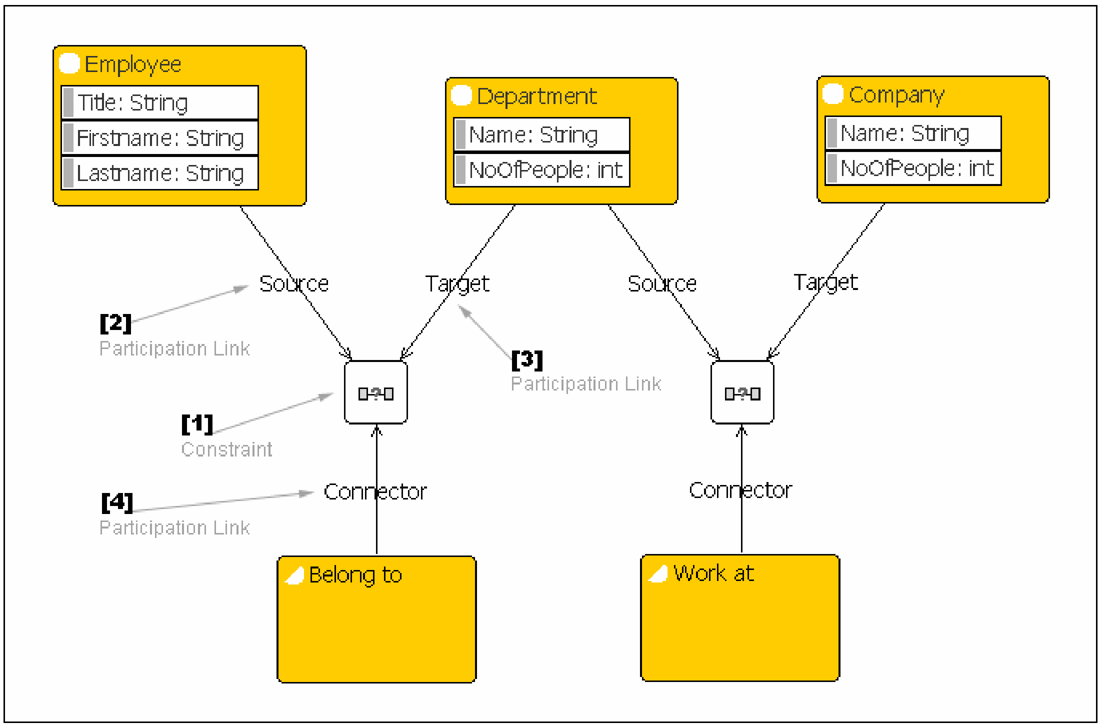


Figure 5-10 Phase three of the Small Business View specification

Figure 5-10 shows the last phase. The connectable *Constraint* (1) is joined by the *Shape* (Employee) as a Source via the *Participation Link* (2), the *Shape* (Department) as a Target via the *Participation Link* (3), and by the *Connection* (Belong to) as a Connector via the *Participation Link* (4). This indicates that an Employee instance can connect to a Department instance via a “Belong to” instance, and not the other way around. (The connectable *Constraint*, right, is used in a similar way.)

5.5 Using several Constraints together

Figure 5-11 shows a simple UML Activity diagram, with an Activity element containing several other elements. The Activity (1) represents a “Log on” activity. It begins with the Action (Get Username), which outputs a valid Username object. The next Action is “Get Password”, which outputs a valid Password object. The Activity (Authenticate User) executes when it receives a valid Username and a valid Password via its input Pins (2). The user is authenticated, and the activity finishes.

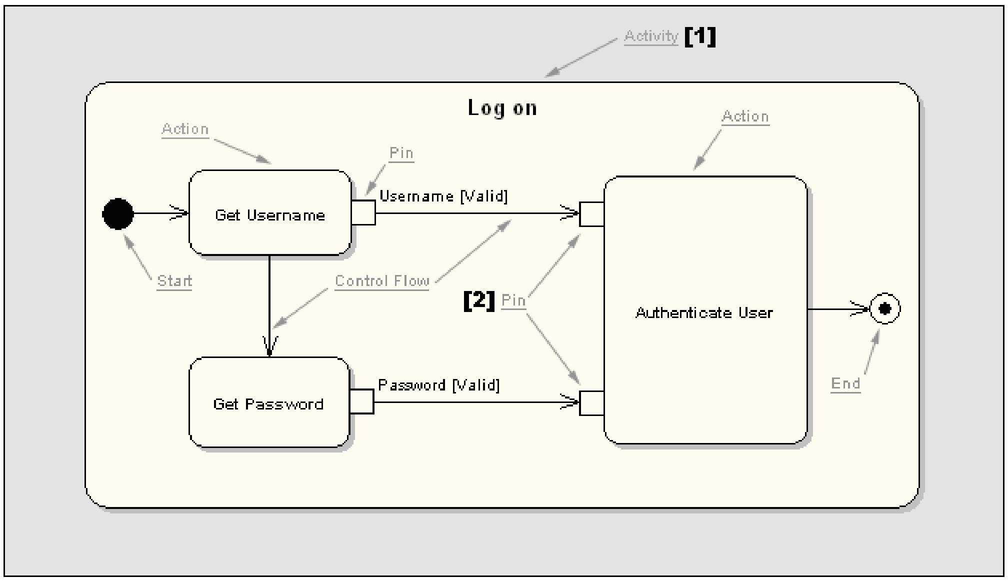


Figure 5-11 A UML Activity diagram

As you can see, an Activity can contain other elements, such as Starts, Ends and Actions (e.g. Figure 5-11^1); an Action can be attached with multiple Pins (e.g. Figure 5-11^2), which are fixed in size. These can be easily achieved by using specific *Constraints*. Figure 5-12 shows an example of using such *Constraints*.

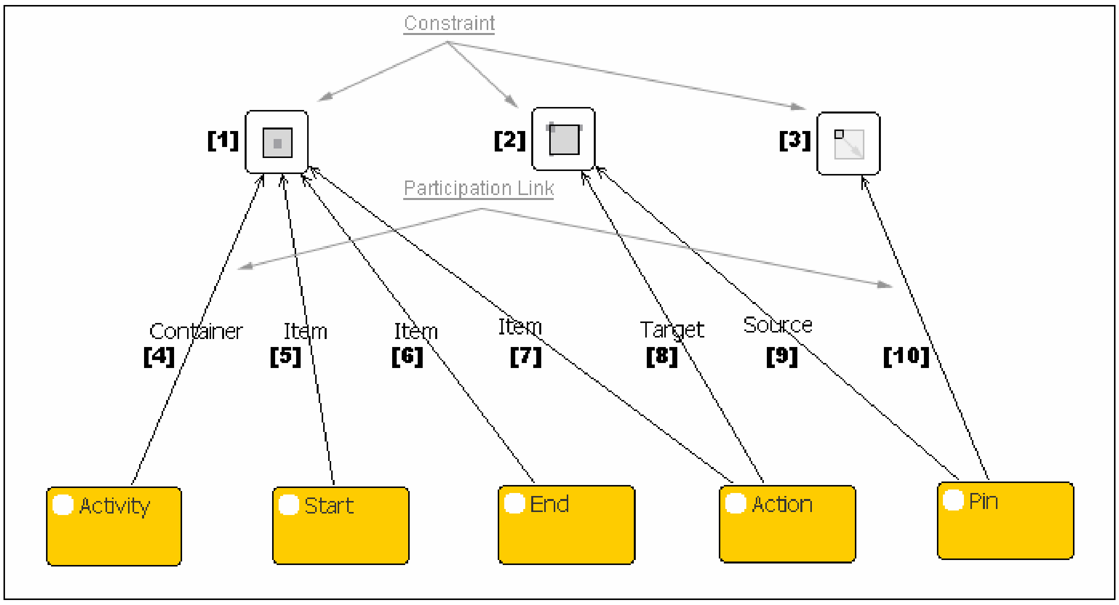


Figure 5-12 Example of using several Constraints

In Figure 5-12, there are three *Constraints*. The containable *Constraint* (1) is joined by the *Shape* (Activity) as a Container via the *Participation Link* (4), the *Shape* (Start) as an Item via the *Participation Link* (5), the *Shape* (End) as an Item via the *Participation Link* (6), and by the *Shape* (Action) as an Item via the *Participation Link* (7). This indicates that an Activity instance can (visually) contain instances of Start, End, and Action (e.g. Figure 5-11[1]). The attachable *Constraint* (2) is similarly joined by the *Shape* (Action) as a Target via the *Participation Link* (8) and by the *Shape* (Pin) as a Source via the *Participation Link* (9). This indicates that an Action instance can be attached with Pin instances (e.g. Figure 5-11[2]). In addition, the resizable *Constraint* (3) is joined by the *Shape* (Pin) alone (without an explicit participation role). This indicates that the size of Pins is constrained, e.g. unchangeable.

5.6 Summary

In this chapter, we illustrated the use of VSL by showing several View Specification Diagrams (VSDs). We in turn demonstrated a basic view specification example, a simple UML object diagram, a Small Business diagram, and a simple UML activity diagram.

CHAPTER 6

VSL TOOL IMPLEMENTATION

6.1 Introduction

In this chapter, we present the design and implementation of our proof-of-concept tool. We first introduce the implementation background, to provide a general description about the Eclipse Platform, Eclipse Modelling Framework (EMF), and Graphical Editor Framework (GEF). We then describe our implementation approach and highlight the two core modules of the tool, the Constraint module and the Back End Generation module. Finally, we go on to discuss each of these modules, respectively.

6.2 The Eclipse Platform and Related Techniques

Eclipse is a Software Development Kit (SDK), which can be used as both a Java™ integrated development environment (IDE) and a tool for building products based on the Eclipse Platform [Eclipse, 2006]. The Eclipse SDK consists of several core Eclipse projects: Platform, Java Development Tools (JDT), and the Plug-in Development Environment (PDE).

The main features of the Eclipse Platform are the following:

- The Eclipse Platform provides a managed windowing system: user interface components (e.g. entry fields and push buttons), docking views and editors, drag and drop, and the ability to contribute menu items and tool bars.
- It can be used as an integration point, so that building a tool on top of the Eclipse Platform enables the tool to integrate with other tools also written using the Eclipse Platform.
- It is built on a mechanism for discovering, integrating, and running modules called plug-ins (in fact the Platform itself is implemented as several core plug-ins). Additional tools can be implemented as Eclipse plug-ins to extend the functionality of the Eclipse Platform, such as working with new content types and doing new things with existing content types.

Many frameworks have been created based on the Eclipse Platform. There are two modelling frameworks that are closely related to our implementation work. They are:

- EMF (Eclipse Modelling Framework) – a Java framework and code generation facility for building tools and other applications based on a structured model [EMF, 2004].
- GEF (Graphical Editor Framework) – a foundation for building rich, interactive user interfaces that are not easily built using native widgets found in the base Eclipse Platform [GEF, 2004].

