Preprint of paper published in Encyclopedia of Software Engineering, 2nd Edition, © Wiley Interscience 2001.

Software Tools

John Grundy and John Hosking

Department of Computer Science, University of Auckland
Private Bag 92019, Auckland, New Zealand
{john-g, john}@cs.auckland.ac.nz

Abstract

Software is growing ever-more complex and new software processes, methods and products put greater demands on software engineers than ever before. The support of appropriate software tools is essential for developers to maximise their ability to effectively and efficiently deliver quality software products. This article surveys current practice in the software tools area, along with recent and expected near-future trends in software tools development. We provide a summary of tool applications during the software lifecycle, but focus on particular aspects of software tools that have changed in recent years and are likely to change in the near future as tools continue to evolve. These include the internal structure of tools, provision of multiple view interfaces, tool integration techniques, collaborative work support and the increasing use of automated assistance within tools. We hope this article will both inform software engineering practitioners of current research trends, and tool researchers of the relevant state-of-the-art in commercial tools and various likely future research trends in tools development.

Introduction

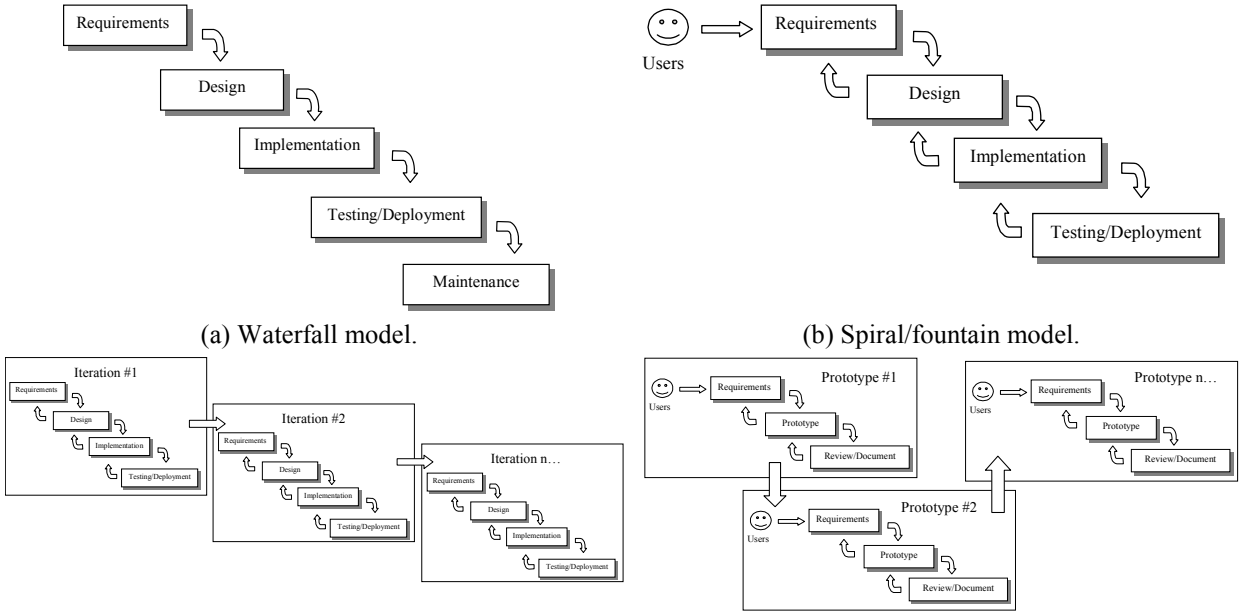
The demand for computer software applications steadily increases as numbers of users, problem domains and application areas grow [42, 51, 95]. New applications of computing technology continue to emerge [51], such as E-commerce [6], data warehousing [61], multi-media systems, mobile computing [53] and groupware [31], many very demanding in terms of software size and complexity [13]. Similarly, as computing hardware and networks become more powerful, the software that we run on them tends to grow to match (or exceed) their capacity [94]. Thus software systems continue to get larger and more complex - and software developers require improved processes, tools and techniques to manage this complexity. In addition, newer software development processes employed or proposed, such as iterative development [15], rapid applications development [75], extreme programming [13] and the personal and team software processes [54], place more demands on developers to organise their activities and software artefacts. New development methods such as component-based systems engineering [97], enterprise applications integration and aspect-oriented programming [60] require additional modelling and management support. New technologies, such as complex middleware (DCOM, CORBA, XML etc) also place more demands on developers to locate, understand, reuse and document parts of software systems during development [90]. New team organisation strategies, including open source development and virtual software teams, greatly complicate the organisation of and communication between developers, as well as the management and construction of software artefacts [107, 12].

The need for good tools to support the myriad of activities that occur during software development is thus greater than ever. Over the years software tools have been developed to support parts of the entire development lifecycle. Tools are used to describe and share software processes, plan and manage team work, capture process performance information and ultimately to improve processes. Tools assist in requirements elicitation, codification and validation. Computer-aided specification and design tools support software analysis and design, and often incorporate round-trip engineering features including code generation and reverse engineering. Programming environments and application generators provide user interface builders, database and message designers, compilers, editors and source-level debuggers. Testing tools range from profiling and performance monitoring tools to test plan generators, test oracles, automated user interface testers and formal refinement and theorem provers. Configuration management and version control tools help manage large repositories of software artefacts. Many tools, particularly software design and configuration management tools, provide varying degrees of change management and tracking. Groupware support, often integrated in many of these tools, facilitates team communication and co-ordination.

This article provides an overview of current practice in software tool features and usage. It also surveys recent research work in the area of software tools, indicating recent and likely near-future trends in tools development. We hope this approach will be of benefit to both software engineering practioners, informing them of likely directions in software tool development, and to software engineering researchers, providing a framework in which to place their own tools-related work. The following section outlines recent directions in software process evolution, product domains and people management. As different software tools are used throughout the software process for different activities, changes in processes, products and project organisation have introduced a variety of new demands on tools and tool developers. A categorisation of software tools aligned with software process activities is then presented, including examples of tools in the various categories, along with a brief explanation of common terms used to describe software tools. This is not a comprehensive review of tools and tool usage, but intended to demonstrate the breadth of the software tools area and the deployment of different kinds of tools throughout the software lifecycle. We describe key elements of the structure of software tools, ranging from repository and data management to multiple view presentation and editing. We then focus on the key issue of tool integration, outlining current approaches to integration, recent research trends and likely near-future integration support in software tools. We describe various strategies for supporting collaborative work with different software tools, facilitating improved teamwork with tools. We review and discuss current research, practice and likely trends in automated support in software tools, alleviating developers of tedious tasks and helping manage ever-growing software complexity. We conclude with a summary of some important factors in successful tool development and deployment.

Software Process, Product and People Changes: Tools Impact

The software processes employed by developers has moved from low-iteration and feedback models like waterfall [92] to medium-iteration and feedback like evolutionary and spiral [15] to high-iteration and feedback such as the Unified Software Process (USP) [52], eXtreme Programming (XP) [13], and Rapid Applications Development (RAD) [75]. These high-iteration models also tend to utilise less formal transition procedures from phase to phase, compared to military and safety critical systems which use rigorous review and sign-off to control processes and maintain accountability [92]. Figure 1 illustrates this trend: older software process models focused on completion or near-completion of earlier phases of development, with limited feedback from later phases to earlier phases. The software tools used to support such development thus didn't need to concern themselves overly much with handling feedback - tools could be designed with a specific lifecycle phase focus, integration mechanisms could be simplistic (mainly focused on export of data to tools used in later phases), and collaborative work support, while manifested in some tools, was often delegated to stand-alone, non-software development-specific products.



(a) Waterfall model. (b) Spiral/fountain model.

(c) Iterative evolutionary model. (d) Rapid Applications Development.

Figure 1. Software process evolution.

Newer process models place a grater emphasis on rapidly developing specifications, designs and prototypes, and evolving these over (potentially) many rapid iterations. Parts of this work are often done by small, tightly

coupled, possibly distributed groups of developers with integration of parts done periodically [12]. Tools used to support such processes need to provide either a much greater range of coverage of supported lifecycle activities, or provide more sophisticated integration mechanisms supporting full bi-directional information exchange for "round-trip" software engineering. The growth in popularity of distributed software development - manifested by outsourcing, asynchronous work, tele-commuting and virtual software teams [36, 12] - also puts much more demands on developers sharing information and working with automated collaborative support. To be truly effective, such collaboration features need to be manifested in the software tools being used [26, 47].

In addition to process changes, the products being developed have steadily grown more complex and diverse. These include the great increase in developing multi-tier distributed systems for E-commerce applications in the '90s, the advent of ubiquitous or pervasive computing systems (wireless devices, heavy use of embedded systems, novel user interface technologies), and the increased demands for end-user computing facilities in software (typified by enhanced user configuration facilities) [53, 70, 74]. The proliferation of new technologies and development methods, including middleware technologies and component-based development, and the much larger systems typically under development, increase the demands on software tools. Developers need improved, high-level support to more effectively specify, design, generate, test and deploy complex distributed systems and legacy systems integration technologies. They need support for storing, retrieving and describing software components and shared interface definitions. Sophisticated domain-specific user interface technologies must be designed, implemented and tested appropriately. End users may even need quite complex facilities to extend and enhance their applications long after deployment. Managing large system software development artefacts is crucial: development teams need support to version, configuration manage, document and reuse vast numbers of artefacts with complex inter-dependencies [68].

Developers themselves have become organised in different ways. This includes the great increase in tele-commuting by software developers, the use of out-sourcing of parts of software projects, the use of packaged, tailored software solutions; and the rise of virtual software teams, often from different cultures and working in disparate locations. This adds to the complexity of managing software development and puts new demands on software tools for all phases of development. In addition, the continuing shortage of skilled software engineers means many people developing software are not adequately trained for the tasks they are performing [38]. While software tools can not disguise or ameliorate inadequate developer skill levels, good tools can nevertheless assist developers with limited skill sets in developing systems with tolerable quality. An additional force is the need to roll out software ever more quickly, particularly in the Dot Com start-up area where generation of cashflow is a strong imperative and so the product must be very quickly conceived, designed, developed and deployed [2]. This leads to developers being in situations where there isn't time for detailed design and rigorous testing. Development tools used need to cater for and partially redress these time-to-market pressures.

