Constructing Component-based Software Engineering Environments: Issues and Experiences

John Grundy [1], Warwick Mugridge [2], John Hosking [2]

[1] *Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand, Ph: +64-7-838-4452, Fax: +64-7-838-4155, jgrundy@cs.waikato.ac.nz*
[2] *Department of Computer Science, University of Auckland, Private Bag, Auckland, New Zealand, Ph: +64 9 3737599, Fax: +64 9 3737453, {john,rick}@cs.auckland.ac.nz*

Abstract

Developing software engineering tools is a difficult task, and the environments in which these tools are deployed continually evolve as software developers' processes, tools and tool sets evolve. To more effectively develop such evolvable environments, we have been using component-based approaches to build and integrate a range of software development tools, including CASE and workflow tools, file servers and versioning systems, and a variety of reusable software agents. We describe the rationale for a component-based approach to developing such tools, the architecture and support tools we have used, some resultant tools and tool facilities we have developed, and summarise possible future research directions in this area.

Keywords: component-based software architectures, multiple views, consistency management, tool integration, task automation

1. Introduction

Software engineering tools are usually complex applications. Many require multiple view support with appropriate consistency management techniques, most need to support multiple user facilities, and all need to support appropriate degrees of integration with other, often third party, tools. Software developers often want to reuse existing tools as well as enhance these tools or develop new tools as appropriate. Integrating a development team's tool set is often essential in order for the team members to use the different tools effectively on a project. It is a challenging task to provide these capabilities while ensuring that any resulting development environment is effective [3, 29, 26].

Many approaches to developing software tools exist, including the use of class frameworks [12], databases [8, 10], file-based integration [6], message-based integration [34], and canonical representations [25, 26]. We have been using a component-based approach to develop, enhance and integrate software tools [16, 17]. Our component-based software architecture includes abstractions useful for software tool development. Meta-CASE tools assist in designing and generating tools that use this architecture. We have developed a variety of useful software engineering tools using our tool set. Our experiences to date have shown that software tools developed with component-based architectures are generally easier to: enhance and extend, integrate with other tools and deploy than tools developed using other approaches.

In the following section we review a variety of approaches to software tool construction, identifying their respective strengths and weaknesses. We then briefly characterise component-based software engineering tools, and provide an overview of our approach to developing such tools. Following this, we focus on how our approaches provide useful abstractions for building tools with support for multiple views, multiple users, tool integration, and task automation. We conclude by summarising our experiences in building and using component-based software engineering tools and outline possible future directions for research.

2. Related Work

Many approaches exist for building and integrating software engineering tools. In the following discussion we use Meyers' taxonomy to loosely group a variety of tools and their architectures, including file-based, message-passing (both local and distributed), database and canonical representation approaches [29]. We briefly identify the advantages and disadvantages of each for building tools and for supporting Tomas' taxonomy of types of integration (data, control, presentation and process) [39].

Many Unix-based software tools use a file system-based approach for integrating tools into an environment [6]. This allows tools to be very loosely integrated, so that building and adding new tools is straightforward.

However, such tools generally provide limited data, control and presentation integration mechanisms.

The appearance of tight user interface and control integration of file-based tools can be achieved with message-passing approaches, as used by FIELD, HP Softbench and DEC FUSE [6, 21, 34]. These approaches allow existing tools to be wrapped and integrated into an environment very effectively. Data and process integration is generally not as well supported.

Many software tools are built using object-oriented frameworks, and use file or database-based persistency. These are often combined with version control and configuration management tools to support team development. Examples of such tools include VisualAge [23], EiffelCASE [24], PECAN [32], and Smalltalk-80 [12]. Such tools often provide very polished user interfaces and software development facilities, but are notoriously difficult to extend and integrate with third-party tools.

Tools using a database and database views to support data management and data viewing and editing include PCTE-based systems [4], SPADE [3] and EPOS [7]. The database provides a unifying data integration mechanism, but message-passing is often employed to facilitate control integration. Process-centred environments, such as SPADE, EPOS and ProcessWEAVER [11], support process integration by providing software process codification and execution facilities. The control of third-party tools is, however, difficult, as control integration with such tools is often very limited [9].