An EMF model can be defined as an application model in UML using a combination of UML diagrams (e.g. Class Diagram, Collaboration Diagram, State Diagram, and etc). Once a model is specified, the EMF generator can create a corresponding set of Java implementation classes shipped with sophisticated model change notification, persistence support, model validation, and a very efficient reflective API for manipulating EMF objects generically.

The GEF component is further separated into two Eclipse plug-ins: Draw2d and GEF. The Draw2d plug-in is a lightweight toolkit containing graphical components called figures, which are used to construct any types of diagrams, documents, or drawings. The GEF plug-in adds editing capability on top of Draw2d, to facilitate the display of any model graphically and support interactions from mouse, keyboard, or the Eclipse workbench.

6.3 Overview of Implementation Approach

Figure 6-1 shows the main components of our VSL tool. This tool is expressed as a meta-tool specification along with several event handlers. The meta-tool specification can be loaded into Marama [Grundy et al., 2006], a set of Eclipse plug-ins for realizing Pounamu-based visual tools. While the tool is running, Marama is responsible for automatically instantiating and deposing related event handler instances as needed. The primary goal of this tool effort was to:

- Provide a multi-view visual environment for specifying various view (diagram) types using the VSL notation.
- Generate the Pounamu view specification artefacts from a view type model as defined.

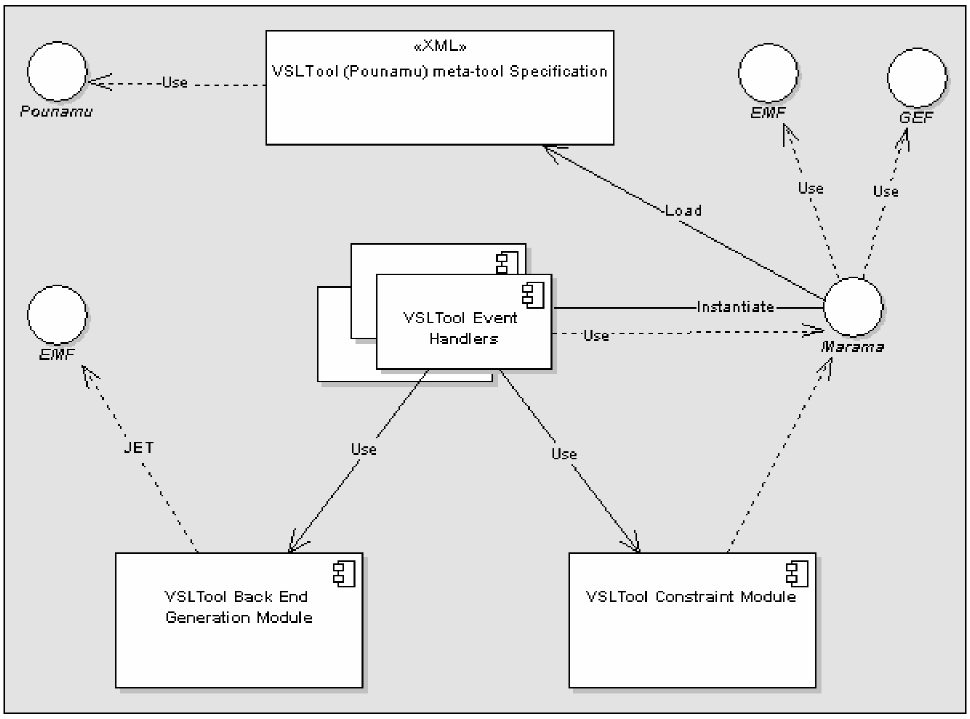

«XML»
VSLTool (Pounamu) meta-tool Specification
Pounamu
Use
EMF
GEF
Use
Use
Load
VSLTool Event Handlers
Instantiate
Use
Marama
EMF
JET
Use
Use
VSLTool Back End Generation Module
VSLTool Constraint Module

Figure 6-1 Implementation of the VSL Tool

Although the (Marama) event handlers required in the tool are very complex, the most of them are relatively easy to implement. The reason is because the use of two important fundamental modules (as shown in Figure 6-1, bottom):

- The Constraint module includes a set of built-in constraint handlers, which are used to support the VSL constraint mechanism.
- The Back End Generation module provides easy-to-use facilities for template-based file generation in relation to view specification models.

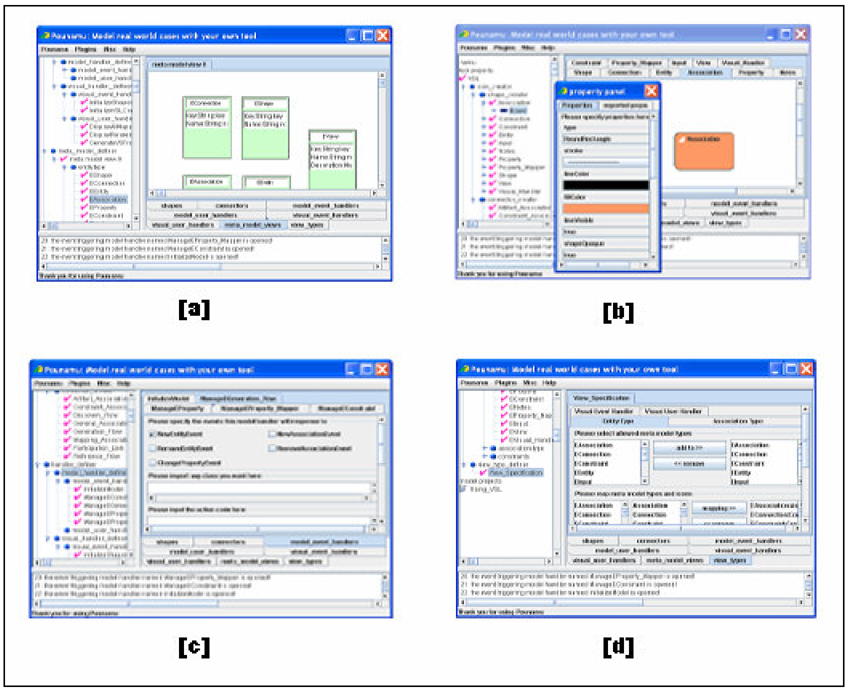


Figure 6-2 The VSL Tool specifications in Pounamu

The meta-tool specification is created in Pounamu using several meta-tool designers, as shown in Figure 6-2. The tool model (entities and associations) data is defined in the meta-model designer (a); the VSL notational elements (shapes and connections) are specified via the shape/connector designer (b); the event handlers (as required by the tool) are added into the specification using the event handler designer (c); and the VS (View Specification) view type is constructed through the view designer based on the notational elements, model data elements, and event handlers that are specified above (d).

Before going to see the details about the Constraint and Back End Generation modules (in the following sections), it helps to have a sense of what the VSL tool dose and what it looks like to the user.

6.3.1 The usage at a glance

When using the VSL tool, a user may perform four sequential steps: the first step is for the user to start Eclipse and then load the tool project specification; the second step is to create a view specification diagram; the third step would be to draw necessary (VSL) graphical elements in the diagram; and the last step is to generate the view specification artefacts (e.g. XML files and Java codes) from the view type model that is defined. Figure 6-3 shows a screen capture of the typical Eclipse main workbench window while the user is working with the VSL tool.

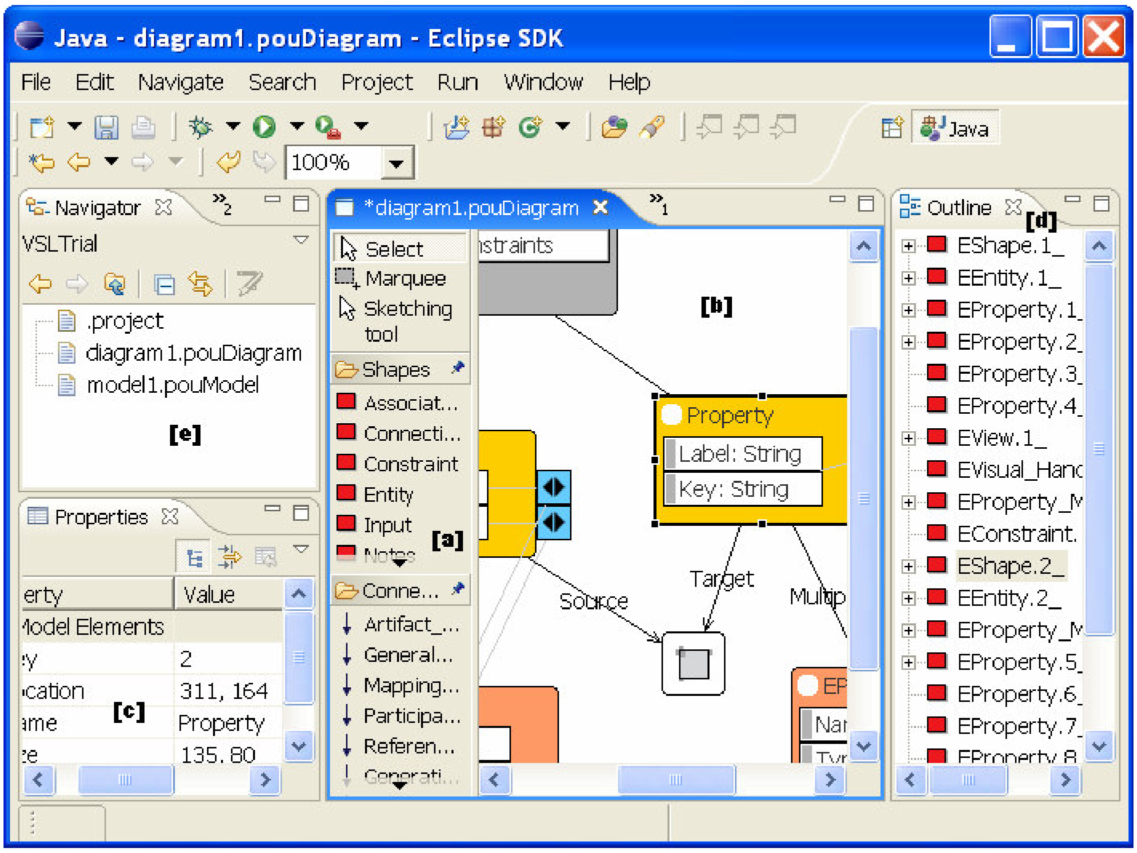


Figure 6-3 The VSL Tool being used in Marama

Graphical elements are accessed via a palette (a); shapes and connections can be directly manipulated in a canvas (b); properties of a selected shapes or connection can be edited using the standard Eclipse property sheet (c); a hierarchical (Marama) outline view of a current-active diagram is provided (d); and the navigator view shows the tool model projects and diagrams available.