Software Tool Features and Usage

There have been tools developed for virtually every phase of the software development lifecycle [42, 51]. For many phases, a large number of tools exist that perform the same or similar functions. Some tools provide facilities that span many different phases. Others are focused on a particular kind of software development task, technology, language or problem. Software tools have been characterised by a wide variety of descriptive terms. Unfortunately many are historical and/or loosely defined in meaning. When reading research papers, trade journal articles or commercial tool vendor white papers, readers may encounter a wide range of terminology, some terms meaning more or less the same thing, others intended to distinguish tools with quite different purpose and features [51]. In the discussion of tool usage and facilities below, we associate a set of widely accepted tool facilities with various descriptive tool categorisations, but don't claim our usage of these terms is in any way definitive.

In Figure 2 we illustrate commonly identified stages of the software lifecycle and associate with these categories of tools commonly deployed during these phases of development. Some kinds of software tools are generally quite limited in scope e.g. testing tools, programming tools and user interface development tools. Some kinds of tools provide support for many phases of development, such as RAD tools, application generators, CASE tools, and 4GLs. Some tools are intended for use throughout the software lifecycle. These include process/project management tools, process-centred environments (PCSEEs) and collaborative work tools. However, these typically address limited activities in each phase and are used in conjunction with more specialised tools.

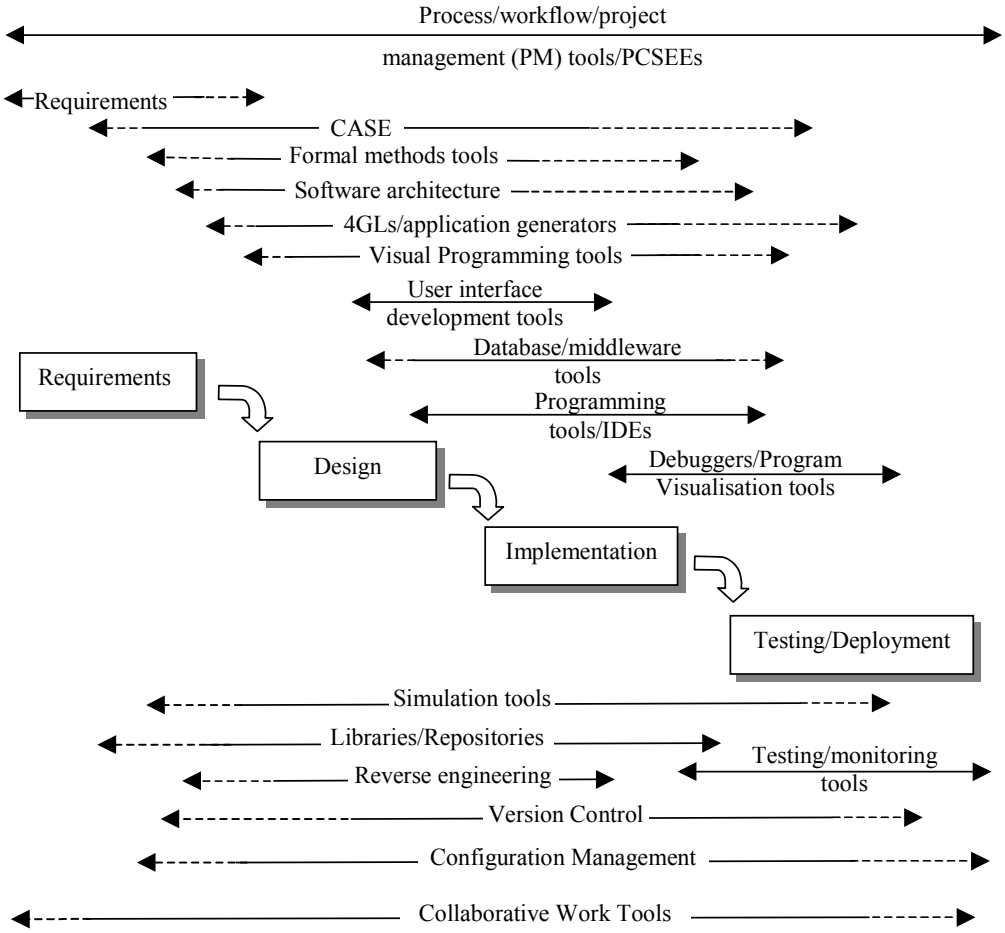


Figure 2. Tool usage throughout the software lifecycle.

Process support tools, such as Process-Centred Software Engineering Environments (PCSEEs) and Workflow Management Systems (WFMS), are used throughout software development to support modelling, enacting and improving software processes [7]. These tools allow developers to characterise the processes they use, "run" these processes, and co-ordinate software development using the processes. Process-centred environments are typically focuses on software development and often tight integration and co-ordination of tools is achieved. Workflow management systems are typically more general-purpose process tools used to co-ordinate a wide range of work processes, software engineering being but one example [8]. Examples of process-support tools offering novel or ground-breaking features include ProcessWeaver [34], SPADE [7], Oz [9] and MILANO [4]. Commonly used commercial tools include Platinum BPWin™ and Process Continuum™[1] and Rational Clear Guide™[2]. Closely related, and sometimes integrated with, PCEs and WFMSs, are Project Management (PM) and cost estimation tools. Again, there are general-purpose PM tools that can be used to manage software teams as well as a wide range of other projects, and there are also many special-purpose software PM tools. Examples of commonly-used or novel project management and cost estimation tools include MS Project™[3], Rational Clear Guide™, Platinum Process Continuum™, Foresight™[4], and KASCET [50].

Requirements engineering tools support the elicitation, codification, validation and evolution of user and system requirements. As more attention has recently been paid to Requirements Engineering as a complex, crucial phase of software development, a greater range of tools for RE have emerged in recent years. Examples range from those adopting informal, natural language descriptions, such as IDIOM [23], Rational Requisite Pro™[5] and ASPIN [24], to those employing formal requirements specification techniques, such as RATS [29], PARSE-DAT [63], and [106]. Most requirements engineering tools address both functional and non-functional system requirements capture and analysis. In contrast, most analysis and specification tools provide developers with

[1] www.platinum.com
[2] www.rational.com
[3] www.microsoft.com
[4] www.pricesystems.com
[5] www.rational.com

mainly functional system specification techniques. These include object-based techniques like Object-oriented Analysis (OOA) and Design (OOD), function-oriented techniques, like Dataflow Diagrams (DFDs), Message Sequence Charts (MSCs) and State-Transition Diagrams (STDs), and data-oriented techniques, like Entity-Relationship Diagrams (ERDs) [95]. Tools for these analysis and design methods should provide comprehensive diagramming and model analysis facilities, among others. Such tools have existed in various forms for many years, many of these tools falling under the ubiquitous-seeming label of "Computer-Aided Software Engineering" (CASE) tools. Examples of representative commercial CASE tools include Rational Rose™, Platinum ERWin™, SimplyObjects™[6] and Software thru Pictures™. The diagramming tool Microsoft Visio™, while not a CASE tool in the sense that it lacks domain-specific diagramming and analysis features, is nevertheless one of the most commonly used design diagramming tools. Examples of ground-breaking research prototype CASE tools include the Interactive Design Environment [103], RPDE^3 [77], Garden [84], SPE [44] and MOOSE [37]. CASE tools often provide many additional development capabilities, including design-level modelling, code generation, documentation and reverse engineering support. A key trend in recent times has been the standardisation of CASE tool modelling methods due to the rise in popularity of the Unified Modelling Language (UML) [16, 51, 67]. Previously, CASE tools provided a variety of disparate analysis and design notations and methods, but most CASE tools now aim to support a large subset of the UML. As the UML is partially defined in terms of a standardised meta-model, this has also led to standardised exchange formats being developed using UML model encodings [67, 90].

Software architecting and design tools are used to model, analyse and refine implementation decisions. Most CASE tools provide such facilities, usually integrated with specification-level features. A variety of specialised software architecture specification and analysis tools have been developed in recent years, owing to the growth in complexity of most system architectures. Examples of specialised architecture and design tools include Clockworks [43], PARSE-DAT [63], and Dali [56]. Many design tools support code generation features, used to generate implementation skeleton code. Many also provide reverse engineering facilities to acquire design-level information from code bases. Many specialised reverse engineering tools also exist. Some work on source code and regenerate design diagrams, while others work on binary code, regenerating source code. Examples of such tools include Rational Rose™[7], Refine/C and SNIFF+ [10], and UQBT [21].

Application builders typically provide a mix of design and implementation facilities, usually tightly integrated in one tool. These are often called 4GL tools (4th Generation Languages) or Application Generators. Many of these tools are focused on design facilities closely aligned to particular implementation languages, technologies and architectures. Many examples exist, and commonly used commercial tools include MS Access™[8] and Sybase Power Designer™[9] (database table, form and report building), Sybase Power++™, Borland Delphi™[10] (desktop and database client application generation), and Sun Forte[11] and Sterling Cool:Gen™[12] (web-based distributed system design and implementation). User interface design tools, including User Interface Management Systems (UIMSs), provide graphical design environments implementing complex graphical user interfaces and structured reports. Examples include Garnet [72], Amulet [73], Sybase Power Builder and Borland Delphi™. Such facilities are very often integrated with programming environments, 4GLs and application generators. Unfortunately many application generators and programming tools still lack adequate integration with requirements engineering and CASE tools While techniques such as code generation and reverse-engineering provide partial solutions, these result in information loss and disjoint development phases [45].

Programming tools and environments focus on implementation-level facilities, usually restricted to a particular language and sometimes incorporating user interface designers, documentation generators, code browsers and symbolic debuggers. While a great many assemblers, compilers, program editors and debuggers have been developed and deployed in isolation, for many years the trend has been for these facilities to be incorporated into Integrated Development Environments (IDEs). A great many examples exist, providing tightly integrated application generator, editor, compiler, debugger and deployment tool facilities. Many now incorporate version control and collaborative work facilities. Commonly-used tools include Visual Age™[13], Visual Basic™ and Visual C++™[14], JBuilder™ and Delphi™[15]. Each provides limited forms of integration with CASE and testing

[6] www.adpative-arts.com
[7] www.rational.com
[8] www.microsoft.com
[9] www.sybase.com
[10] www.inprise.com
[11] www.javasoft.com
[12] www.sterling.com
[13] www.software.ibm.com
[14] www.microsoft.com
[15] www.inprise.com

tools, though more recently quite tight integration with version control and configuration management tools. Many research systems introducing facilities now commonly taken for granted in IPEs have been developed, including the Cornel Program Synthesizer [88], PECAN [83], RPDE^3 [77] and FIELD [85].