A variety of tool generation approaches have been developed. These typically produce database-integrated tools, such as KOGGE [10] and that of Backlund et al [2], or tools which use a canonical program representation, such as MultiView [1], Escalante [28] and Vampire [27]. Generated tools typically have good data, control and presentation integration, and can provide good process integration via the use of process-centered environments [3, 26]. However, integrating generated environments and tools produced using different architectures or generated by different systems has proved very difficult, resulting in limited presentation, data and control integration [3, 14, 26]. If collaborative work facilities are needed in such environments, they usually have to be built into the architecture and generator [3, 9, 13].

Some environments are designed to allow the addition of other tools. Examples include TeamWAVE [35], wOrlds [5], Oz [40], and Xanth [22]. The architectures of these environments usually focus on supporting control integration with other tools, and to a lesser extent presentation, data and process integration. Integration is typically limited to tools built with the same architecture, though Xanth and Oz provide quite flexible integration mechanisms for a wide range of tools.

3. Component-based SEEs

The component-based architectural approaches used by some software tools loosely correspond to a combination of message, database and canonical representation approaches. Components communicate via event propagation and/or message invocation, but often use other components to manage distributed data. Examples of environments using this approach include TeamWAVE, COAST [37], CoCoDoc [38], and SPE-Serendipity [14]. The use of software components to model tools and parts of tools typically allows more effective data and control integration than with other approaches. Presentation and process integration are also effectively supported if components are well designed for extension. Bridges between different component architectures and the use of Oz-style enveloping techniques allow component-based architectures to provide very effective third-party tool integration.

Component-based software engineering environments use a set of integrated components, with each component providing a tool or part of a tool used in the enviroment. Many of these tool components are reusable in other environments and possibly in other domains. Components are typically designed with a minimal knowledge of and dependency on other components, to facilitate reuse and deployment by plug-and-play. As a result, tools built using this model are typically highly reusable and externally controllable by other components. If carefully designed, these component-based tools thus tend to be more readily extended and integrated than tools developed using other approaches.

Figure 1 illustrates the concept of component-based software tools. This example environment consists of several components, which implement different software engineering tools and facilities. The editor, code generator and debugger share an abstract syntax tree (AST) and compiled code representation. The workflow tool co-ordinates use of these tools by monitoring events generated by them and by sending them messages. The workflow tool and data representation components use a distributed file server to manage shared data.

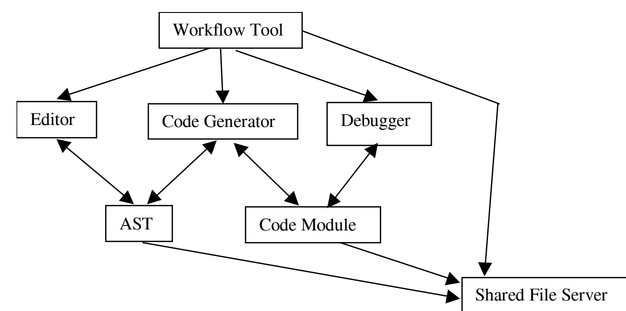


Figure 1. A simple component-based SEE.

Any of these tools or data representation components could be replaced another that satisfies the replaced interface (and required semantics). New tools or components can be added, and may interact with existing components by monitoring events they generate or by sending them messages. Some of these tools could usefully be deployed in other domains. For example, the

workflow tool and shared file server could be used in an Office Automation environment.

When designing and building component-based software engineering tools, a variety of issues must be addressed, including:

- Appropriate identification of components, allocation of data and behaviour to components and design of component interfaces.
- Design of common components for software tool abstractions such as repository management, multiple view, and collaborative work support.
- Appropriate design of components to permit external monitoring, control and user interface extension, thus supporting component reuse, enhancement and integration.
- Provision of appropriate architectures and tools so that tool developers can effectively design, build and deploy component-based tools.
- Provision of appropriate support to tool end users for deploying and integrating tools and enhancing environment behaviour via task automation agents.