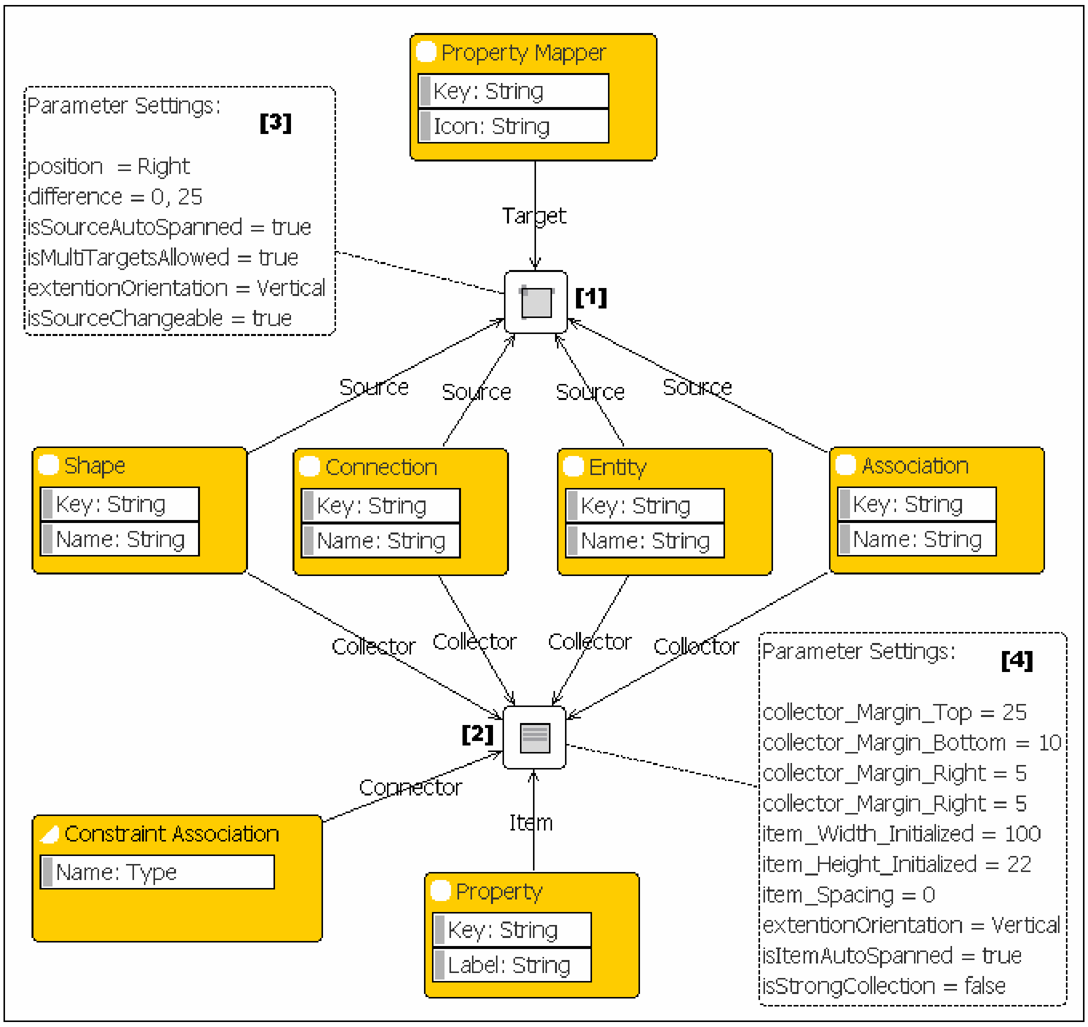
6.4 The Constraint Module

The Constraint module contains a set of constraint handlers, which is essentially built on top of the Marama event handler model. In this way, a constraint handler can be regarded and even used as a specialized Marama event handler; therefore, it can certainly achieve whatever a Marama event handler is incapable to achieve e.g. receiving and propagating change descriptions, and maintaining consistency between multiple views of the same common repository.

However, there are several advantages to use constraint handlers, in stead of Marama event handlers, in the VSL tool. These advantages are the following:

- Constraint handlers are usually ready-to-use components; therefore, much low-level (error-prone) programming work may be avoided.
- They have been specifically chosen, designed, and implemented, targeting to the specification problems (e.g. controlling the size of shapes or restricting the use of a connection between particular shapes) that are commonly encountered by many modellers.
- They are fairly easy to apply and highly reusable: each of them corresponds to a (VSL) *Constraint* of a specific type, such as “Attachable” and “Collectable”, and thus it can be automatically instantiated using the related information that is specified in the diagram.
- They are customizable; implementing user-defined and extending existing constraint handlers are supported. This allows the Constraint module to be extended further by developers.

Figure 6-4 shows part of the VSL tool view specification, which is used to describe the concepts and illustrate the processes that are presented in the following subsections.



Parameter Settings: **[3]**

position = Right
difference = 0, 25
isSourceAutoSpanned = true
isMultiTargetsAllowed = true
extentionOrientation = Vertical
isSourceChangeable = true

Parameter Settings: **[4]**

collector_Margin_Top = 25
collector_Margin_Bottom = 10
collector_Margin_Right = 5
collector_Margin_Right = 5
item_Width_Initialized = 100
item_Height_Initialized = 22
item_Spacing = 0
extentionOrientation = Vertical
isItemAutoSpanned = true
isStrongCollection = false

Figure 6-4 Part of the VSL Tool view specification

In Figure 6-4, there are two *Constraints* used in the diagram. The attachable *Constraint* (1) is joined by the *Shapes* (Shape, Connection, Entity, and Association as "Source" participants) and the *Shape* (Property Mapper as a "Target" participant). The collectable *Constraint* (2) is connected with the *Shapes* (Shape, Connection, Entity, and Association as "Collector" participants), the *Shape* (Property as an "Item" participant), and the *Connection* (Constraint Association as a "Connector" participant). Also shown are two *Annotations*. The top (3) visualizes the parameter settings defined in (1), whereas the bottom (4) represents the parameter settings specified in (2).

In the following subsections, we first introduce the common infrastructure that helps support the development of constraint handlers, and then briefly demo how to implement a custom constraint handler based on this infrastructure.

6.4.1 The Common Infrastructure

The common infrastructure is made up of three core Java classes, which form an underlying implementation structure for developing constraint handlers. The main aim of this infrastructure is to provide a consistent implementation style throughout all the constraint handlers, and to provide a mechanism for instantiating constraint handlers automatically (as specified in the diagram). Figure 6-5 shows a UML Class diagram containing these three classes.

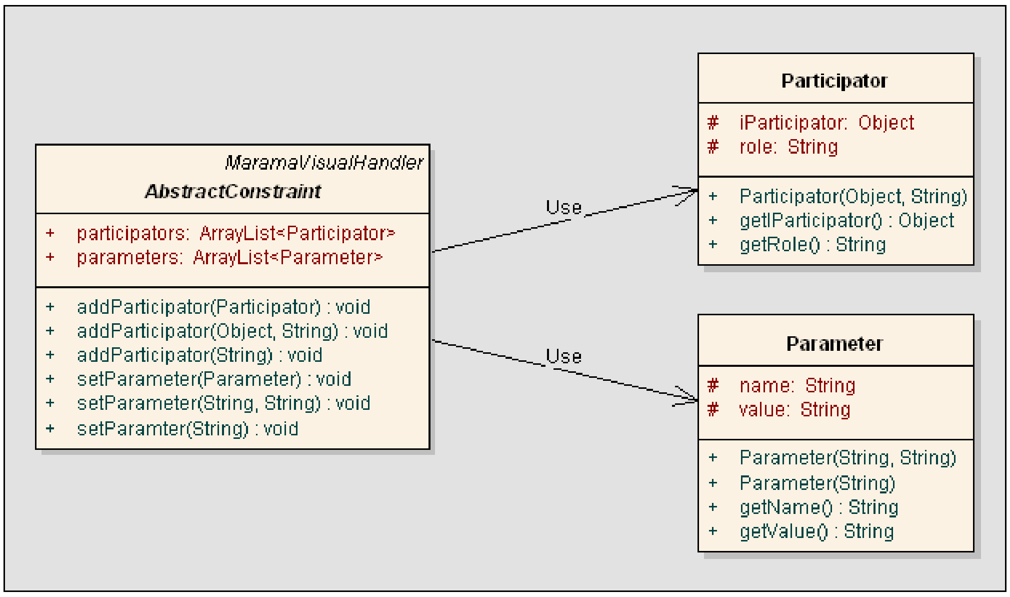
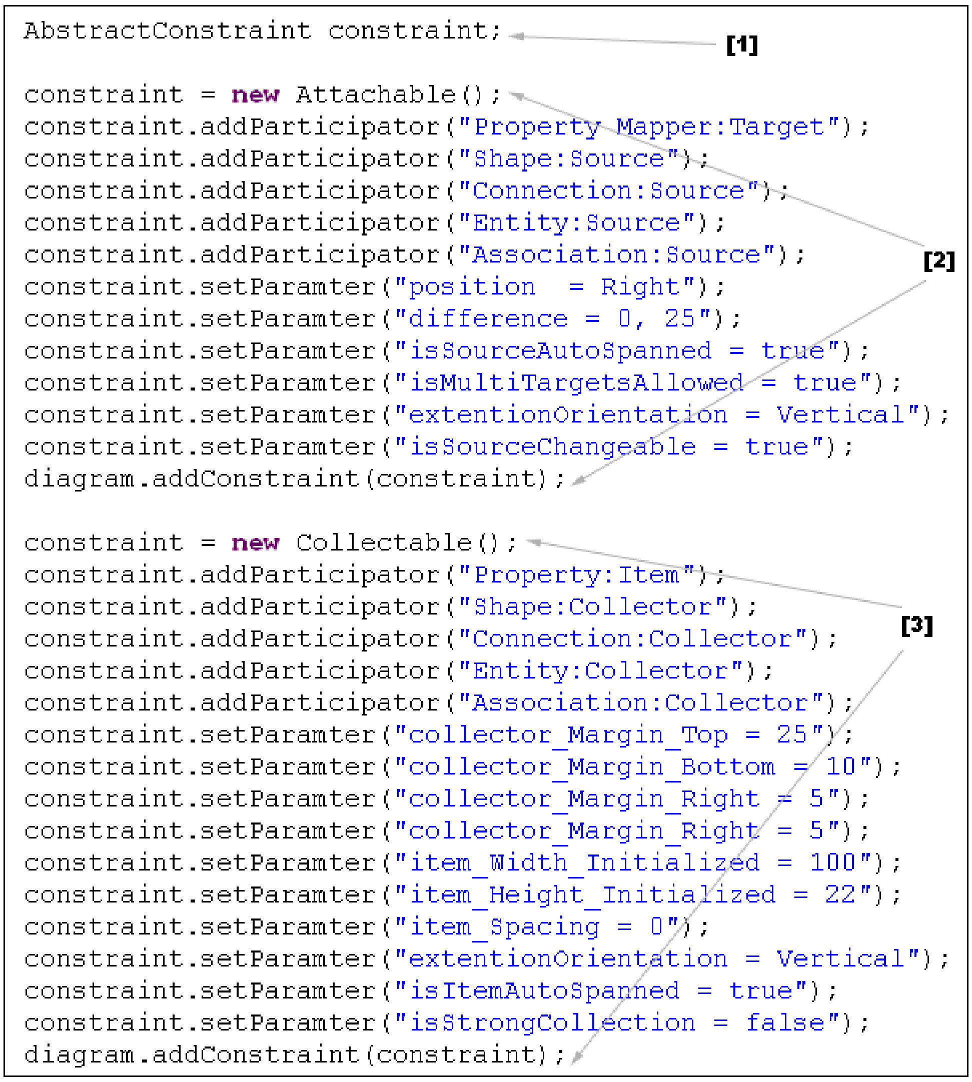