Visual language systems offer an alternative approach to programming and program visualisation, emphasising the use of graphical program representations. The idea is to exploit 2- and 3-dimensional representations to assist developers better understand and construct programs. Typically such systems include tightly integrated visual editors and debuggers and some include design-level system modelling features. Successful commercial examples include LabVIEW [57] and Prograph [41]. Some examples of ground-breaking visual language environments include Escalante [66], Vampire [65], and Forms/3 [19]. In recent years the use of the UML visual notation has had a major impact on visual language development, with many tools using parts of the UML notation in their visual formalisms.

Debugging, testing, monitoring and other evaluation tools are used to assess the quality of software, or to assist developers in identifying and correcting errors in software. Debugging tools are typically focused on low-level source code analysis, usually providing step-through, breakpoint, variable monitoring and simple data structure visualisation support. Almost all integrated programming environments typically provide a debugger of this nature. Algorithm animation and software visualisation tools provide higher-level access to software algorithms and data structures through (typically) visual representations of code structure, call graphs, stylised algorithm representations and so on. Examples include Balsa [18], Tango [96] and Mocha [5]. A wide variety of testing tools have been developed. These include tools to generate and run unit and integration test cases (e.g. Rational Team Test™); test oracles that interpret test run results (e.g. [79]); formal program refinement tools (e.g. [20]); and large distributed systems monitoring tools. The article on Testing Tools further explains and illustrates this important area of software tools.

Version control and configuration management tools provide support for single developers or teams of developers to manage their software artefacts as they evolve. Versioning tools help to manage the creation, organisation and use of multiple versions of software artefacts (traditionally source code, but may also include compiled units, designs and specifications, and documentation). Such tools often adopt a check-in/check-out style of operation, allowing multiple developers to share multiple versions of artefacts in a controlled way. Examples of such tools include the older Source Code Control System (SCCS) and Revision Control System (RCS) Unix-based tools, to the modern Microsoft Visual SourceSafe™ and Voodoo [89] document management systems. Configuration management tools, many incorporating versioning features, provide support for building systems from selected versions of each software artefact. In large systems, the number of permutations becomes enormous, and careful management of system configuration is essential to keep the process tractable. Examples include Rational ClearCASE™[16], Platinum CCC/Harvest™[17] and Continuous Change Manager Suite™[18]. A related area is change control management, which is manifested in tools used throughout the software engineering lifecycle. Most requirements engineering and CASE tools provide limited support for tracking changes made in one part of a system and their likely impact on others parts of a system [35]. Similarly, application generators and programming environments typically provide make-like facilities to ensure correct source and code file dependencies are maintained. A few dedicated change management tools have been developed to assist with dependency management and change impact analysis, including Continuous Change Manager™ and PLEIADES [98]. Many development tools now incorporate dedicated versioning and configuration management facilities, e.g. IBM Visual Age™ and Borland Delphi™. Others, like Rational Rose™, assume the use of a related configuration management tool like ClearCASE™. The article on Configuration Management tools further explains and illustrates this important and still rapidly growing area of software tools.

A wide variety of tools have been developed that use "formal methods", or rigorous mathematically based specification techniques to aid in software engineering. These include tools used to facilitate formal specification editing and basic syntax/semantic checking, such as UQ* [1]; tools to perform formal proof reasoning on specifications, such as ARC [78]; tools to facilitate program refinement [20], and formal methods-based testing tools [91]. In recent years formal methods tools have also been developed for areas such as requirements engineering, visual language specification and construction, and user interface construction and evaluation.

Documentation tools offer support for building user documentation from system specifications and code. Typically CASE tools provide system documentation support (in the form of analysis and design specifications),

[16] www.rational.com
[17] www.platinum.com
[18] www.continuous.com

and while some offer limited user documentation support, specialised tools typically provide more targeted features. Additional system documentation tools have also been developed, including Javadoc, MS Help™ compiler[19], and literate programming tools [93]. Many software tools are used to gather and manage metrics about the software process and artefacts under development. Typically most project management tools, CASE tools and testing tools provide such support. In other tools more metrics recording and analysis facilities have become evident, including requirements capture tools, some programming environments and documentation tools.

Collaborative work tools provide various features enabling a group of developers to work together as a distributed team (in terms of time and/or space). Many software tools come with (usually limited) collaborative work support [37, 46], but many specialised tools have been developed to facilitate group work. Many of these deployed on software projects aren't specialised to software work. Examples include Lotus Notes™[20], MS Outlook™ and MS NetMeeting™, BSCW [11], and ConversationBuilder [55].

It should be noted that some software tool descriptions encompass many of the above categories. For example, the term "CASE" is usually applied to analysis, design and round-trip engineering tools, but is sometimes used to refer to all of the integrated software tools deployed on a project. Software Engineering Environments (SEEs), Integrated Development Environments (IDEs) and Integrated Project Support Environments (IPSEs) are terms, roughly inter-changeable, that refer to a tool or tightly integrated tool set that supports a wide range of software engineering tasks. Integrated Programming Environments (IPEs) and Application Builders usually refer to tightly integrated tool sets that focus on implementation, with limited design, documentation and testing facilities. However, all of these labels given to tools are not always applied uniformly by vendors and tools researchers. For example, in recent years the term "IDE" has been used to describe programming-oriented integrated environments like Delphi, Visual Basic/C++ and Visual Age, when these tools actually have little design, documentation or testing support built into them.

As the construction of software tools is itself usually a very complex software engineering task, many systems have been developed that specifically support the engineering of software tools. These applications are usually referred to as meta-CASE tools, tool builders or tool generators [42]. These meta-tools typically support the definition of tool data structures and views on tool data structures, provide input/output specification techniques, and generate some or all of the code to implement the specified tool [30, 59, 37]. Many meta-CASE tools have been developed, but there are a number of testing tool generators, programming environment/tool generators and some very general-purpose application generators applied to building software tools. Examples of tool-building systems include the Synthesizer Generator [88], Dora [82], KOGGE [30], JComposer [47] and Meta-MOOSE [37]. Computer-Aided Method Engineering (CAME) tools typically provide facilities similar to Meta-CASE tools, but typically focus on supporting the definition and co-ordination of use of parts of software development methods. These often combine software process definition and software artifact repository management facilities with support for specifying analysis, design and testing tools. Examples of CAME tools include Decamerone [49] and MetaEDIT+ [59].

Software Tool Structure

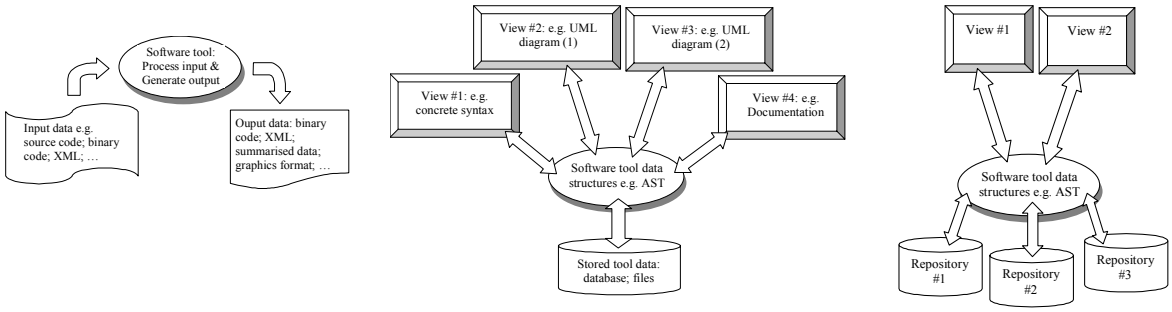
Some software tools are relatively simple programs or suites of programs that read input software artefact definitions (e.g. source code, design encodings or test plans), apply various operations to this input data, and generate an appropriate output structure (e.g. compiled code, documentation or test result analysis). Typically these follow the architecture sketched in Figure 3 (a). When building such tools, tool developers typically have to parse the input file format(s) to extract the data structures the tool uses. Input formats may range from well-structured formats such as XML, test data, database table records and various custom delimited text or binary formats, to formats requiring extensive parsing, such as program source, make-style build and configuration information, and even English requirements (textual typically, though occasionally graphical figures too) [3, 69]. Output generation is typically a more straightforward process, though some work has been done defining "unparser" generators to assist with generating "concrete" representations of software artefact data models [64].

Other software tools are very complex pieces of software providing sophisticated information management which is accessed through multiple views, typically a mixture of graphical and textual renderings of their data structures. Such tools usually have one or more "repositories" that represent the various data structures embodied in the tool, with "views" providing visualisations of data, some of which are editable by developers [83, 82, 47,

[19] www.microsoft.com
[20] www.software.ibm.com

81]. Figure 3 (b) illustrates the structures of this style of tool - a repository data structure (managed in memory or in a database) has either a single or multiple projections (views), which are rendered in text and graphics, some of which provide users with editing capabilities. Changes made to views are applied to the repository, and the repository broadcasts changes to all necessary views, which redisplay themselves as appropriate. CASE tools, 4GL environments and integrated programming environments typically adopt such a structure: repository contains software model (analysis, design and documentation; GUI and database specifications; program code structure) and multiple views provide various projections of this data (OOA/D editors, dialogue boxes for specifying code generation parameters; GUI and table designers; source code editors, class structure browsers and debugger windows). As the need for separation of concerns in software engineering has continued to grow [77, 35], multiple views have been manifested in requirements tools, architecture design tools and process management tools [35, 46]. Some tools adopt "federated", or distributed repositories, as shown in Figure 3 (c) [17, 46]. Such tools may partition data into different data management systems for efficiency or ease-of-construction, or may adopt decentralised architectures with replicated data, typically to ensure robustness (quality of service) and performance [46].