Figure 2 shows the Serendipity-II software process modelling and enactment environment, a component-based software engineering tool that we have developed [18]. Serendipity-II illustrates how components can be designed, built and combined to build a sophisticated software engineering environment. Visual process modelling views provide editors and multiple views of process models, using components with extensible user interfaces and events that can be monitored. The graphical representation components are separated from the process model data representation components so that these can be developed independently. Task automation agents are specified visually with components in the model reused by end users. Reusable components are used to implement repository management, multiple user editing and versioning support, event history management, and communication facilities.

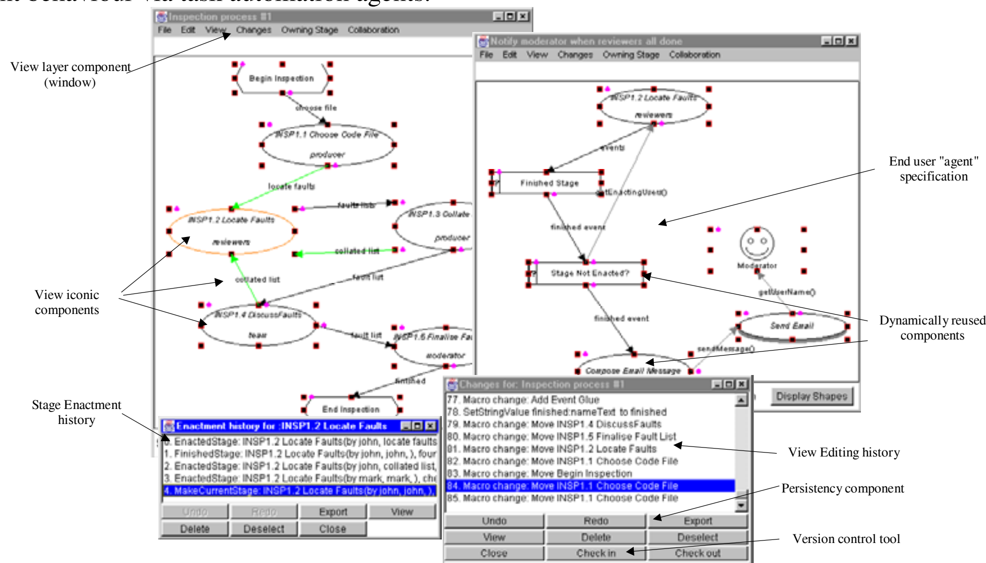


Figure 2. Serendipity-II: an example component-based software engineering tool.

In the following sections we outline our component-based approach to environment development, and demonstrate how that approach can be used to construct tools such as Serendipity-II, and describe our experiences in developing such tools.

4. Overview of Our Approach

JViews is a component-based software architecture and Java class framework which we have developed for implementing complex CASE tools, design environments and Information Sytems [16]. JComposer is a component development environment which generates and reverse-engineers JViews components. This is used in conjunction with the BuildByWire iconic editor design tool [30], and the JVisualise run-time component visualisation and configuration tool [16], to design and implement JViews-based environments.

Jviews provides a set of abstractions for modelling and implementing software components. The JViews framework is implemented using JavaBeans [32]. JViews explicitly supports the design and implementation of applications with multiple, editable views of information, and multiple, distributed users. Many abstractions relating to these capabilities are lacking in other component-based toolkits.

Figure 3 shows the modelling of part of Serendipity-II in JViews. Components represent units of data and functionality that can be statically or dynamically linked to other components via relationships. A flexible event propagation and response model allows components to monitor other components and respond to state change or other events. Events from other components can be acted

on, but can also be modified, vetoed or prevented from reaching other components. Events can be stored to support modification histories, undo/redo and versioning. A variety of persistency, component distribution and collaborative editing and communication facilities are provided by reusable JViews components. These make building new environments much easier, and also allow tools added to an existing environment to find and use components providing such facilities.