MaramaVisualHandler
AbstractConstraint
+ participators: ArrayList<Participator>
+ parameters: ArrayList<Parameter>
+ addParticipator(Participator) : void
+ addParticipator(Object, String) : void
+ addParticipator(String) : void
+ setParameter(Parameter) : void
+ setParameter(String, String) : void
+ setParamter(String) : void
Use
Use
Participator
# iParticipator: Object
# role: String
+ Participator(Object, String)
+ getIParticipator() : Object
+ getRole() : String
Parameter
# name: String
# value: String
+ Parameter(String, String)
+ Parameter(String)
+ getName() : String
+ getValue() : String

Figure 6-5 Class diagram of the common infrastructure

The AbstractConstraint class (Figure 6-5, left) is an abstract base class, which much be inherited by all the constraint handlers. It itself inherits the MaramaVisualHandler class so that any constraint handler instance can be used as a MaramaVisualHandler instance.

In addition, it uses the other two classes, Participator and Parameter. The Participator class is meant to encapsulate a participant object, usually a shape/connection type with a participation role, whereas the Parameter class is used to encapsulate a parameter object, typically a parameter name with a desired value.

Also shown are the attributes and methods contained in these classes. In the following, we briefly describe them by using part of the Java file that is generated by the VSL tool from the view specification illustrated in Figure 6-4.



```
AbstractConstraint constraint;

constraint = new Attachable();
constraint.addParticipator("Property Mapper:Target");
constraint.addParticipator("Shape:Source");
constraint.addParticipator("Connection:Source");
constraint.addParticipator("Entity:Source");
constraint.addParticipator("Association:Source");
constraint.setParamter("position  = Right");
constraint.setParamter("difference = 0, 25");
constraint.setParamter("isSourceAutoSpanned = true");
constraint.setParamter("isMultiTargetsAllowed = true");
constraint.setParamter("extentionOrientation = Vertical");
constraint.setParamter("isSourceChangeable = true");
diagram.addConstraint(constraint);

constraint = new Collectable();
constraint.addParticipator("Property:Item");
constraint.addParticipator("Shape:Collector");
constraint.addParticipator("Connection:Collector");
constraint.addParticipator("Entity:Collector");
constraint.addParticipator("Association:Collector");
constraint.setParamter("collector_Margin_Top = 25");
constraint.setParamter("collector_Margin_Bottom = 10");
constraint.setParamter("collector_Margin_Right = 5");
constraint.setParamter("collector_Margin_Right = 5");
constraint.setParamter("item_Width_Initialized = 100");
constraint.setParamter("item_Height_Initialized = 22");
constraint.setParamter("item_Spacing = 0");
constraint.setParamter("extentionOrientation = Vertical");
constraint.setParamter("isItemAutoSpanned = true");
constraint.setParamter("isStrongCollection = false");
diagram.addConstraint(constraint);
```

Figure 6-6 Part of the instantiation code generated

The above exposed code (Figure 6-6) is divided into three parts: (1), (2), and (3). In the top part (1), a variable called "constraint" is declared as the AbstractConstriant Type. Then, it is assigned with a new instance of Attachable (one of the built-in constraint handlers) in the start of the middle part (2). Following this, there are several methods being called through the variable: some of them (e.g. addParticipator) take a comma-separated string value (e.g. "Property: Item", as denoted in Figure 6-3), initializing and adding a participant object; others (e.g. setParameter) similarly take a string value (e.g. "Collector_Margin_Top = 25"). Finally, the Attachable instance is loaded into the system runtime, as shown by the last line of (2). Similar processes are carried out in the bottom part (3) as is presented in (2).

6.4.2 Implementing a constraint handler

To implement a custom constraint handler, the first and most critical step would be to create a Java class inheriting the base class, AbstractConstriant. An example of such a signature is given as follows:

```
public class Attachable extends AbstractConstraint
```

Then, the developer would overload the AddParticipator method depending on what kinds of the participants and how many of them that the constraint handler can cope with. An example of such an overloaded method is shown as follows:

```
public void addParticipator(Participator participator)
{
      super.addParticipator(participator); [1]

      if(participator.role.toLowerCase().contains("source")) [2]
      {
            sourceTypes.add((String) participator.getIParticipator());
```

```
        }
        else if(participator.role.toLowerCase().contains("target"))
        {
                targetTypes.add((String) participator.getIParticipator());
        }
}
```

Typically, a duplicate call would be made to the parent method (1), and then the rest of code, starting from (2), would be responsible for initializing the related inner data items based on the object that is passed into the method.

After that, the developer would similarly overload the setParameter method with respect to the control variables that the constraint handler can have. An example of such an overloaded method is listed as follows:

```
public void setParameter(Parameter parameter)
{
        super.setParameter(parameter);

        if(parameter.getName().toLowerCase().equals("position"))
        {
                position = Position.valueOf(parameter.getValue());
        }
        else if(parameter.getName().toLowerCase().equals("difference"))
        {
                String[] segments = parameter.getValue().split(",");
                difference = new Point(Integer.parseInt(segments[0].trim()),
                                Integer.parseInt(segments[1].trim()));
        }
}
```

The rest of the work would be as same as implementing a Marama event handler.

For more detailed information, please refer to Appendix A, in which the full source code of a built-in constraint handler is provided.

6.5 The Back End Generation Module

The Back End Generation module contains facilities for generating view specification artefacts, mainly text files such as XML files and (Java) source code. Some facilities such as generation model elements provide an easy-to-access interface to view specification models. Other facilities such as JET (Java Emitter Templates) templates are used to encode the kinds of files to generate. Figure 6-7 shows the back end generation approach.

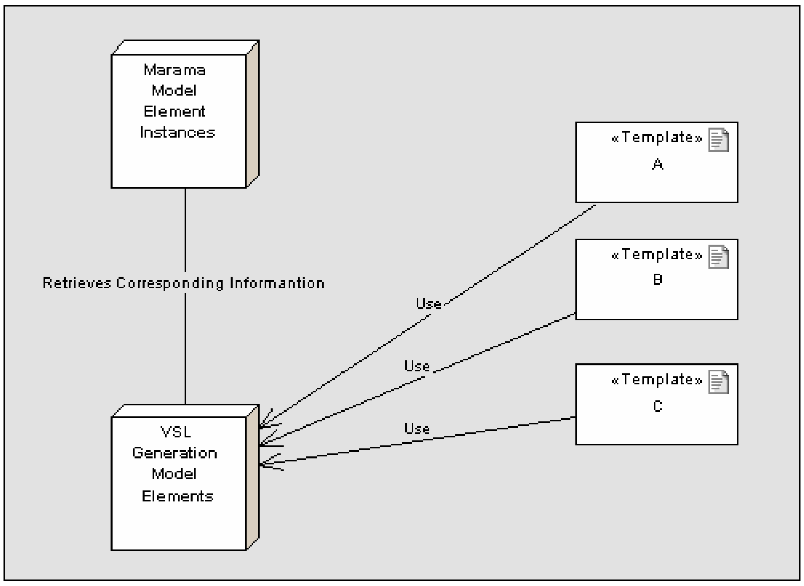
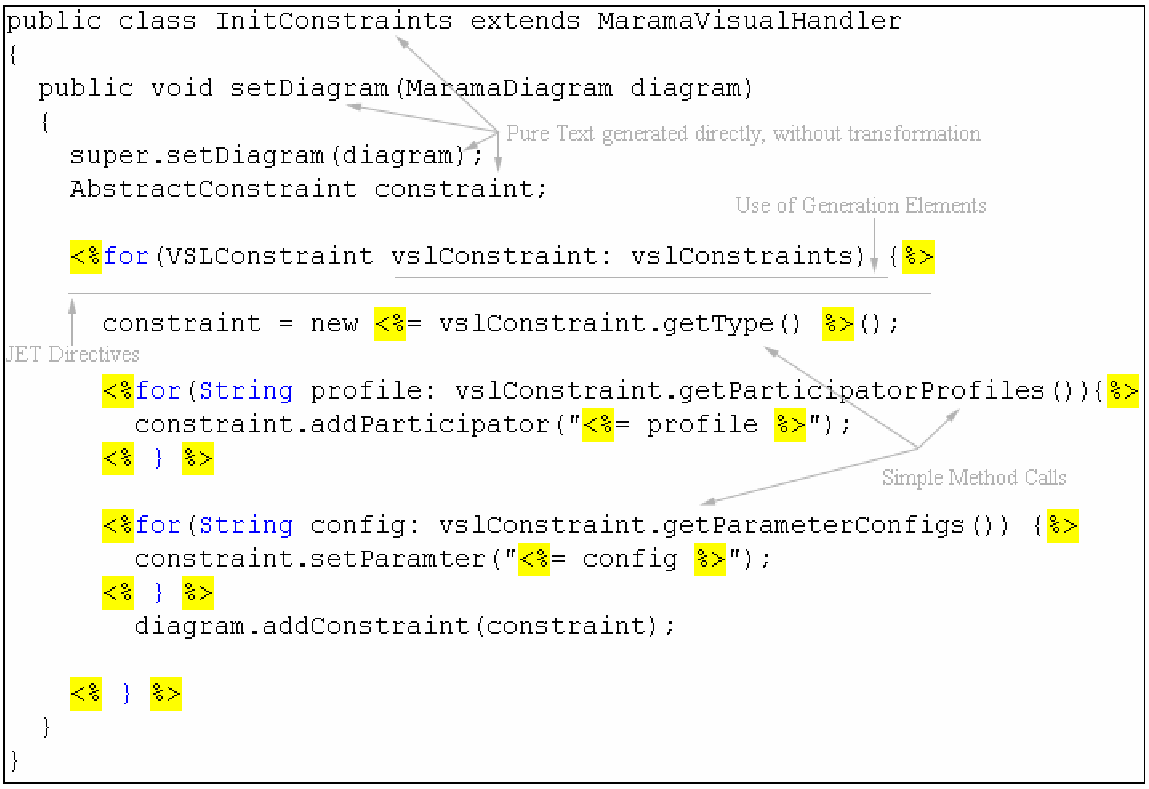