Challenges when building such complex tools include the design of the repository, design of the view data structures and rendering, the provision of appropriate editing facilities, and the "synchronisation" of views (or view consistency management). A repository structure must be rich enough to hold sufficient tool information to enable constraint checking, code generation and import/export of tool data to be supported. It also must be made persistent, either incrementally (as it is updated) or periodically (i..e. batch save/load of data) [101, 33, 37, 58]. Tool views must provide developers with appropriate notational symbols for their development tasks: process models and project plans, analysis and design diagrams, GUI layout and interaction, source code and data structure contents, test plans and results, configurations and version differences and so on. Editing views must be translated into appropriate repository changes, and editing must be effective and efficient, allowing developers to realise real productivity gains with their software tools [80]. Ideally users of tools should be able to change the rendering and editing mechanisms to suit their needs [84, 47]. View consistency management is, in a general sense, very hard. Most tools adopt simplistic approaches, and some avoid the problem by allowing some repository data to be viewed only in one view (often a dialogue box) [45].



(a) Batch input/output-driven (b) Interactive, multiple views, incremental persistency (c) Federated repositories

Figure 3. Typical structures of software tools.

Various interesting trends have emerged with respect to software tool structure in recent years. In the late '80s and early '90s, many tool researchers and some tool vendors focused on developing complex tool repositories which all software tools were expected to make use of. The Portable Common Tool Environment (PCTE) Object Management System (OMS) was a very popular example of this, becoming a repository of choice for much research tool work [101, 102]. Some commercial tools use relational or object databases for much the same facilities. In recent years, much less evidence of such repository work has been reported in research literature, though some efforts still emphasise the potentially great benefits of such an approach [33, 17, 37, 58]. Most commercial tools use custom sequential or indexed file formats to maintain their data. This is typically for efficiency reasons - despite much research effort OMS-style tool repositories always seemed to greatly decrease tool response times and never proved sufficiently scalable for large project applications [17].

The multiple views provided by many tools have increased in richness, particularly as many commercial CASE, 4GL and integrated programming environment tools have increased in functionality. For example, Rational Rose™ exploits multiple views in nearly all of its diagramming facilities, allowing developers to partition their analysis and design models into many overlapping views using the same design notations [37, 81, 82]. The need to present the same information in various ways and to keep multiple views consistent naturally grows under such a scenario. However, the diversity of interfaces provided to developers has not grown at the same rate.

While a few novel interface techniques have been explored (such as MUD-style interfaces for project management and 3D virtual worlds for structure browsing and test evaluation [28]), most software tools are limited to textual and 2D box-and-line style interfaces. Sophisticated view consistency management techniques have been applied to requirements, design and implementation tools [35, 82, 69]. Most software tools still do not, however, provide very good consistency management support, adopting quite rigid and limited "regenerate all affected information"-style consistency mechanisms.

In the mid-'80s and at various points in the '90s "structure editing", or language-based editing, appeared in various research tools. Most graphical and tabular views in tools are "structure edited" i.e. the user applies constrained direct-manipulation editing operations to the view contents to affect view and repository data structure changes [82]. Structure editing of textual views was popularised by work like the Cornel Program Synthesiser [88], Dora [82] and Mjolner [64]. Many structure editors for documentation, programming and formal specification have been produced over the years [1, 88]. While structure editing has some excellent theoretical advantages (e.g. it can incrementally parse and check the semantics of code, offer accurate code completion to users and ensure no syntactic errors), due to its rigidness it is a classic illustration of an interface with poor usability [105]. Experienced programmers find it too limiting and confining and to our knowledge no recent commercial tools exploit it to anywhere near its full capacity. However, various interesting examples of design-centric and program-centric editors have emerged during the '90s that employ language-sensitive parsing, highlighting, cut-and-paste and hypertext facilities. Many commercial CASE tools and programming environments employ these techniques in their textual editors.

Some software tools have tried web-based interfaces, where (some) phases of software engineering are conducted with repository on a host machine (accessed via a web server) and views on developer machines, accessed via a web browser (using HTML, applets, XML/XSL, CGIs or Active Server Pages). Examples include ColD SPA [62], WebME [100], and [39]. The classical advantages of web-based systems, including highly distributed systems, thin clients and easy-to-upgrade services and interfaces, are typically realised by these tools. However, as many software tools require thick client-based, sophisticated desktop editing facilities and desktop tool integration facilities, it is unclear whether there will be a mass-move to web-based software tools, or whether only limited kinds of web-based software tools will emerge. Some tools work almost as well through web-based interfaces as desktop ones e.g. configuration management tools, project management tools, document repositories and many collaborative work tools. Others, like GUI designers, application generators, programming tools and CASE editors, require highly interactive facilities, data exchange mechanisms and sophisticated user interfaces which are still very hard to realise through current web-based technologies.

Software Tool Integration

When building and deploying software tools, tool developers and tool users are conscious that tools are not used in isolation. It is very rare to find a single, monolithic tool that incorporates every facility a development team is ever likely to want for a development project. In recent years in many areas of Software Engineering there has been a trend to developers choosing the "best of breed" methods, processes and technologies with which to engineer their systems. Software tools are no exception; developers want to use the best tool (for them) for each phase in the software lifecycle of their software process, thus realising maximum productivity and software quality gains from their tool-set [25, 67, 80]. This implies software tools must work with each other - they must exchange data and control events, must provide a uniform, consistent interface for developers, and it should be possible to co-ordinate their usage on a project [104]. Tools that do one or more of these very well are typically referred to as "tightly integrated" tools, while those that do them in limited ways (or not at all) are "loosely integrated" tools.

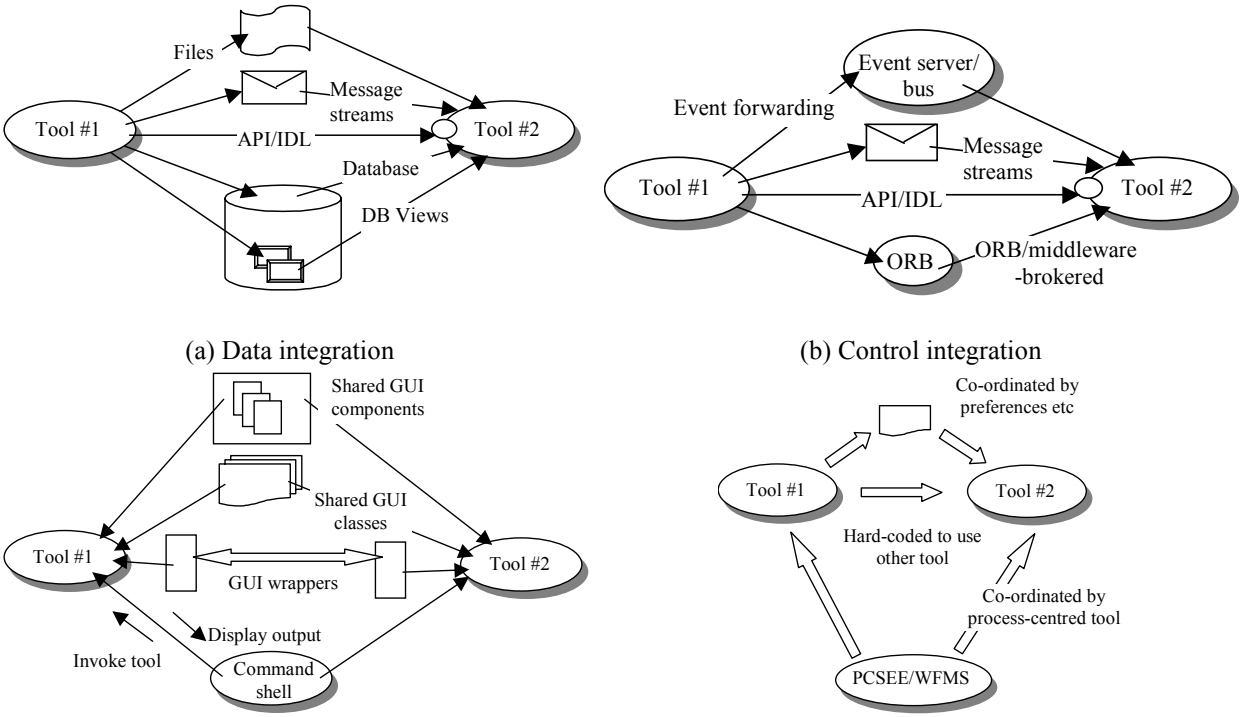


Figure 4. Examples of tool integration strategies.

Figure 4 illustrates common approaches to facilitating software tool integration. Many tools interact by exchanging data through files containing source or binary program code, custom data structure encodings, a common exchange format or XML-style formats [87, 90]. Custom formats have the advantage they can be tailored to particular tool sets, tool data structures, or implementation languages and platforms. They have the disadvantage of limiting tool integration if two 3rd party tools don't agree on the format of the data to be exchanged: translators or adaptors are then needed. "Common" exchange formats, including XML DTDs oriented towards tool integration (e.g. the XMI encoding standard for UML models) offer the potential for much easier 3rd party tool integration. Unfortunately for many kinds of software artefacts that tools may wish to exchange, no common exchange formats have gained universal acceptance. Examples of those proposed, or that are becoming more widely used, include the OMG's UML XMI encoding and the Workflow Management Coalitions workflow exchange format [90, 8], but these only address part of the overall richness of software engineering artefacts about which tools may want to exchange information.

Data integration using a shared database, shared data structures or shared software components offer more complete integration solutions. Sharing the same database/data structures removes the need to import/export data (and build suitable tool components to do this), reduces redundancy and potentially makes multiple view consistency easier [69, 33]. This technique was very popular in the late '80s/early '90s but recent usage has declined. Unfortunately agreeing on database formats and technologies and on data structure definitions and implementations is usually even harder for tool developers than agreeing on exchange formats. This is because different tool development teams choose databases and data structures to best-suit their tools' needs. Another tool that developers might wish to integrate with their existing toolset may require a much different kind of data structure/database approach for its core services. Document repositories and version control systems are examples of successful use of the shared files/shared database concept to achieve (loose) software tool integration (and limited forms of collaborative work support). The rise of component-based systems engineering [97] offers the possibility of a third strategy – sharing software components or at least accessing tool data via well-defined software component interfaces. Some component-based software tools have been developed that use these techniques [47].