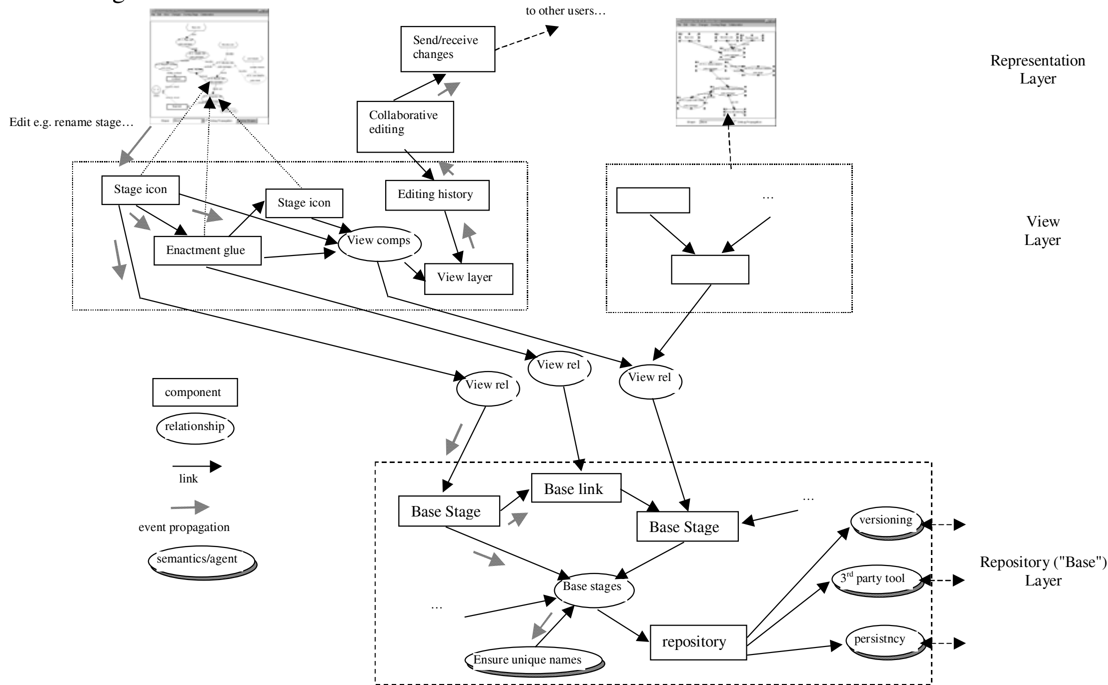


Figure 3. Some JViews components from Serendipity-II.

It is difficult to implement complex, component-based systems without good CASE tool support. We have developed JComposer to provide such support for systems developed with the JViews architecture. JComposer provides multiple views of JViews component specifications. It generates JViews classes to implement component models, and can be used to reverse engineer JViews classes into a high-level, visual architecture description language, similar to that used in Figure 3.

Windows (1) and (2) in Figure 4 show JComposer being used to model various Serendipity-II repository and view components. The component designer can specify information about components in various views, including functional and non-functional requirements, attributes, methods, relationships and events supported by the component, view mappings, and detailed code generation information. A novel event filter and action language allows static and dynamic event handling behaviour for components to be captured at a high level. This language was adapted in Serendipity-II to provide an agent specification and deployment language for end users. Various validation tests can be performed on a component model in JComposer to ensure generated JViews components are correct.

Windows (3) and (4) in Figure 4 show the BuildByWire iconic editor design and generation tool. BuildByWire allows software tool builders to design complex iconic representations and generate JavaBean implementations of these and their editors. JComposer then allows tool developers to link JViews view components to these JavaBean components. The separation of presentation and view data has proved very effective in allowing tool developers to easily reuse iconic forms and replace view component iconic representations.

We have also developed a run-time component visualisation tool, JVisualise, that allows tool users to inspect and modify tool components using JViews component visual representations. Serendipity-II itself has been reused to provide a process modelling and enactment tool for JComposer and other JViews-based tools. Serendipity-II software agent specifications can include component representations from other tools that can be monitored or sent messages, facilitating control and process integration within an environment.