Marama Model Element Instances
«Template» A
«Template» B
«Template» C
Retrieves Corresponding Informantion
Use
Use
Use
VSL Generation Model Elements

Figure 6-7 The back end generation approach

A view specification is visually specified in the diagram but essentially stored as various Marama model elements in the common repository (as shown in Figure 6-5, top left). A generation model element (bottom left) is used to encapsulate a corresponding Marama model element, and it is specifically designed for working with Jet templates (obtaining data is much simpler than directly using a Marama model element, only a straight method call e.g. as shown in Figure 6-8). A Jet template (which is a sequence of string inserted with special directives) uses one or more generation model elements for generating template-based files.



```
public class InitConstraints extends MaramaVisualHandler
{
  public void setDiagram(MaramaDiagram diagram)
  {
    super.setDiagram(diagram);
    AbstractConstraint constraint;

    <%for(VSLConstraint vslConstraint: vslConstraints) {%>

      constraint = new <%= vslConstraint.getType() %>();

      <%for(String profile: vslConstraint.getParticipatorProfiles()){%>
        constraint.addParticipator("<%= profile %>");
      <% } %>

      <%for(String config: vslConstraint.getParameterConfigs()) {%>
        constraint.setParamter("<%= config %>");
      <% } %>
        diagram.addConstraint(constraint);

    <% } %>
  }
}
```

Pure Text generated directly, without transformation
Use of Generation Elements
JET Directives
Simple Method Calls

Figure 6-8 Part of the instantiation template

For detailed information about how to write a JET template, please refer to [EMF, 2004].

6.6 Summary

In this chapter, we discussed the design and implementation of our prototype VSL tool. We introduced the implementation background, and explained the basic design of our tool. We then detailed some core areas of our implementation, especially the Constraint module and the Back End Generation module.

This chapter is essentially intended to show how a tool for VSL could be built and how we managed to do our own prove-of-concept implementation.

CHAPTER 7

EVALUATION

7.1 Introduction

In this chapter, we provide an evaluation of the language and tool that we developed for multi-view visual environments. We first present an evaluation of our language and tool using Green and Petre's cognitive dimensions framework [Green and Petre, 1996]. We then give a brief, retrospective analysis on the strengths and weaknesses of our approach.

7.2 Cognitive Dimensions Evaluation

The cognitive dimensions framework was developed to provide a broad-brush evaluation technique for interactive devices and for non-interactive notations. It proposes a small vocabulary of terms that capture diverse and often competitive aspects of structure, especially visual programming environments. As a set of discussion tools, the framework offers a plain basis for evaluation by indicating the characteristics of cognitive artefacts that should be concerned. We will briefly introduce the cognitive dimensions and apply them to our language and tool.

Abstraction Gradient

An abstraction is a grouping of similar elements to be treated as one entity. In this case, Abstraction Gradient refers to the level of abstraction present in a given language. A language can be regarded as abstraction-hating, abstraction-tolerant or abstraction-hungry, based on the minimum starting level of abstraction and the capability to incorporate further abstractions. Choosing the proper abstraction level for a language is not easy; more abstractions may increase the comprehensibility of a language, but simultaneously decrease its simplicity.

VSL is somewhat of a mixed system in relation to abstractions, of which there are several used in the notation. Some abstractions, such as property mappers and constraints, are explicit to the user. Other abstractions, such as flows and associations, are implicit. These abstractions are carefully kept small and simple; thus, the starting level in the system is relatively low for a target user.

During the course of our language design, we were struggling with the decision whether or not to use an extra abstraction around structural elements. The use of the abstraction allows all the structural elements to be treated as a single entity. This increases the applicability of VSL, but also introduces a number of complicating issues, such as escalating the difficulty of using the notation and doubling the amount of time required to implement our tool.

Closeness of Mapping

Closeness of Mapping is a measurer to reflect how easy the user can use the language to solve the corresponding real world problem. The closer the language is to the problem domain, the easier it is for the user to perform related tasks. In a good practice, the entities in the problem domain would be directly mapped to the entities in the language. Thus, the operations on those problem entities would likewise be mapped directly to the operations on those entities in the language.

Our problem domain is to use VSL as a high-level language to generate corresponding view specification artefacts. The graphical notation extends from a notation that is familiar to the domain users, using a common iconic form. Although the mechanism of specifying element properties is a bit uncommon, this would be straightforward to end users. We feel that the mapping links and constraints are intuitive in this context. There is, however, a huge step from this visual form to the underlying textual representations (e.g. XML files and Java code).

Consistency

Consistency in the context of language is regularly expressed via its antonym, “inconsistency”, as it is initially difficult to define. A language is considered inconsistent if it represents similar concepts or performs related tasks in an unpredictable manner. An easy way to measure the consistency of a language is to ask, “When a person knows some of the language, how much of the rest can be successfully guessed?”

We feel that VSL is consistent. We have not noticed any particular areas of inconsistency and there are certainly areas of planned consistency. For example, all the constraints are used in a consistent way (via the participation pattern), regardless of the strengths they enforce or the number of parties they handle. Also, the visual shapes chosen are predictable: schema elements are represented by rounded-corner rectangles, and their mappings are noted by lines.

Diffuseness/Terseness

Diffuseness is meant to measure how compactly (e.g. the number of symbols and space) the notation achieves the results with respect to other notations that achieve the same results. While the level of diffuseness that should be aimed in a notation remains unclear, both extremes of being too diffuse and too terse are unexpected. When expressed by an extremely diffuse notation, the information takes more time to read and is more difficult to remember. Whereas, when it is represented by an overly terse notation it is difficult to understand. This is because it has to be conveyed in very subtle distinctions, which increases the difficulty of readability.

We begin the language design with a clear goal that VSL will be kept as compact as possible, since a view specification diagram tends to be more diffuse with the addition of entities. Because of this reason, we abandoned several alternative symbols (e.g. Mapping Console and Formula) in our final design. It is difficult to judge whether VSL is diffuse or terse without comparing it to another language. Generally, VSL is less diffuse than JComposer [Grundy et al., 1998a]. It is, however, more diffuse than other, less-related visual languages such as MetaBuilder [Ferguson et al., 2000], due to the fact that an entity is usually represented by two symbols in VSL – a visual symbol and a model symbol. On the whole, however, we feel that VSL is relatively terse.

Error-proneness

Errors can be further separated as mistakes and slips. A slip is doing something that you didn't mean to do (where you would normally know what to do), whereas a mistake is almost unavoidable in situations where performing tasks is extremely difficult. Error-proneness refers to how likely the user will make a slip in the course of using the language. Some typical examples of slips are often found in textual programming languages, e.g. typing errors and paired-delimiter systems.

Although visual languages in general are more immune to these types of errors, we notice one unavoidable issue is common. That is, it is quite easy for users to make slips in specifying inter-property mappings. This is often caused by specifying too many inter-property mappings in a small area in which the user might pick a wrong property to begin or end with. Unfortunately, we were not able to introduce any effective means in the tool to detect and avoid this kind of slip because they remain correct in the syntax.

Hard Mental Operations

Hard Mental Operations refer to tasks that are caused hard to accomplish or concepts that are caused difficult to comprehend by the notation. To test hard mental operations, one must first combine two or three constructs to see if the end result gets incomprehensible. If the answer is "yes", he must then try to identify whether there is a way in some other language to make this result comprehensible. Unless no trace is found (it is actually a complicated problem to deal with), then it is valid.

We feel that VSL is generally easy to use; the mapping lines and constraints are intuitive and fairly easy to understand. Still, there is some complicity in the use of combining different flow elements (e.g. a Generation Flow with a Search Flow). It is certain this would be a bit difficult for the user to comprehend. However, we cannot conclude whether it falls into the category of creating a "hard mental operation" because none of the languages we found supports the feature.

Hidden Dependencies

A hidden dependency is an invisible relationship between two components so that one of them is dependent on the other. The main reason to avoid hidden dependencies is to teach users how to make each operation and what impacts are caused by that operation. Without knowing all the information, a user is often confused and is more prone to perform wrong (perhaps even harmful) actions in the system.

This type of dependency may occur in VSL due to the declarative nature of the specification language. Some flow elements, such as Search Flow and Reference Flow, are transferred into corresponding artefacts that are hidden in the background. Moreover, the use of various constraints hides large amounts of tedious specification details behind the scenes. However, in our experience we feel that these kinds of dependencies are unlikely to cause the user to make any serious mistakes.

Premature Commitment

Premature Commitment implies that the user is forced to make a decision before the relevant information is made available. The problem likely arises when an order of doing things is enforced by the working environment that contains a large number of internal dependencies in the notation, rendering the order inappropriate.

As a specification language, VSL constraints a few orders that are almost impossible to avoid. For example, the user should have a set of structure elements ready before the mapping specification can be laid out. Fortunately, most of these orders are obvious to target users, and would be appropriate in this occasion. However, VSL (like many visual languages) encounters some problems related to the physical layout of the diagrammatic elements making up the notation. Questions such as "where to place the first icon?" and "which icon should be picked up over another?" can be difficult, especially when a user does no have a solid understanding of the diagram.

Nevertheless, there is one area that may fall into the category of creating a "premature commitment". That is, the user would be expected to place and define an Input prior to the creation of a corresponding Reference Flow. This tends to be a problem because the Reference Flow is dependent on the Input, but the required information is not fully visible.

Progressive Evaluation

Progressive Evaluation refers to the ability to test partial systems in the course of development. Immediate feedback assists a user in identifying potential problems that are difficult to debug and solve in a later stage, especially when a user has less experience in the task domain.

This is a particularly difficult aspect to evaluate at the langue level but the tool level. Once the user starts placing icons in the diagram, the tool can generate the view specification artefacts at any time. However, (as the limited time involved in this thesis work) the tool currently does not give any meaningful feedback to the user.

Role-Expressiveness

Role-Expressiveness is intended to describe how clearly the notation expresses itself in a variety of roles. Such a role can refer to a single entity or a combination of several different entities. A common way to enhance role-expressiveness is realized by the use of meaningful or well-known identities.

We feel that the role-expressiveness is good in VSL. The shapes chosen are familiar to target users, using a common iconic form. Functional related elements are represented by similar shapes with carefully chosen secondary aspects, such as different colours, icons, or both. This further improves the role-expressiveness by visually dividing the objects contained in a diagram into functional groups.

VSL tool is implemented as a multi-view visual environment in which multiple views of a single model are allowed. It is possible to create a modularised model, with each view of the model displaying a single related group of entities from that model.

Secondary Notation and Escape from Formalism

Secondary Notation refers to the manner in which extra information is carried by means other than the formal syntax of the language. For example, programmers often use indents, extra brackets and new lines to make the code easy to understand by others (even themselves). Escape from Formalism is closely related to secondary notation. It is meant to indicate whether the language contains the power to specify the information that is unsupported by its formalism. The most common example of this is the use of comments in many programming languages.