Control integration is achieved in many ways: Field, DEC FUSE and HP Softbench use message-oriented solutions where tools exchange messages (tool events) via a centralised message broker [85]. Algorithm animation systems and debuggers are typically integrated with run-time systems in conceptually similar ways, though many have been hard-coded to operate with only one language/virtual machine [18, 3]. In recent years,

RPC and distributed object approaches have also been used to facilitate control event exchange between tools [32]. These typically require tools to agree on standardised interfaces for parts of software tools, and this, like exchange formats and shared database repository schemas, has so far proved challenging for tool vendors to do. Simplified OLE/COM event exchange has been used by a variety of tools to enable (usually a restricted subset) of tool facilities to be accessed in one tool by another. An excellent example is the way Rational RequisitePro™ accesses and annotates Microsoft Word™ documents to facilitate requirements capture and tracking[21]. As with data exchange, as more tools adopt component-based architectures and technologies, it is likely much more control (and data) integration will be facilitated via this approach. A challenge, as with common exchange format definition, remains to formulate common component interface standards to enable a wide range of software tools to be easily and effectively integrated by developers through visual composition tools or scripting languages, without requiring re-engineering and while still remaining efficient for their primary task.

Presentation integration has typically been achieved by the use of a common toolkit e.g. tcl/tk, MFC or Java JFC. However, it is still possible for 3rd party tools to use these (very general-purpose) GUI libraries in vastly different (and inconsistent) ways. Language-based environments, like Smalltalk, LISP and most Prolog programming environments, often provide many tools implemented using the language and its environment facilities, leading to close tool presentation (and often data) integration [40]. Field uses GUI wrappers over message-and-file-integrated tools to provide a seemingly integrated tool user interface [85]. Many large tool sets, like those of Rational, use common libraries of user interface components which give tools from the same suite a common look-and-feel. Some developments in component-based systems and end user computing tools point towards much more flexible tool interfaces, where developers may configure tool interfaces to suit themselves, or multiple tools configure each other dynamically to achieve a consistent presentation to developers [48].

Process integration has become more important as larger teams and tool sets are used on projects, and virtual teams and highly distributed software development has become more common. In the early ‘90s many “process-centred environments” were developed which very tightly co-ordinated software tool usage [7, 9, 46]. Some recent software tools sets have leveraged this work to provide developers with well-co-ordinated groups of tools. Unfortunately, these approaches typically require a lot of knowledge about particular tool facilities and interfaces to be defined (unfortunately, in many cases, in a hard-coded way). General-purpose project management and workflow tools have been used to co-ordinate software tool usage [8], but generally provide a much lesser range of process integration strategies than dedicated process-centred tools. To be effective, process integration often requires a reasonable degree of data, control and presentation integration, and thus is dependent on trends in these other tool integration areas.

Collaborative Work Support

Developers need tool support to assist them co-operating with other developers to engineer software [33, 37, 47]. This support may provide co-ordination technologies, including project and process management support, locking of shared versions or artefacts, floor control, notification and conflict resolution strategies. Communication support typically includes text messaging (email and “chat”) and document annotation, but may also include audio and/or video communications. Composition support typically includes document versioning, version merging and differencing, synchronous (tightly-coupled) editing, and multiple awareness features (multiple cursors and scroll bars, highlighting, fish-eye lens and so on). Figure 5 illustrates some collaborative support features software developers may typically want to make use of.

[21] www.rational.com

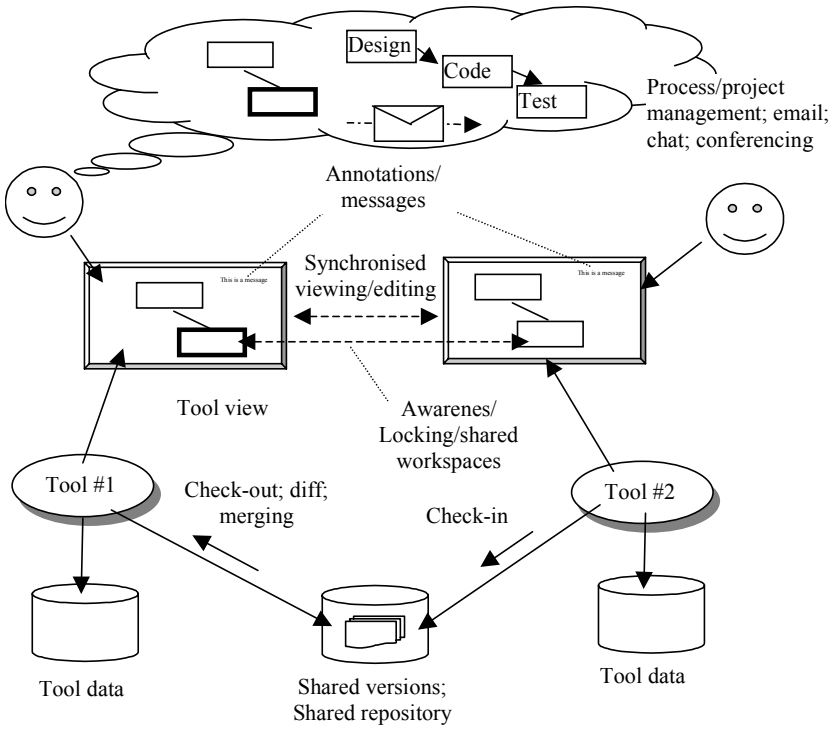


Figure 5. Examples of collaboration support.

Many software tools come with collaborative work support built-in, although often developers use general-purpose tools, particularly for communication (e.g. email and chat tools) and sometimes document management (though software engineering-specific document management tools are often used) [11, 46]. Stand-alone collaborative work tools or components have the advantage that they may be reused in several situations, and often have been tried and tested on many projects (not just software ones). Their main disadvantages are the lack of integration with software tools and that they may lack software engineering-specific features, which may make their features more useful to developers . A trend over the past 10 or more years is to add more facilities to software tools to support collaborative work, or at least to make software tools more open to the addition of or use with collaborative work-supporting tools [26, 37, 47].

The most widely used mechanism for supporting collaborative software engineering with tools is to exchange tool data files. This can be done using shared document repositories (or even just FTP), check in/check out versioning systems, networked file systems and intranets, or even email attachments [11, 46, 76, 68]. Some tools have built-in facilities for data model exchange between multiple instances of the tool being used by different developers. Many tools provide built-in versioning and differencing support [26, 46, 64]. The reason for building in version control is that the versioning, differencing and merging support can make use of the particular software artefacts being developed e.g. present UML model differences graphically in a UML diagram view; make “informed” merge decisions when merging code files, user interface specifications or test plans; and so on [64].

Forms of "shared workspaces" are used by a variety of software tools to facilitate collaborative editing of documents, viewing of test results and monitoring of complex systems. The most common tends to be collaborative editing support, found in a variety of CASE tools and programming editors in particular, but is also found in some process management tools, requirements engineering tools, documentation tools, user interface builders and collaborative code review and testing tools. Some tools support flexible forms of editing allowing developers to freely move between tightly coupled editing and version-based asynchronous editing [27, 46].

Many tools have been augmented with shared annotation and messaging facilities. These allow developers to make "notes" against software artefacts which are viewable by other developers, or to communicate with other developers via email/chat-style messaging. The advantage of adding such facilities to tools rather than using off-the-shelf facilities is the ability to add context information to messages/annotations e.g. what artefact is being discussed, what process stage is enacted/task being actioned by developers etc. [7, 34, 46].

Automated Assistance in Software Tools

As software grows in complexity, software processes become more flexible yet complex, and more developers must co-operate and co-ordinate their work, software tools providing developers editing, reviewing and management facilities are not in themselves sufficient to ensure optimal project productivity. The number of tasks developers must manually perform with their tools, no matter how effective and efficient the tools are, continues to increase. Eventually either overwhelms developers or leads to them not performing (often critical) tasks e.g. they avoid or reduce appropriate project management metrics capture, detailed design analysis and rigorous software testing.

The solution is provision of various forms of task automation in the software tools developers use - the tools carry out perhaps a wide range of activities for the developer at appropriate times and inform the developer of results of actions in appropriate ways [71, 90, 46]. Many automation facilities have been used in tools, and in recent years more and more have tended to be added. Figure 6 illustrates some of these facilities.

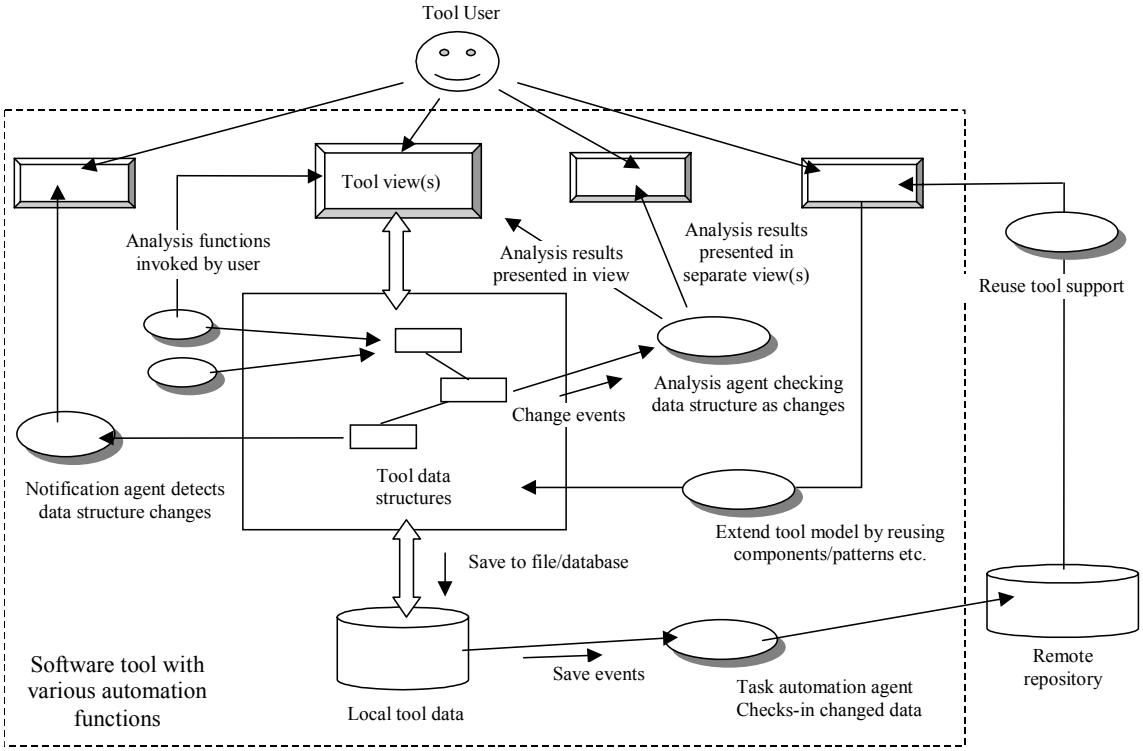


Figure 6. Examples of automation in software tools.