5. Multiple View Support

JViews represents a tool repository's data structure using components. Semantics are embodied in structural components or specified by additional components that monitor structural component events. Multiple views are supported by relationship components that link structural repository components to view components. Events generated due to modifications of the structural components are monitored by the relationship and forwarded to the view components and vice-versa. Thus

when view components that represent repository structures are modified, iconic representations are updated and redisplayed. Editing histories are maintained for views and repository components, using a common reused history component.

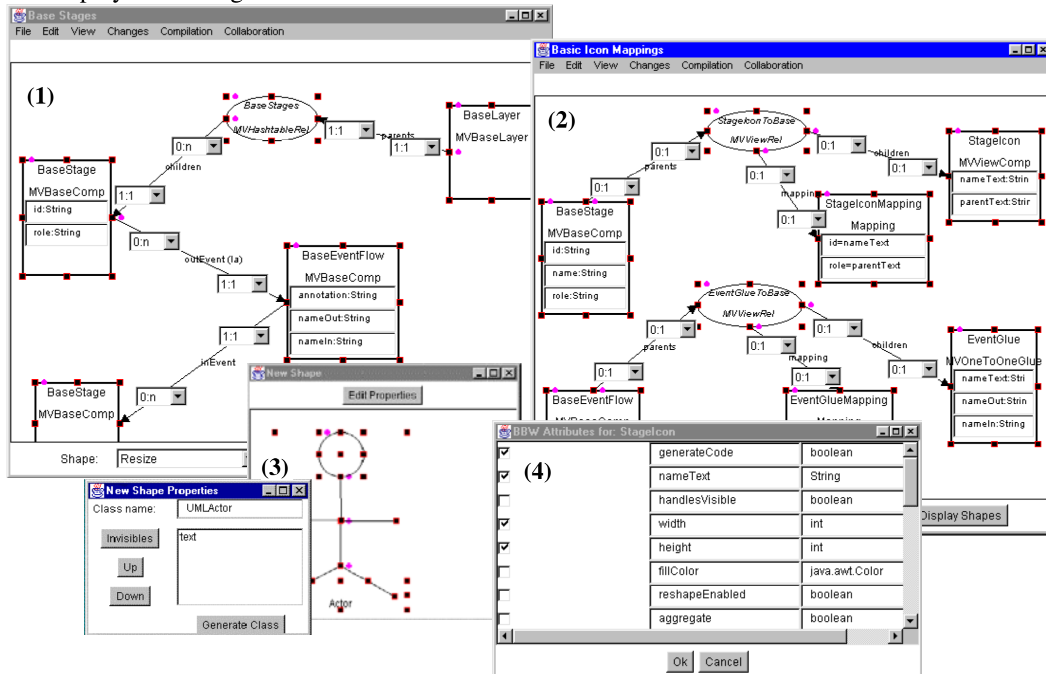
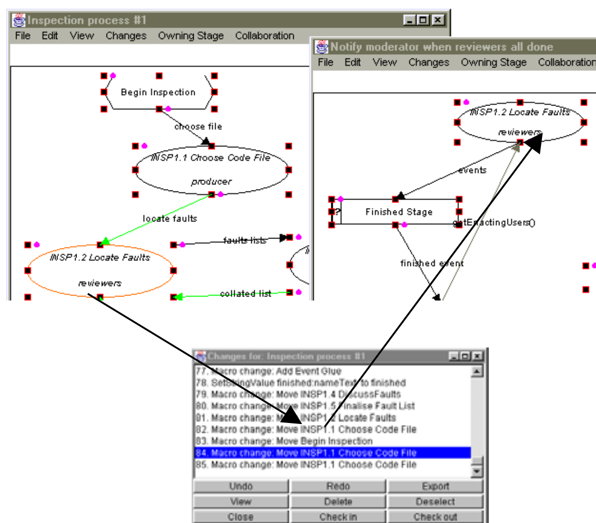


Figure 4. JComposer and BBW metaCASE tools in use.