In VSL we feel that the secondary notation is relatively weak, due to the box-and-line design. The structure elements can, in principle, be laid out anywhere in a diagram, but in practice the layout is dominated by the desire to keep the mapping flow lines reasonably tidy. We doubt it would be a good idea to enforce the layout in this case; nevertheless, doing it manually often requires too much effect from the user. Fortunately, we feel in VSL that we performed well with regard to the escape from formalism. Comments can be attached to primitives as well as groups of objects.

Viscosity: Resistance to Local Change

Viscosity implies how much work the user has to make in order to perform a small change, especially when the change impacts a large amount of other changes. In general, the environment with its comprehensive support and the language having more abstractions may act well in viscosity.

We feel that VSL has low viscosity. Changing mapping lines is easy for the user to perform. Although the steps involved in reassigning constraints may vary depending on the actual case, all the operations required are both simple and obvious. Altering structural elements in few cases may cause some difficulty, but the situations (e.g. a structural element is used completely wrong, as required to change all its properties and associated mappings and constraints) are extremely unlikely to happen.

There are also certain supports shipped with the tool environment. Joined connections are automatically eliminated after corresponding icons are deleted. If an entity is dropped from the repository, all the corresponding icons disappear in the related diagrams. Most importantly, in the VSL tool updates are automatically in turn propagated to any other components that need to change in response to the update, which makes the change even easier to perform. However, we were restricted by the limited time involved in this thesis work; so many helpful facilities have not been supplied, although concrete ideas were in place.

Visibility and Juxtaposability

Visibility denotes simply whether desired information is reachable without cognitive work. As compared to hidden dependencies, visibility is a measure of the number of steps required to discover and display a piece of information. Juxtaposability is an important part of visibility. It refers to the ability to see and compare any two portions of the program on screen next to each other at the same time.

We feel in VSL that the visibility is great. Most information is readily available and expressed by the (VSL) diagrams. Although the other information is hidden in the background, this information is fairly easy for the user to access through the property sheet. The VSL tool certainly has good juxtaposability as well, with the ability to have multiple views open side by side, displaying different parts of the same view specification. In fact, the comparison can be done even easier through the corresponding facility that is supplied by the tool environment.

7.3 General Comments

On the whole, we are very pleased with both the language and the tool we have created. By the large, we feel the approach we have taken is very promising and offers several unique benefits over other approaches to providing view specification support for multi-view visual environments.

First, we feel the fact that VSL as a visual language is a major benefit. The visual notation provides an intuitive medium for communication about view definition. The specification diagrams can be rapidly and simply created, either on paper or on a computer screen, and even more easily interpreted and understood. We feel that VSL is clearly easier to use than other textual languages that have been designed for the same purpose.

Second, we feel that VSL offers more advanced mapping support over related visual languages. Complex view-to-model mappings are explicitly allowed; thus, it is easier for the user to create various view types according to a common model, as is often required. For example, a diagram component may be created with external values passed in, may be associated to multiple model components, or may not be physically related to any model component whatsoever.

Third, we feel that the use of a declarative style in VSL is a strong point of the approach. The visual language hides much of the complicity of view specifications; therefore, the user can concentrate on the high level design of the view specification rather than the underlying implementation techniques. Moreover, the use of common-targeted visual constraints further reduces the effort involved and lowers the entry level required for end users to start with.

Thus, we feel that the high-level, declarative approach to Pounamu view specification and generation is particularly good. The use of our VSL tool significantly reduces the view specification effort required for developing Pounamu-based multi-view visual environments.

However, we still have some reservations. First of all, the visual language has a few issues that have been noted in the cognitive dimensions evaluation. Secondly, we feel that there are still some improvements that can be made in the language, based on our experiences throughout this thesis study. Finally, our tool currently does not support all the features that are defined in VSL.

Our suggestions for future extensions to the language and the tool are discussed in the next chapter.

7.4 Summary

In this chapter, we carried out an evaluation of our VSL language and the prototype VSL tool that we have developed. We first applied the cognitive dimensions framework in this evaluation, and then generally discussed the strengths and weaknesses of our approach.

CHAPTER 8

CONCLUSION AND FUTURE WORK

8.1 Overall Summary

In this thesis, we have investigated the issues involved in providing view specification support for the development of multi-view visual environments. We also presented a high-level, declarative approach to Pounamu view specification and generation.

We began with an overview of the background technology and a comprehensive review of the related research into view specification support for multi-view visual environments, with a main focus on view consistency management (an essential component of view specification). Based on the knowledge learned from this review, we outlined our plan of solution to improve Pounamu view specification, which is used as the motivation and exemplar application of our ideas (which have broader applicability). After discussing the experience we learned during our language design, we then presented the View Specification Language (VSL), a visual modelling language we created for view specification, and several typical VSL examples. Following this, we presented a prototype implementation of a VSL tool, which can be used for Pounamu-based view generation. Finally, we presented an evaluation of the prototype VSL tool and VSL itself, where we argue that VSL has great potential as a modelling language to facilitate the effort of view specification and generation in developing multi-view visual environments.

8.2 Contributions of this Thesis

The main contributions of this thesis are:

- We provide VSL, a domain-specific visual language, for view specification. The notation contained provides an intuitive medium for communication about view definition. The specification diagrams can be rapidly and simply created, on either paper or a computer screen, and easily comprehended and interpreted. More importantly, VSL offers many high-level features such as advanced mapping support (e.g. *Generation Flow*, *Search Flow* and *Reference Flow*) and easy-to-apply visual control (e.g. various *Constraints*). These features reduce the difficulty of view specification and lower the entry level requirements for starting users.

- We also provide a high-level, declarative approach to Pounamu view specification and generation. This approach uses VSL as its modelling language; it provides many high-level, easy-to-use facilities; it produces diagrams (rather than, for example, textual artefacts); and it automatically generates Pounamu-based view specification artefacts as specified. We feel that our approach has significantly reduced the view specification effort required for developing Pounamu-based multi-view visual environments.

8.3 Suggestions for Future Work

There are a variety of directions in which this project could be further extended. Here we wish to highlight a number of areas of possible further research we have identified. Some of the areas we discuss are improvements or natural extensions of our work, while others are possible applications of our work.

In terms of the future work that could be done on the language, there are many possible areas of extension. Some that have occurred to us are:

- Divide the single VSL notation into a set of functional distinct notations, each of which is responsible for a specific perspective of view definition, so that a view definition can be expressed by a clear set of different diagrams (views). This could lead to a better clarity of view specification, and thus reduce the error-proneness.

- Provide an abstract intermediate mapping layer (e.g. allowing use of abstract model elements, each of which may partially or completely represent one or more concrete model elements) between view elements and corresponding model elements, so the complexity of a range of view-to-model mappings could be reduced.

- Introduce more common-required (visual) constraints, possibly a full coverage of those used by UML, and enhance the current VSL constraint extension mechanism, perhaps adopting a flow-based approach, e.g. Kaitiaki [Liu et al., 2005].

- Investigate how to migrate VSL to other multi-view development systems that are similar to Pounamu, and how to apply VSL to a range of other applications such as translation of notational elements into different terminals and translation of notational elements in general.

There are also areas of possible future work involved in the processes associated with the language and supported by a VSL tool. They are listed as follows:

- Better (diagram) editing support, such as automatic completion of simple specification takes, smart layout of diagram components, and advanced visualization of diagrams (e.g. selectively emphasizing part of a diagram).

- Support for automated mapping (e.g. automatically matching a view element/property to a potential model element/property), perhaps even an investigation into some practical techniques in Artificial Intelligence.

- Further investigation into the tool integration with other systems, possibly incorporating a standard set of back end generation techniques, such as XSLT for XML documents, Jet for programming source code (currently used), and QVT-based methods for model transformation.