Analysis (or "checking") functions range from classical syntax/semantic checks by compilers to CASE tool model consistency checks to formal program specification reasoning to the application of various metrics/design guidelines to assess software quality. Such checking may occur under developer direction, but in some tools occurs optimistically by the tool. The Borland Delphi 5™ compiler is a good example, performing such tasks incrementally, informing the developer of simple errors unobtrusively as the developer modifies code. Some software tools are built specifically to perform checking of artefacts produced by other tools. Examples include requirements checkers, formal specification verification tools, and test oracles run over test result data [78, 79].

Reusing function or class libraries, remote object interfaces and software components is a challenging task, particularly as the number of artefacts that can be reused continues to grow. Much work has been done developing reuse repositories or library tools/tool facilities to aid developers [68, 71, 76]. Common approaches use type, semantic or execution categorisations to organise the library, and use natural language or formal specification terms to retrieve components from the library. Recent trends to developing "component marketplaces" [68] have emphasised the need for good repository organisation, querying and retrieval facilities. Typically repository management tools and querying/reuse tools can be separated. However, the reuse tools often need to be built into other software tools that actually reuse retrieved components, as they need to modify the data structures of these tools to properly incorporate and configure the reused components. Again, the

general problem of lack of agreed standards for tool interfaces and data structures makes it difficult to produce generalised reuse tools that can be deployed with 3rd party software tools.

A related issue is reuse of collections of software artefacts: class frameworks, design patterns, product-line architectures, requirements templates, analysis models, test plans and data, and process fragments [63, 90, 86]. Due to the changes in software processes, reuse of both individual components and collections of process-to-testing artefacts has become more common, requiring appropriate tool support. A variety of tools have been produced in recent years to support pattern reuse [86], again often needing to be incorporated into design and programming tools to be effective. Many process and project management tools incorporate process template reuse support, as do requirements and testing tools.

The term "software tool agent" has become a popular way to to describe parts of tools, or discrete tools in their own right, that perform –task such as monitoring tool data structure state changes and incrementally performing syntax and semantic analysis checks, and assisting with component reuse [90, 76, 29, 74]. An excellent example are the "design critics" in UML/Argo that "watch" a developer design a system and give continuous (unobtrusive) feedback on various parts of the evolving design. Recently many tools have been extended to include various other kinds of task automation agents e.g. perform automatic check-in/check-out of software artefacts; automatically notify developers when artefacts they are interested in change; automatically start/invoke other tool functionality and so on. The aim is to remove the performing of "tedious" tasks from the developer and have the tool automate or semi-automate them. Many programming and CASE tools incorporate "agents" that automatically regenerate hyperlinks, perform code/design completion, and co-ordinate tool usage [81]. Such automation needs to be done well in a tool i.e. the correct automations applied at appropriate times, otherwise the agents may become a hindrance to the developer.

The concept of "intelligent tools" often tries to address one or more of the preceding areas - the tool has some notion of domain-specific knowledge, or "intelligence" deployed to assist developers to more effectively engineer their software with the tool. Many examples exist, including the notions of organisational memory assisting process and component reuse, intelligent repository organisation, and various agent-based software tool applications [71, 76, 90].

Tool Development, Deployment and Assessment

Many software tools are very complex, distributed, multi-user software applications in their own right, typically with a demanding user community, and engineering such tools is a significant software engineering challenge [42]. While software tools can be built using other software tools, including general-purpose software engineering project management, CASE, programming and testing tools, the complexity of software tools and their particular needs has led to specialised tool construction tools. Such meta-tools assist developers in building and integrating software tools: typically these are called meta-CASE tools, but may also be referred to as tool generators or builders or sometimes CAME tools [42, 49, 59]. These typically give tool developers specialised facilities aimed at specifying and constructing tool repositories and views, developing view editors, and providing multi-user and/or tool integration service abstractions [59, 30]. Many software tools also come with limited extension facilities, such as scripting languages or APIs, which allow them to be extended in limited ways by tool users.

When developing software tools, tool developers typically need to perform a number of tasks. These are usually codified in meta-CASE tools but these tasks can, of course, be performed using any general-purpose software tool set. Key steps include:

- *Repository Definition*. Defining repository data structures, constraints and persistency mechanisms. Some tool builders provide textual or visual specification languages to do this in an abstract way, while others provide reusable classes with tool repository component abstractions [30, 37, 59]. Many tools based their representations on abstract syntax trees or graphs, with annotations for semantics capture [88, 64]. Some tool builders enforce the use of PCTE or similar database repositories, while others adopt custom file/database formats or even XML or similar "general" information encodings [32, 90]. A key challenge for tool developers is to ensure the tool provides users (i.e. software developers) with good data validation constraints and analysis facilities.
- *View Definition*. Defining view notations and editors. Tool developers must specify, for multiple-view supporting tools, the projections of repository data to use [85, 69, 47]. They must also specify the concrete syntactical appearance of tool data: language-oriented concrete textual syntax e.g. C++ or Java code, English documentation, test data representation in tables; box-and-line visual representations e.g. UML, ER and DFD diagrams, program static/dynamic structure visualisations as networks; or 3D renderings of

complex data e.g. debugging traces, performance monitoring information and complex requirements. For most tools they must specify input/output mechanisms: for graphical editors the direct manipulation facilities; for text and table editors the editing, parsing and unparsing approaches; for tools that read and write files, the data parsing (reading) and generation (writing) to be done.

- *Integration Mechanisms*. Tool integration, multi-user support and automation facilities must be specified and implemented. Some meta-CASE systems provide some of these as "inherent" in all of their generated software tools [37, 59]. Others require such facilities to be implemented using general-purpose programming languages [85, 87] or "plugged in" using software components or similar mechanisms [47]. Often these facilities will end up determining the success or otherwise of the tool - in order to be scalable, extensible and usable on multi-person projects, appropriate facilities of these kinds are essential.

When choosing and/or developing and deploying software tools, 3rd party or self-developed, a development team needs to consider various factors. Essentially a variety of selection criteria need to be applied to the candidate tools or tool specifications to determine their ultimate suitability. Not all are appropriate for all teams, team processes and software products. For example, the tools used by a small, tight-knit team to rapidly develop a simple product prototype that will be abandoned after presentation to clients are likely to be vastly different to those tools used by a large, distributed team to fully implement and deploy a large, enterprise-wide software system that must be maintained over a long period. Some key selection criteria include:

- *Development process and tools synergy*. Given the development process and team organisation to be employed, do the tools fit into this process? If rapid applications development-style of process is to be used/being used, tools typically must be very tightly integrated (ideally a single tool being sufficient), allowing rapid iteration from requirements through to testing. More staged iterative development tends to tolerate less well integrated tools and tools with overlapping feature sets. Distributed teams and virtual software teams require tools with good version control, configuration management and project/process management and co-ordination support.
- *Tool feature set appropriate*. The nature of the product under development and likely implementation technologies can greatly influence tool choice. Stand-alone, small software products using standard technologies can make do with very general-purpose tools for most life-cycle phases. Software being produced for enterprise systems requiring complex middleware and data management, or software for embedded systems or making use of highly specialised APIs typically require tools designed for these domains. Testing embedded and real-time software is much more challenging than most other kinds of software. Highly distributed systems with massive replication or partioning of software is very complex to design, implement, deploy and test, usually requiring tools specialised to this domain.
- *Integration and extensibility of tools*. For projects likely to undergo extended change, incorporate new requirements and technologies, or be taken over by different developers, care must be taken to ensure tools used support new tools being deployed by developers or tool enhancements [51, 69]. In addition, as tool vendors may disappear and their tools may in the future no longer be supported, ensuring general data exchange formats are used by a tool can safe-guard against large amounts of tool-specific software artefacts becoming unusable. It is no longer sufficient to archive just source code and make files for software projects - a rich range of artefacts must be retained and be reusable.
- *Tool usability*. Assessing the "quality" of software tools is incredibly difficult using traditional measures. This is because the organisational cost of "proper" tools assessment requires a very large commitment to tool trials and trial metrics analysis. While some organisations can afford this, and need to do such comprehensive tools assessment, many cannot. Various approaches have been developed to assessing software tools, particularly CASE tools, many focusing on the "usability" of the tool interface, or comparison of tool features to required checklists [25, 80]. Typically such approaches involve careful analysis of tool features and feature manifestation, sometimes involving limited tool trialing on representative software development problems or parts of problems. It should be noted that relying on tool vendor publicity is not always a very accurate measure of true tool feature provision and behaviour.

Summary

Software processes, methods, technologies and team management have all undergone continuing change in recent years, mostly with increasing complexity of software products being driving factors. Software tools that provide appropriate facilities to developers can make or break a software project. Tools are used throughout the software lifecycle, some tools being very focused and specialised for particular tasks, methods and technologies, others being quite general-purpose and widely used throughout a software project lifetime. Good choice of software tools for a project is thus essential.

Some key trends over the past 5-10 years have emerged in the software tools research and practice communities. As software has grown more complex, tools often need to provide ways of managing complexity through multiple views on the same software artefacts. This trend to supporting separation of concerns has expressed itself from requirements capture tools to CASE tools to programming support and testing tools. Tools themselves have begun to leverage leading-edge technologies such as web-based interfaces and infrastructures, 3D virtual worlds, high-performance networks and open, extensible component-based designs. Tool integration is still a key issue, with some tool sets adopting tightly integrated but inflexible infrastructures, while others provide looser, more open but less powerful models. A great deal of work has been done investigating standardised interchange formats, shared tool repositories and process-centred environments, but only limited successes with these have so far occurred. Component-based software engineering tools would appear to offer many advantages in support effective tool integration and tailoring, but few such tools have yet gained wide-spread acceptance. The need for collaborative work support, from informal project team communications, version management and shared workspaces, to large project co-ordination strategies, is almost certain to continue to grow. The advent of virtual teams and highly distributed nature of much software development requires effective support for these collaborative activities. The increase in complexity in software product, changes in software team organisation and advent of highly iterative software processes greatly complicates a developer's tasks. Automation in software tools goes some way to alleviating this, by reducing the number of developer-performed "tedious" tasks, enhancing the software analysis facilities available to developers, and allowing more proactive and effective change notification and reuse practices to be performed.