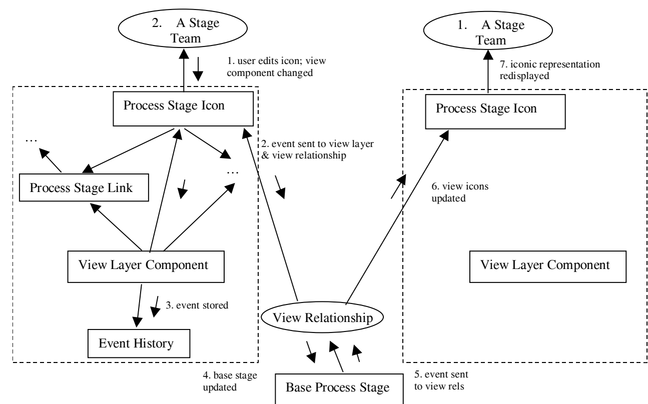
Figure 5 shows how multiple views are modelled for parts of Serendipity-II using JViews. In Fig. 5(a) two process stage views contain reference to the same stage. An editing history for one view is also shown. Fig. 5(b) shows the JViews components corresponding to each view, and the shared repository, together with event flows when one of the process stage icons is modified. The event representing the change is recorded in the view's history component, and is also propagated via the view relationship to the repository component representing the process stage. This causes the repository component to update its state, thus generating events that are propagated to the corresponding process stage icon components in other views. The latter modify their state accordingly to maintain consistency with the initial view.

This is one example of a range of sophisticated inconsistency management facilities provided by JViews view components. These provide tool users with a range of techniques for keeping information consistent and for managing inconsistencies [9].

JViews view components also support extensible user interfaces; other components may modify their interfaces appropriately to support seamless user interface extension [9].



(a) Example of multiple views in Serendipity-II.



(b) JViews architecture supporting multiple views.

Figure 5. Multiple view support in JComposer.

We have found the software component-based approach effective for supporting multiple views. New views and view components can be plugged into an environment without altering existing components. Our JViews components generate event objects that can be stored, undone and redone. Hence undo/redo and version control facilities can be achieved by adding components to manage these stored events. These event history components can be replaced, at run-time, with others, providing different facilities as necessary. Effective inconsistency management in large, multiple view software engineering environments is typically one of the most difficult features to provide. JViews' component based approach provides many useful abstractions that can be combined together simply to produce inconsistency management solutions that are much more difficult to provide with more conventional approaches [9].

6. Collaborative Work and Tool Integration Support

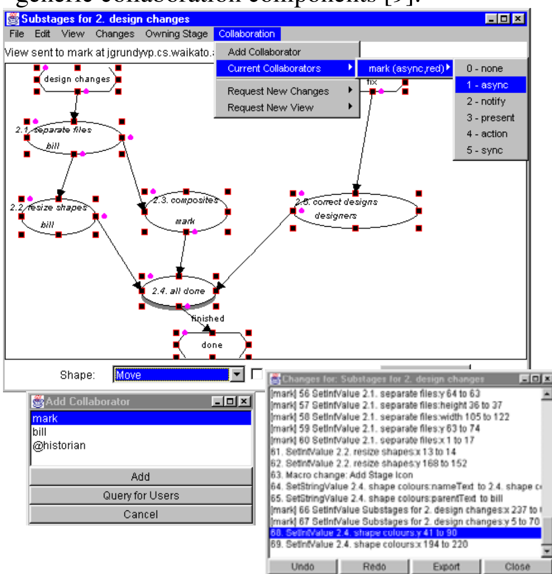
Nearly all multi-user software engineering tools build multiple user support into the tool as it is developed. In contrast, our component-based approach allows collaborative work-supporting facilities to be added as needed to a tool. This may even be done at run-time. JViews components broadcast state change events which can be monitored by other components, both before and after the stage change occurs (the former permitting veto of an operation before it takes effect). Synchronous and asynchronous editing facilities (including locking) can thus be added to existing JViews-based environments through the addition of extra collaboration components monitoring existing event sources and forwarding them to other environments. This can be done with no modification to the existing environment or to the added generic collaboration components [9].