APPENDIX A

Source code of the Attachable constraint handler

```
package nz.ac.auckland.cs.VSL.Constraint;

import java.util.ArrayList;
import java.util.List;
import java.util.Vector;

import nz.ac.auckland.cs.marama.model.diagram.MaramaConnection;
import nz.ac.auckland.cs.marama.model.diagram.MaramaShape;

import org.eclipse.draw2d.geometry.Dimension;
import org.eclipse.draw2d.geometry.Point;
import org.eclipse.draw2d.geometry.Rectangle;
import org.eclipse.emf.common.notify.Notification;
import org.eclipse.gef.commands.Command;

public class Attachable extends AbstractConstraint
{
        private Vector<MaramaShape> renderingSources = new Vector<MaramaShape>();

        // Participators as roles of string shape type
        public List<String> sourceTypes = new ArrayList<String>();
        public List<String> targetTypes = new ArrayList<String>();

        // VSL location connector
        private String connectorType = "Constraint_Association";

        // Parameters as specific types
        public Position position = Position.Inner;
        public Point difference = null;

        public boolean isTargetAutoLocated = false;

        public boolean isSourceChangeable = false;

        public boolean isSourceAutoSpanned = false;

        public int source_InnerMargin_Right = 0;
        public int source_InnerMargin_Bottom = 0;

        public boolean isTargetMoveable = false;

        public boolean isMultiTargetsAllowed = false;
        public Orientation extentionOrientation = Orientation.Vertical;
        public int target_Spacing = 0;

        @Override
        public void addParticipator(Participator participator)
        {
                super.addParticipator(participator);

                if(participator.role.toLowerCase().contains("source"))
                {
                        sourceTypes.add((String) participator.getIParticipator());
                }
                else if(participator.role.toLowerCase().contains("target"))
                {
                        targetTypes.add((String) participator.getIParticipator());
                }
        }

        @Override
        public void setParameter(Parameter parameter)
        {
                super.setParameter(parameter);

                if(parameter.getName().toLowerCase().equals("position"))
                {
                        position = Position.valueOf(parameter.getValue());
                }
                else if(parameter.getName().toLowerCase().equals("difference"))
                {
                        String[] segments = parameter.getValue().split(",");

                        difference = new Point(Integer.parseInt(segments[0].trim()),
Integer.parseInt(segments[1].trim()));
                }
                else
if(parameter.getName().toLowerCase().equals("isTargetAutoLocated".toLowerCase()))
                {
                        isTargetAutoLocated = Boolean.parseBoolean(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("isSourceChangeable".toLowerCase()))
                {
                        isSourceChangeable = Boolean.parseBoolean(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("isSourceAutoSpanned".toLowerCase()))
                {
                        isSourceAutoSpanned = Boolean.parseBoolean(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("source_InnerMargin_Right".toLowerCase()))
                {
                        source_InnerMargin_Right = Integer.parseInt(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("source_InnerMargin_Bottom".toLowerCase()))
                {
                        source_InnerMargin_Bottom = Integer.parseInt(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("isTargetMoveable".toLowerCase()))
                {
                        isTargetMoveable = Boolean.parseBoolean(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("isMultiTargetsAllowed".toLowerCase()))
                {
                        isMultiTargetsAllowed = Boolean.parseBoolean(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("extentionOrientation".toLowerCase()))
                {
                        extentionOrientation = Orientation.valueOf(parameter.getValue());
                }
                else
if(parameter.getName().toLowerCase().equals("target_Spacing".toLowerCase()))
                {
                        target_Spacing = Integer.parseInt(parameter.getValue());
                }
        }
```

```
@Override
public void beforeExecute(Command command)
{
	MaramaShape shapeChanged = shapeChanged(command);
	if(shapeChanged != null)
	{
		boolean isTarget = isTarget(shapeChanged);

		if(shapeDeleted(command) != null)
		{
			if(isTarget) targetDeleted(shapeChanged);
		}
	}
}

@Override
public void afterExecute(Command command)
{
	if(this.renderingSources.size() != 0)
	{
		for(MaramaShape source : this.renderingSources)
		{
			this.render(source);
		}
		this.renderingSources.clear();
	}
}

@Override
public void notifyChanged(Notification e)
{
	MaramaShape shapeChanged = shapeChanged(e);
	if(shapeChanged != null)
	{
		boolean isSource = isSource(shapeChanged);
		boolean isTarget = isTarget(shapeChanged);

		if(shapeAdded(e) != null)
		{
			if(isSource) sourceAdded(shapeChanged);
			if(isTarget) targetAdded(shapeChanged);
		}
```

```
                    if(shapeResized(e) != null)
                    {
                            if(isSource) sourceResized(shapeChanged);
                            if(isTarget) targetResized(shapeChanged);
                    }

                    if(shapeMoved(e) != null)
                    {

                            if(isSource) sourceMoved(shapeChanged);
                            if(isTarget) targetMoved(shapeChanged, (Point)
e.getOldValue());
                    }

                    if(shapeDeleted(e) != null)
                    {
                            if(isSource) sourceDeleted(shapeChanged);
                            //if(isTarget) targetDeleted(shapeChanged);
                    }
            }
        }

        // Source events start
        protected void sourceAdded(MaramaShape source)
        {
                this.autoLocate();
        }

        protected void sourceResized(MaramaShape source)
        {
                this.render(source);
        }

        protected void sourceMoved(MaramaShape source)
        {
                this.render(source);
        }

        protected void sourceDeleted(MaramaShape source)
        {
                this.autoLocate();
        }
        // Source events end
// Target events start
protected void targetAdded(MaramaShape target)
{
        MaramaShape source = this.getFirstAvailableSourceByIntersection(target);

        if(source != null)
        {
                this.addTarget(source, target);
                this.render(source);
        }
        else this.autoLocate();
}

protected void targetResized(MaramaShape target)
{
        MaramaShape source = this.getSourceByTarget(target);

        if(source != null)
                this.render(source);
}

protected void targetMoved(MaramaShape target, Point orginalTargetLocation)
{
        MaramaShape orginalSource = this.getSourceByTarget(target);
       MaramaShape newSource = this.getFirstAvailableSourceByIntersection(target);

        // Move into a source
        if(orginalSource == null && newSource != null)
        {
                this.addTarget(newSource, target);
                this.render(newSource);
        }
        else if(orginalSource != null && newSource != null &&
                        !orginalSource.getBounds().intersects(target.getBounds())&&
                        orginalSource != newSource)
        { //From a container move into another container
                if(this.isSourceChangeable)
                {
                        this.removeTarget(orginalSource, target);
                        this.render(orginalSource);
                        this.addTarget(newSource, target);
                        this.render(newSource);
```

```
                    }
                    else this.render(orginalSource);
                }
                else if(orginalSource != null)
                { // Move while connected to a source
                        if(this.isTargetMoveable)
                        {// if the position is inner, only allow move inside the source
                                Dimension difference =
target.getBounds().getLocation().getDifference(orginalTargetLocation);
                                Rectangle bounds =
                                        new
Rectangle(orginalSource.getBounds().getLocation().translate(difference),
orginalSource.getSize());
                                this.setShapeConstraint(orginalSource, bounds);
                        }
                        else this.render(orginalSource);
                }
        }

        protected void targetDeleted(MaramaShape target)
        {
                MaramaShape source = this.getSourceByTarget(target);

                if(source != null)
                        this.renderingSources.add(source);
        }
        // Target events end

        protected boolean isSource(MaramaShape shape)
        {
            if(shape == null)
            {
                throw new NullPointerException("shape");
            }

            return sourceTypes.contains(shape.getShapeType());
        }

        protected boolean isTarget(MaramaShape shape)
        {
                if(shape == null)
            {
                throw new NullPointerException("shape");
```

```
    }

    return targetTypes.contains(shape.getShapeType());
}

protected void autoLocate()
{
        if(this.isTargetAutoLocated)
        {

        }
}

protected void render(MaramaShape source)
{
        List<MaramaShape> targets = this.getTargetsBySource(source);

        if(targets.size() == 0)
                return;

        setEnabled(false);

        Point location = source.getBounds().getLocation();

        if(position == Position.Bottom)
        {
                location.y += source.getHeight();
        }
        else if(position == Position.Right)
        {
                location.x += source.getWidth();
        }
        else
        {
                // Do nothing for the rest
        }

        if(difference != null)
                location.translate(difference);

        List<MaramaConnection> connectors = this.getConnectorsBySource(source);
        for (MaramaConnection connector : connectors)
        {
```

```
                        if(!connector.isHidden()) connector.setHidden(true);
                        MaramaShape target = connector.getTarget();
                        this.assignTargetIndex(source, target);

                        Point targetLocation = location.getCopy();

                        if(this.position == Position.Top)
                        {
                                targetLocation.y -= target.getHeight();

                                this.setShapeConstraint(target, new
Rectangle(targetLocation, target.getSize()));

                                if(extentionOrientation == Orientation.Vertical)
                                        location.y -= (target.getHeight() +
this.target_Spacing);
                                else
                                        location.x += (target.getWidth() +
this.target_Spacing);
                        }
                        else if(this.position == Position.Bottom)
                        {
                                this.setShapeConstraint(target, new
Rectangle(targetLocation, target.getSize()));

                                if(extentionOrientation == Orientation.Vertical)
                                        location.y += (target.getHeight() +
this.target_Spacing);
                                else
                                        location.x += (target.getWidth() +
this.target_Spacing);
                        }
                        else if(this.position == Position.Left)
                        {
                                targetLocation.x -= target.getWidth();

                                this.setShapeConstraint(target, new
Rectangle(targetLocation, target.getSize()));

                                if(extentionOrientation == Orientation.Vertical)
                                        location.y += (target.getHeight() +
this.target_Spacing);
                                else
```

```
                                        location.x -= (target.getWidth() +
this.target_Spacing);
                        }
                        else if(this.position == Position.Right)
                        {
                                this.setShapeConstraint(target, new
Rectangle(targetLocation, target.getSize()));

                                if(extentionOrientation == Orientation.Vertical)
                                        location.y += (target.getHeight() +
this.target_Spacing);
                                else
                                        location.x += (target.getWidth() +
this.target_Spacing);
                        }
                        else if(this.position == Position.Inner)
                        {
                                this.setShapeConstraint(target, new
Rectangle(targetLocation, target.getSize()));

                                if(extentionOrientation == Orientation.Vertical)
                                        location.y += (target.getHeight() +
this.target_Spacing);
                                else
                                        location.x += (target.getWidth() +
this.target_Spacing);
                        }
                }

                if(this.isSourceAutoSpanned)
                {
                        // Get ride of the last spare spacing
                        if(extentionOrientation == Orientation.Vertical)
                                location.y -= this.target_Spacing;
                        else
                                location.x -= this.target_Spacing;

                        Rectangle bounds = source.getBounds();
                        MaramaShape theHigestShape = this.findHighestShape(targets);
                        MaramaShape theWidestShape = this.findWidestShape(targets);

                        if(this.position == Position.Top || this.position ==
Position.Bottom)
```

```
                        {
                                if(extentionOrientation == Orientation.Vertical)
                                {
                                        if(bounds.width < theWidestShape.getWidth())
                                                bounds.width = theWidestShape.getWidth();
                                }
                                else
                                {
                                        if(bounds.width < location.x - bounds.x)
                                                bounds.width = location.x - bounds.x;
                                }
                        }
                        else if(this.position == Position.Left || this.position ==
Position.Right)
                        {
                                if(extentionOrientation == Orientation.Vertical)
                                {
                                        if(bounds.height < location.y - bounds.y)
                                                bounds.height = location.y - bounds.y;
                                }
                                else
                                {
                                        if(bounds.height < theHigestShape.getHeight())
                                                bounds.height = theHigestShape.getHeight();
                                }
                        }
                        else if(this.position == Position.Inner)
                        {
                                if(extentionOrientation == Orientation.Vertical)
                                {
                                        if(bounds.width <= theWidestShape.getWidth())
                                                bounds.width = theWidestShape.getWidth() +
this.source_InnerMargin_Right;

                                        if(bounds.height <= location.y - bounds.y)
                                                bounds.height = location.y - bounds.y +
this.source_InnerMargin_Bottom;
                                }
                                else
                                {
                                        if(bounds.width <= location.x - bounds.x)
                                                bounds.width = location.x - bounds.x +
this.source_InnerMargin_Right;
```

```
                                if(bounds.height <= theHigestShape.getHeight())
                                        bounds.height = theHigestShape.getHeight() +
this.source_InnerMargin_Bottom;
                        }
                }

                this.setShapeConstraint(source, bounds);
        }

        setEnabled(true);
    }

    // Helpers start
    private MaramaShape findHighestShape(List<MaramaShape> shapes)
    {
        if(shapes == null || shapes.size() == 0)
            throw new NullPointerException("Parameters");

        MaramaShape theHighestShape = null;

        for(MaramaShape shape : shapes)
        {
            if(theHighestShape == null)
                theHighestShape = shape;
            else
            {
                if(shape.getHeight() > theHighestShape.getHeight())
                    theHighestShape = shape;
            }
        }

        return theHighestShape;
    }

    private MaramaShape findWidestShape(List<MaramaShape> shapes)
    {
        if(shapes == null || shapes.size() == 0)
            throw new NullPointerException("Parameters");

        MaramaShape theWidestShape = null;

        for(MaramaShape shape : shapes)
        {
                if(theWidestShape == null)
                        theWidestShape = shape;
                else
                {
                        if(shape.getWidth() > theWidestShape.getWidth())
                                theWidestShape = shape;
                }
        }

        return theWidestShape;
}

private boolean isSourceFull(MaramaShape source)
{
        if(source == null)
                throw new NullPointerException("Parameters");

        if(this.isMultiTargetsAllowed)
        {
                return false;
        }
        else
        {
                if(getTargetsBySource(source).size() == 0)
                        return false;
        }

        return true;
}

@SuppressWarnings("unchecked")
private MaramaShape getFirstAvailableSourceByIntersection(MaramaShape target)
{
        if(target == null)
                throw new NullPointerException("Parameters");

    List<MaramaShape> shapes = getDiagram().getChildren();

    for(MaramaShape shape : shapes)
    {
        if(shape != target &&
                        isSource(shape) &&
                !isSourceFull(shape) &&
                        shape.getBounds().intersects(target.getBounds()))
        {
                return shape;
        }
    }

    return null;
}

private MaramaShape getSourceByTarget(MaramaShape target)
{
        if(target == null)
                throw new NullPointerException("Parameters");

        MaramaConnection connector = null;

        try
        {
                connector = this.getConnectorByTarget(target);
        }
        catch (Exception e)
        {
                e.printStackTrace();
        }

        if(connector != null)
        {
                return connector.getSource();
        }

        return null;
}

private void addTarget(MaramaShape source, MaramaShape target)
{
        if(source == null || target == null)
                throw new NullPointerException("Parameters");

        this.createNewConnection(this.connectorType, source, target);
}

private void removeTarget(MaramaShape source, MaramaShape target)
{
        if(source == null || target == null)
                throw new NullPointerException("Parameters");

        this.deletePositioningConnector(target);
}

@SuppressWarnings("unchecked")
private List<MaramaShape> getTargetsBySource(MaramaShape source)
{
        if(source == null)
                throw new NullPointerException("Parameters");

        List<MaramaShape> targets = new ArrayList<MaramaShape>();

        List<MaramaConnection> connections =
source.getSourceConnections(this.connectorType);
        for (MaramaConnection connection : connections)
        {
                if(isConnector(connection))
                        targets.add((MaramaShape) connection.getTarget());
        }

        return targets;
}

private void assignTargetIndex(MaramaShape container, MaramaShape item)
{
        if(container == null || item == null)
                throw new NullPointerException("Parameters");

        if(container.getIndex() > item.getIndex())
                item.setIndex(container.getIndex());
}

private boolean isConnector(MaramaConnection connection)
{
        if(connection == null)
        {
                throw new NullPointerException("connection");
        }

        if(this.connectorType.equals(connection.getConnectionType()) &&
                        isSource(connection.getSource()) &&
                        isTarget(connection.getTarget()))
        {
                return true;
        }

        return false;
}

private void deletePositioningConnector(MaramaShape target)
{
        if(target == null)
                throw new NullPointerException("Parameters");

        try
        {
                this.deleteConnector(this.getConnectorByTarget(target));
        }
        catch (Exception e)
        {
                e.printStackTrace();
        }
}

@SuppressWarnings("unchecked")
private List<MaramaConnection> getConnectorsBySource(MaramaShape source)
{
        if(source == null)
                throw new NullPointerException("Parameters");

        List<MaramaConnection> connectors = new ArrayList<MaramaConnection>();
        List<MaramaConnection> connections =
source.getSourceConnections(this.connectorType);

        for(MaramaConnection connection : connections)
        {
                if(isConnector(connection)) connectors.add(connection);
        }

        return connectors;
}

@SuppressWarnings("unchecked")
private MaramaConnection getConnectorByTarget(MaramaShape target) throws Exception
{
        if(target == null)
                throw new NullPointerException("Parameters");

        List<MaramaConnection> connectors = new ArrayList<MaramaConnection>();
        List<MaramaConnection> connections =
target.getTargetConnections(this.connectorType);

        for(MaramaConnection connection : connections)
        {
                if(isConnector(connection)) connectors.add(connection);
        }

        if(connectors.size() == 1)
                return connectors.get(0);

        if(connectors.size() > 1)
                throw new Exception("More than one source have been found on the
target");

        return null;
}
// Helpers end

@Override
public MaramaConnection createNewConnection(String type, MaramaShape parent,
MaramaShape child)
{
        if(this.isExecuting())
                return super.createNewConnection(type, parent, child);

        return null;
}

@Override
public void deleteConnector(MaramaConnection shape)
{
        if(this.isExecuting())
                super.deleteConnector(shape);
}

@Override
public void setShapeConstraint(MaramaShape shape, Rectangle newBounds)
{
        if(this.isExecuting())
                super.setShapeConstraint(shape, newBounds);
}
}
```