It should be noted that most recent trends in software tools are mutually necessary and generally compatible. The increase in demand for separation of concerns in tools via multiple views is more complicated when multiple developers share the views, requiring improved view consistency, collaboration and co-ordination support. The demand for tools with facilities oriented towards particular development methods and implementation technologies requires enhanced tool integration facilities so more general-purpose tools can be effectively used with highly specialised tools. The need to extend tools with developer-specified automation agents requires the use of sophisticated tool integration facilities like message exchange via standard exchange formats, open tool APIs and component-based tool architectures. The almost certain continued increase in software process, product and people management complexity means the choice, development and use of software tools will continue to be a crucial part of project planning and management. Commercial software tools provide many enhanced facilities over those of only a few years ago. However, there are still many fruitful areas of tools research that can be pursued and results moved through into commercial tools to help enhance practitioner productivity and software product quality.

Acknowledgements

The helpful comments of the anonymous referees in improving an earlier draft of this article are gratefully acknowledged. Support for related software tools research by the authors has been provided by the New Zealand Public Good Science Fund and the University of Auckland Research Committee.

References

1. Allison W., Carrington D., Jones T., Stewart-Zerba L., Welsh J. Visualising software documents in a generic development environment. In *Proceedings of the 1997 Australian Software Engineering Conference*, IEEE CS Press, pp.49-59.
2. Arthur, C. 10 ways to bulletproof your dot com. *Internet Business*, June 2000, *Internet Business Magazine*, pp.36-42.
3. Avrahami, G., Brooks, K.P., and Brown, M.H. A Two-View Approach to Constructing User Interfaces*, ACM Computer Graphics*, **23** (3), 1990, 137-146.
4. Agostini, A., De Michelis, G. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, **9** (3-4), Aug. 2000, Kluwer Academic Publishers, pp.335-63.
5. Baker, J.E., Cruz, I.F., Liotta, G., Tamassia, R. A new model for algorithm animation over the WWW. *ACM Computing Surveys*, **27** (4), Dec. 1995, pp.568-572.
6. Bambury, P. A. 1998. Taxonomy of Internet Commerce, *First Monday,* **3** (10), www.firstmonday.dk.
7. Bandinelli, S., Di Nitto, E., and Fuggetta, A. Supporting cooperation in the SPADE-1 environment, *IEEE Transactions on Software Engineering*, **22** (12), 1996.
8. Barnes, A. and Gray, J. Workflow products as a tool construction technology for process-centred SEEs, In *Proceedings of 2000 Conference on Software - Methods and Tools*, Australia, Nov 2000, IEEE CS Press.
9. Ben-Shaul, I.Z., Heineman, G.T., Popovich, S.S., Skopp, P.D. amd Tong, A.Z., and Valetto, G. Integrating Groupware and Process Technologies in the Oz Environment. In *9th International Software Process Workshop:The Role of Humans in the Process,* Ghezzi, C., IEEE CS Press, Airlie, VA, October 1994, pp. 114-116.

10. Bellay, B, Gall, H. An evaluation of reverse engineering tool capabilities. *Journal of Software Maintenance*, **10** (5), 1998, Wiley, pp.305-31.
11. Bentley, R., Horstmann, T., Sikkel, K., and Trevor, J. Supporting collaborative information sharing with the World-Wide Web: The BSCW Shared Workspace system. In *Proceedings of the 4th International WWW Conference,* Boston, MA, December 1995.
12. Beranek, P.M. The impacts of relational and trust development training on virtual teams: an exploratory investigation. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*. vol.1, 2000, pp.10.
13. Beck, K. Embracing change with extreme programming. *Computer*, **32** (10), Oct. 1999, IEEE CS Press, pp.70-7.
14. Bolin, S. 1998. E-commerce: a market analysis and prognostication, *StandardView,* **6** (3): 97-108.
15. Boehm, B.W. A spiral model of software development and enhancement, *IEEE Computer*, **21** (5), May 1988.
16. Booch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modelling Language User Guide*, Addison-Wesley, 1998.
17. Bounab, M., Godart, C. Tool integration in distributed environments: an experience report in a manufacturing framework. *Journal of Systems Integration*, **8** (1), March 1998, Kluwer Academic Publishers, pp.31-51.
18. Brown, M.H., Hershberger J. Color and sound in algorithm animation. *Computer*, **25** (12), Dec. 1992, pp.52-63. USA.
19. Burnett, M.M. Types and type inference in a visual programming language. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*.IEEE CS Press. 1993, pp.238-43.
20. Carrington, D., Hayes, I., Nickson, R., Watson, G., Welsh, J. A program refinement tool. *Formal Aspects of Computing*, **10** (2), 1998, Springer-Verlag, pp.97-124.
21. Cifuentes, C., Van Emmerik, M., Ramsey, N. The design of a resourceable and retargetable binary translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, IEEE CS Press, 1999, pp.280-291.
22. Cooperstock, J.R., Fels, S.S., Buxton, W., Smith, K.C. Reactive environments. *Communications of the ACM*, **40** (9), Sept. 1997, pp.65-73.
23. Cybulski, J.L., Reed, K. Computer-assisted analysis and refinement of informal software requirements documents. In *Proceedings 1998 Asia Pacific Software Engineering Conference*, 1998, IEEE CS Press, pp.128-35.
24. Cyre, W.R., Thakar, A. Generating validation feedback for automatic interpretation of informal requirements. *Formal Methods in System Design*, 10 (1), Feb 1997, Kluwer Academic Publishers, pp.73-92.
25. Daneva, M. A best-practice based approach to CASE tool selection, In *Proceedings of the 4th IEEE International Software Engineering Standards Symposium*, IEEE CS Press, pp. 100-110.
26. Dewan, P. and Riedl, J. Towards Computer-Supported Concurrent Software Engineering , *IEEE Computer*, **26** (1), January 1993, pp.17-27.
27. Dewan, P. and Choudhary, R. Coupling the User-Interfaces of a Multiuser Program, *ACM Transactions on Computer Human Interaction*, **2** (1), 1995, pp. 1-39.
28. Dossick, S.E. and Kaiser, G.E. Distributed Software Development with CHIME, In *Proceedings of the 1999 ICSE Workshop on Software Engineering over the Internet*, http://sern.cpsc.ucalgary.ca/~maurer/ICSE99WS/ICSE99WS.html.
29. Eberlein, A., Kremer, R. The application of an intelligent requirements engineering tool in an agent-based framework. In *Proceedings of the 2000 Canadian Conference on Electrical and Computer Engineering*, IEEE CS Press. vol.1, pp.220-224.
30. Ebert, J., Suttenbach, R., and Uhe, I., Meta-CASE in practice: A Case for KOGGE, In *Proceedings of the 9th International Conference on Advanced Information Systems Engineering,* LNCS 1250, Springer-Verlag, Barcelona, Spain, 1997, pp. 203-216.
31. Ellis, C.A., Gibbs, S.J., Rein, G.L. Groupware: some issues and experiences. *Communications of the ACM*, **34** (1), Jan. 1991, pp.39-58.
32. Emmerich, W., CORBA and ODBMSs in Viewpoint Development Environment Architectures., In *Proceedings of the 4th International Conference on Object-Oriented Information Systems,* Springer Verlag, 1997, pp. 347-360.
33. Emmerich, W., Arlow, J., Madec, J., and Phoenix., M., Tool Construction for the British Airways SEE with the O2 ODBMS, *Theory and Practice of Object Systems*, **3** (3), 213-231, 1997.
34. Fernström, C. ProcessWEAVER: Adding process support to UNIX. In *2nd International Conference on the Software Process: Continuous Software Process Improvement,* IEEE CS Press, Berlin, Germany, February 1993, pp. 12-26.
35. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B., Inconsistency Handling in Multiperspective Specifications, *IEEE Transactions on Software Engineering*, **2** (8), 569-578, August 1994.
36. Flynn, N.A. Schmoozing at the home office: reflections on telecommuting and flexible work arrangements for IT Professionals. In *Proceedings of the 1999 User Services Conference for University and College Computing Service Organizations (SIGUCCS '99)*. ACM Press, pp.61-66.
37. Furguson, R.I., Parrington, N.F., Dunne, P., Archibald, J.M. and Thompson, J.B. MetaMOOSE – an Object-oriented Framework for the Construction of CASE Tools, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 19-32.
38. Ge, Y. & Sun, J. 2000. E-commerce and computer science education, in *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, ACM Press: 250-255.
39. Goedicke, M., Meyer, T. Web-based tool support for dynamic management of distribution and parallelism in integrating architecture design and performance evaluation. In *Proceedings 1999 International Symposium on Software Engineering for Parallel and Distributed Systems*, IEEE CS Press, pp.156-63.
40. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading MA., 1984.
41. Golin, E.J. Tool review: Prograph 2.0 from TGS systems. *Journal of Visual Languages & Computing*, **2** (2), June 1991, pp.189-194.