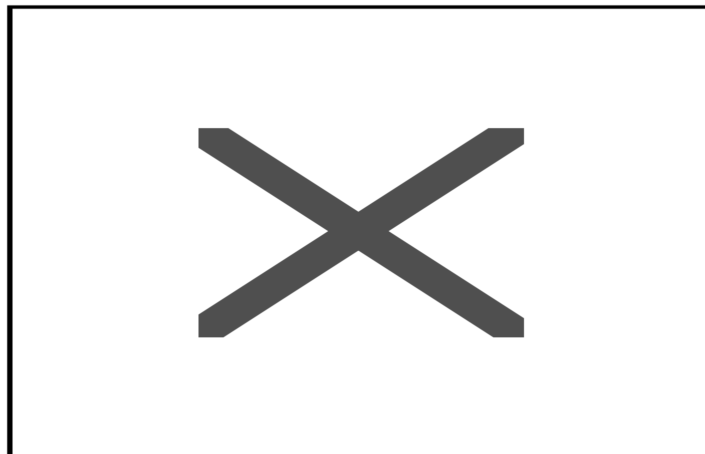
Figure 6 (a) shows a "collaboration" menu in use in Serendipity-II to configure the "level" of collaborative editing with a colleague: asynchronous, synchronous and "presentation" (i.e. show editing changes to others as they occur but don't action them). The "change history" dialogue on the bottom, right hand side shows a history of editing events for the user's process model. Some changes were made by the user ("John"), and others by a collaborator ("Mark").

The illustration in Figure 6 (b) shows how these collaborative editing components were added to Serendipity-II. Such components may be added to any JViews-based environment, with no change to the components or the components that make up the environment.

A "collaboration menu" component is created when the user specifies that they want a view to be collaboratively edited. This component listens to editing changes in the view, and records them in a version record component. If the user is in presentation or synchronous editing mode with another user, the changes are propagated to that user's environment. This is acheieved via a decentralised, point-to-point message exchanging system comprising of a “change sender” component and a “change receiver” component in each user’s environment. A sender propagates view editing changes to each other users’ environment who is interested in such a change i.e. all those who are collaboratively editing the view.

A user’s environment receives view editing changes and passes them to the appropriate view collaborative editing component. This then stores and presents the received change in a dialogue (presentation mode editing) or actions it on the view (synchronous mode editing). In asynchronous editing mode, users request a list of changes made to another user's view and select, via a dialogue, those they wish to have applied to their own version of the view.





(a) Example of asynchronous collaborative editing. (b) Software components supporting collaborative editing.

Figure 6. Collaborative view editing and versioning support

To support the replication of components (via object versioning), JViews has abstractions that are used to maintain copies of collaboratively edited views. When events that describe changes generated in one view are propagated to another user's environment any component references are translated appropriately.

Tool integration is supported in JViews-based environments in a similar manner to multiple views. Components which are part of one tool can request notification of events from components which are part of another tool, or can send these other tool components messages. This facilitates both control and data integration. JViews provides abstractions for identifying and communicating between components that are running in different virtual machine environments or that are resident on different physical machines.

Figure 7 shows a Serendipity-II component monitoring an event history component associated with a JComposer view. When the Serendipity-II component is sent a modification event from the JComposer component, it passes this on to a component that determines if the event is of interest. If so, a third Serendipity-II component notifies the user of Serendipity-II.

Component-based software tools facilitate integration more readily than most other architectures for building tools, as components have well-defined interfaces with event monitoring mechanisms built in. Component-based tools also tend to be engineered for reuse and extension, allowing other components to externally control them and extend their user interfaces as necessary.

The success of this approach can be seen with Serendipity-II. It's component-based construction allows it to be seamlessly "bolted on" to any other JViews-based tool, providing that tool with integrated process modeling and enactment capabilities, without modification to the original tool [18].