LIST OF REFERENCES

[Amor, 1997]	R. Amor, "A Generalised Framework for the Design and Construction of Integrated Design Systems", Ph.D. thesis, Department of Computer Science, University of Auckland, Auckland, New Zealand, 1997, 350pp.
[Eclipse, 2006]	Eclipse Consortium, Eclipse Platform – Version 3.1, 2006, available at http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html.
[EMF, 2004]	Eclipse Consortium, Eclipse Modelling Framework (EMF) – Version 2.0.2, 2004, available at http://www.eclipse.org/emf.
[Ferguson et al., 1999]	R. Ferguson, N. Parrington, P. Dunne, J. Archibald and J. Thompson, "MetaMOOSE: an object-oriented framework for the construction of CASE tools", in Proc Int Symp on Constructing Soft. Eng. Tools (CoSET'99) LA, May 1999.
[Ferguson et al., 2000]	R.I. Ferguson, A. Hunter and C. Hardy, "Metabuilder: The diagrammer's diagrammer", in: M. Anderson, P. Cheng and V. Haarslev, editors, Theory and Application of Diagrams (2000), pp. 407-421.
[GEF, 2004]	Eclipse Consortium, Eclipse Graphical Editing Framework (GEF) – Version 3.1.1, 2004, available at http://www.eclipse.org/gef.
[Glinert, 1990]	E.P. Glinert, "*Visual programming environments*", paradigms and systems, New York: IEEE Computer Society Press, 1990.

[Green and Petre, 1996] T.R.G. Green and M. Petre, "Usability analysis of visual programming environments: a 'cognitive dimensions' framework", Journal of Visual Languages and Computing 1996 (7), pg 131-174, (1996).

[Grundy et al., 1998a] J.C. Grundy, W.B. Mugridge and J.G. Hosking, "Visual specification of multiple view visual environments", In Proc IEEE VL'98, Halifax, Nova Scotia, Sept 1-4, 1998, IEEE CS Press, pp. 236-243.

[Grundy et al., 1998b] J.C. Grundy, W.B. Mugridge and J.G. Hosking, "Inconsistency Management for Multiple-View Software Development Environments", IEEE Transactions on Software Engineering, 24 (11), Nov. 1998, 960-981.

[Grundy et al., 2000] J.C. Grundy, W.B. Mugridge and J.G. Hosking, "Constructing component-based software engineering environments: issues and experiences", Information and Software Technology, 42 (2), January 2000, Elsevier.

[Grundy and Hosking, 2001] J.C. Grundy and J.G. Hosking, "Software Tools", In Wiley Encyclopedia of Software Engineering, 2nd Edition, Wiley Interscience, December 2001.

[Grundy et al., 2001] J.C. Grundy, W.B. Mugridge, J.G. Hosking and P. Kendal, "Generating EDI Message Translations from Visual Specifications", In *Proceedings of the 2001 IEEE Automated Software Engineering Conference*, San Diego, 26-29 Nov, IEEE CS Press, 2001, 35-42.

[Grundy et al., 2004] J.C. Grundy, J.G. Hosking, R. Amor, W.B. Mugridge and M. Li, "Domain specific visual languages for specifying and generating data mapping system", Journal of Visual Languages and Computing, vol. 15, no. 3-4, June-August 2004, Elsevier, pp 243-263.

[Grundy et al., 2006] J.C. Grundy, J.G. Hosking, N. Zhu, and N. Liu. “Generating Domain-Specific Visual Language Editors from High-level tool Specifications”. 2006.

[Hill et al., 1994] R.D. Hill, T. Brinck, S.L. Rohall, J.F. Patterson and W. Wilner, “The Rendezvous Architecture and Language for Constructing Multi-User Applications,” *ACM Transactions on Computer-Human Interaction*, vol. 1, no. 2, 81-125, June 1994.

[Kelly et al., 1996] S. Kelly, K. Lyytinen and M. Rossi, “Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment”, in Proceedings of CAiSE'96, LNCS 1080, Springer-Verlag, Crete, Greece, May 1996.

[Klein and Schürr, 1997] P. Klein and A. Schürr: “Constructing SDEs with the IPSEN Meta Environment”, in *Proc. 8th Conf. on Software Engineering Environments SEE'97*, Los Alamitos: IEEE Computer Society Press (1997), 2-10.

[Krasner and Pope, 1988] G.E. Krasner and S.T. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, Journal of Object-Oriented Programming, vol. 1, no. 3, 8-22, 1988.

[Ledeczi et al., 2001] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle and G. Karsai, “Composing Domain-Specific Design Environments”, *Computer*, 44-51, Nov, 2001.

[Li et al., 2002] Y. Li, J.C. Grundy, R. Amor and J.G. Hosking, “A data mapping specification environment using a concrete business form-based metaphor”, In *Proceedings of the 2002 International Conference on Human-Centric Computing*, IEEE CS Press, 2002, 158-167.

[Liu et al., 2005] N. Liu, J.G. Hosking and J.C. Grundy, "A Visual Language and Environment for Specifying Design Tool Event Handling", In Proc. VL/HCC'2005, Dallas, Sept 2005.

[McIntyre, 1995] D.W. McIntyre, Design and implementation with Vampire, "*Visual Object-Oriented Programming*", Manning Publications, Greenwich, CT, USA, 1995, Ch 7, 129-160.

[McWhirter and Nutt, 1994] J.D. McWhirter and G.J. Nutt, "Escalante: An Environment for the Rapid Construction of Visual Language Applications", *Proc. VL '94*, pp. 15-22, Oct. 1994.

[Meyers, 1991] S. Meyers, "Difficulties in Integrating Multiview Editing Environments", IEEE Software, 8 (1), 1991, 49-57.

[Minas and Viehstaedt, 1995] M. Minas and G. Viehstaedt, "DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams", Proc. VL '95, 203-210 Sept. 1995.

[Peltier et al., 2000] M. Peltier, F. Ziserman and J. Bézivin, "On levels of model transformation", In XML Europe 2000, Paris, France, June 2000, Graphic Communications Association, pp. 1–17.

[UML, 2000] Unified Modeling Language (UML) Specification v1.3, Object Management Group Document formal/00-03-01, 2000, available from http://www.omg.org.

[Warmer and Kleppe, 1998] J. Warmer and A. Kleppe, "The Object Constraint Language: Precise Modeling with UML", Addison-Wesley, 1998.

[Zhu et al., 2004] N. Zhu, J.C. Grundy and J.G. Hosking, "Pounamu: a meta-tool for multi-view visual language environment construction", *Proc VL/HCC'04*, pp. 254-256.