42. Gray, .J.P., Liu, A. and Scott, L. Issues in software engineering tool construction, *Information and Software Technology*, **42** (2), Elsevier, 73-77.
43. Graham, T.C.N., Morton, C.A. and Urnes, T. ClockWorks: Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages and Computing*, Academic Press, pp. 175-196, July 1996.
44. Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B., Connecting the pieces, Chapter 11 in *Visual Object-Oriented Programming*. Manning/Prentice-Hall, 1995.
45. Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Inconsistency Management for Multiple-View Software Development Environments, *IEEE Transactions on Software Engineering*, **24** (11), November 1998, IEEE CS Press, pp. 960-981.
46. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, **2** (5), September/October 1998, IEEE CS Press.
47. Grundy, J.C., Hosking, J.G., and Mugridge, W.B. Constructing component-based software engineering environments: issues and experiences*, Information and Software Technology*, **42** (2), January 2000, Elsevier.
48. Grundy, J.C. and Hosking, J.G. Developing Adaptable User Interfaces for Component-based Systems, In *Proceedings of the 2000 Australian Conference on User Interfaces*, IEEE CS Press, 2000.
49. Harmsen, F., Brinkkemper, S., Oei, H. Situational method engineering for information system project approaches. *IFIP Transactions A: Computer Science & Technology*, **A-55**, 1994, pp.169-194.
50. Harrison, A., Thombs, D., Lee, M.P., Whittaker C, Gillies LC, Saddleton D. Progress and plans for KASCET-a knowledge assisted software cost estimation tool. In *Proceedings of the 1999 Conference on Empirical Assessment in Software Engineering*. Keele Univ. Vol.1, 1999.
51. Harrison, W., Ossher, H. and Tarr, P. Software Engineering Tools and Environments: A Roadmap, *The Future of Software Engineering*, Finkelstein, A. Ed., ACM Press, 2000.
52. Henderson-Sellers, B., Collins, G., Graham, I. OPEN and RUP: how do they compare? *Journal of Object-oriented Programming*, **13** (4), July-Aug. 2000, SIGS Publications, pp.35-38.
53. Hoffman, R. Living in nomads land: Managing mobile devices. *Network Computing*, **11** (14), July 2000, CMP Media Inc, pp.61-78.
54. Humphrey, W. Using a defined and measured personal software process, *IEEE Software*, May 1996, 77-88.
55. Kaplan, S.M., Carroll, A.M. Supporting collaborative processes with Conversation Builder. *Computer Communications*, **15** (8), Oct. 1992, pp.489-501.
56. Kazman, R., and Carriere, S.J. Playing Detective: Reconstructing Software Architecture from Available Evidence, *Automated Software Engineering*, **6** (2), April, 1999, 107-138.
57. Keller, J.P. Teaching PID and fuzzy controllers with LabVIEW. *International Journal of Applied Engineering Education*, **16** (3), 2000, pp.202-11.
58. Kelter, U., Monecke, M. and Platz, D. Constructing Distributed SDEs using an Active Repository, in *Proceedings of the 1st International Symposium on Constructing Software Engineering Tools*, Los Angeles, 17-18 May 1999, University of South Australia, Australia, pp. 149-157.
59. Kelly, S., Lyytinen, K., and Rossi, M., Meta Edit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, In *Proceedings of CAiSE'96,* Lecture Notes in Computer Science 1080, Springer-Verlag, Heraklion, Crete, Greece, May 1996, pp. 1-21.
60. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M. and Irwin, J. Aspect-oriented Programming, In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, Finland (June 1997), Springer-Verlag, LNCS 124.
61. Kumar, R.L. Justifying data warehousing investments, *Journal of Database Management*, **11** (3), July-Sept. 2000, Idea Group Publishing, pp.35-6.
62. Lee, J.D., Hickey, A.M., Zhang, D. Santanen, E., Zhou, L.. ColD SPA: a tool for collaborative process model development. In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, IEEE CS Press, vol.1, 2000, pp.10.
63. Liu, A. Dynamic Distributed Software Architecture Design with PARSE-DAT, In *Proceedings of the 1998 Australasian Workshop on Software Architectures*, Melbourne, Australia, Nov 24, Monash University Press.
64. Magnusson, B., Bengtsson, M., Dahlin, L. An Overview of the Mjølner/ORM Environment: Incremental Language and Software Development, In *Proceedings of TOOLS '90,* Paris, France, Prentice-Hall, pp. 635-646.
65. McIntyre, D.W., Design and implementation with Vampire, In *Visual Object-Oriented Programming*, M. Burnett and A. Golberg and T. Lewis Eds, Manning Publications, Greenwich, CT, USA, 1995.
66. McWhirter, J.D. and Nutt, G.J., Escalante: An Environment for the Rapid Construction of Visual Language Applications, In *Proceedings of the 1994 IEEE Symposium on Visual Languages,* IEEE CS Press, 1994.
67. Melewski D. Wherefore and what now, UML? *Application Development Trends*, **6** (12), Dec. 1999, pp.61-68.
68. Meling, R.; Montgomery, E.J.; Sudha Ponnusamy, P.; Wong, E.B.; Mehandjiska, D. Storing and retrieving software components: a component description manager, In *Proceedings of the 2000 Australian Software Engineering Conference*, Canberra, Australia, 28-29 April 2000, pp. 107 -117.
69. Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software, **8** (1), 1991, pp. 49-57.
70. Morch, A. Tailoring tools for system development*, Journal of End User Computing* **10** (2), 1998, pp. 22-29.
71. Motta, E., Fensel, D., Gaspari, M. and Benjamins, R. Specifications of Knowledge Components for Reuse, In *Proceedings of The 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 36-43.
72. Myers, B.A., "Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces," *COMPUTER*, **23** (11), 71-85, 1990.

73. Myers, B.A., "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Transactions on Software Engineering*, **23** (6), 347-365, June 1997.
74. Nwana, H.S., Ndumu, D.T., and Lee, L.C. ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems. In *Proceedings of 3rd Practical Application of Intelligent Agents and Multi-Agents,* 1998.
75. Ondrejik, J.M. CASE tools and rapid applications development. In *Proceedings of the 1997 Summer Computer Simulation Conference Simulation and Modeling Technology for the Twenty-First Century*. 1997, San Diego, CA, USA, pp.475-80.
76. Oussalah, M. and Messaadia, K. An all-reuse methodology for KBS components library, , In *Proceedings of The 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, Germany, June 16-19 1999, KSI Press, pp. 187-191.
77. Ossher, H. and Harrison, W. Support for change in RPDE^3, In *Proceedings of the 4th ACM Symposium on Software Development Environments*, ACM Press.
78. Parashkevov, A. and Yantchev, J.ARC - A Tool for Efficient Refinement and Equivalence Checking for CSP, In *Proceedings of the 1996 IEEE International Conference on Algorithms and Architectures for Parallel Processing*, Singapore, June 11-13, 1996.
79. Peters, D.K., Parnas, D.L. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, **24** (3), March 1998, pp.161-173.
80. Phillips, C., Mehandjiska, D., Griffin, D., Choi, M.D., Page, D. The usability component of a framework for the evaluation of CASE tools, In *Proceedings of the 1998 Conference on Software Engineering: Education and Practice*, IEEE CS Presss, pp. 134-141.
81. Quatrani, T. *Visual Modeling With Rational Rose™ and Uml,* Addison-Wesley, 1998.
82. Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R. Dora - a structure oriented environment generator, *IEE Software Engineering Journal*, **7** (3), 1992, pp. 184-190.
83. Reiss, S.P. PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering, **11** (3), 1985, pp. 276-285.
84. Reiss, S.P. Working in the GARDEN Environment for Conceptual Programming, IEEE Software, **4** (11), 1987, pp. 16-26.
85. Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment, IEEE Software, **7** (7), 1990, pp. 57-66.
86. Reiss, S.P. Working with patterns and code, In *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences*, IEEE CS Press, pp. 243.
87. Reiss SP. The Desert environment. *ACM Transactions on Software Engineering & Methodology*, **8** (4), Oct. 1999, pp.297-342.
88. Reps, T. and Teitelbaum, T. Language Processing in Program Editors, *Computer*, **20** (11), 1987, pp. 29-40.
89. Reichenberger, C. Concepts and techniques for software version control. *Software-Concepts & Tools*, **15** (3), 1994, pp.97-104.
90. Robbins, J. Hilbert, D.M. and Redmiles, D.F. Extending design environments to software architecture design, *Automated Software Engineering*, **5** (3), July 1998, 261-390.
91. Rosenblum, D.S. Formal methods and testing: why the state-of-the art is not the state-of-the practice. *SIGSOFT Software Engineering Notes*, **21** (4), July 1996, pp.64-66.
92. Royce, W.W. Managing the development of large software systems, In *Proceedings of the 9th International Conference on Software Engineering*, IEEE CS Press.
93. Ruys, T.C, Brinksma, E. Experience with literate programming in the modelling and validation of systems. In *Proceedings of the 4th International Conference on Theory and Practice of Software*, Springer-Verlag.1998, pp.393-408.
94. Sinha, A.K. Extending Moore's Law through advances in semiconductor manufacturing equipment. In *Proceedings IEEE 2000 First International Symposium on Quality Electronic Design*, IEEE CS Press, pp.243-4.
95. Sommerville, I. Software Engineering, 5th Edition, Addison-Wesley, 1996.
96. Stasko, J.T. TANGO: a framework and system for algorithm animation. *SIGCHI Bulletin*, **21** (3) Jan. 1990, pp.59-60.
97. Szyperski, C.A. *Component Software: Beyond Object-oriented Programming*, Addison-Wesley, 1997.
98. Tarr, P., Clarke, L.A. Consistency management for complex applications. In *Proceedings of the 1998 International Conference on Software Engineering*. IEEE CS Press, pp.230-239.
99. Taylor, R.N., Belz, F.C., Clarke, L.A., Osterweil, L.J., Selby, R.W., Wileden, J.C., Wolf, A.L. and Young, M. Foundations for the Arcadia Environment, In *Proceedings of the 3rd SIGSOFT Symposium on Software Development Environments*, ACM Press.
100. Tesoriero, R., Zelkowitz, M. A Web-based tool for data analysis and presentation. *IEEE Internet Computing*, **2** (5), Sept.-Oct. 1998, pp.63-9.
101. Thomas I. PCTE interfaces: supporting tools in software-engineering environments. *IEEE Software*, **6** (6), Nov. 1989, pp.15-23.
102. Thompson AK. CASE data integration: the emerging international standards. *ICL Technical Journal*, **8** (1), May 1992, pp.54-66.
103. Wasserman, A.I., Pircher, D.T., Shewmake, D.T., and Kersten, M.L., Developing Interactive Information Systems with the User Software Engineering Methodology, *IEEE Transactions on Software Engineering*, **12** (2), February 1986, pp. 326-345.
104. Wasserman, A. Tool Integration in Software Engineering Environments, in Software Engineering Environments: International Workshop on Environments, Berlin, 1990, Springer-Verlag.
105. Welsh, J., Broom, B., and Kiong, D. A Design Rationale for a Language-based Editor," *Software - Practice and Experience*, **21** (9), 1991, pp. 923-948.

106. Wieringa, R.J., Saake, G. Formal analysis of the Shlaer-Mellor method: towards a toolkit of formal and informal requirements specification techniques. *Requirements Engineering*, **1** (2), 1996, Springer-Verlag, pp.106-131.
107. Zhao, L., Elbaum, S. A survey on quality related activities in open source. *SIGSOFT Software Engineering Notes*, **25** (3), May 2000, ACM Press, pp.54-57.