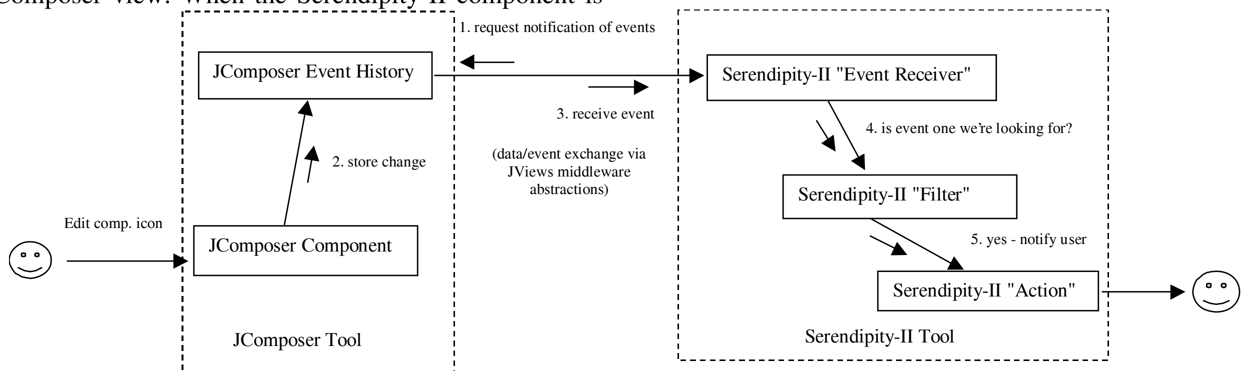


Figure 7. A simple tool integration example.

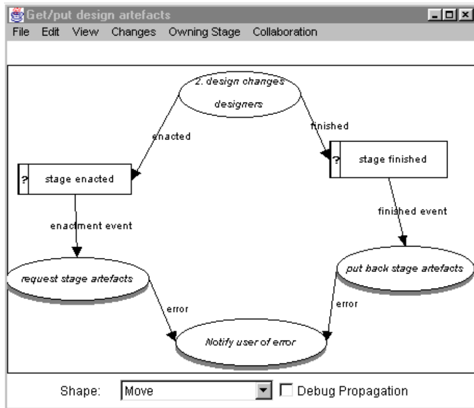
7. Environment Extension and Task Automation

Software engineering environments need to support end user configuration of tools, extension of the environment, and automation of various tasks. For example, the user may wish to be notified of specific changes or to have additional constraints enforced. These capabilities permit the environment to be adapted to changing work processes and tool sets of software developers.

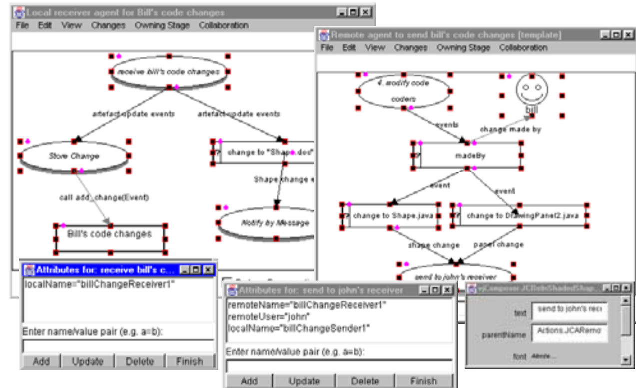
Serendipity-II allows for user enhancement via its visual event filtering and actioning language. This allows users to add, configure and link components into the environment. These event filtering and actioning models can then be deployed as "software agents" which modify an environment's composition and behaviour.

Figure 8 shows two examples of such agents: a simple task automation agent that also illustrates tool integration, and a distributed notification agent. The icons in these models represent Serendipity-II processes (ovals with label and role name), event filters (square icons with question mark on left side), event actions (shaded icons with single label) and JViews components (square icons). Adding these icons to a Serendipity-II agent specification view creates appropriate JViews components that handle events or messages sent to the component. The example on the left instructs Serendipity to download or upload files to/from a shared file server when a particular process stage is enacted (started) or finished.

The example on the right has two parts. The agent on the left runs in user John's environment. The one on the right runs in Bill's environment, and gathers changes made by Bill to the "Shape" and DrawingPanel2" classes while doing the "modify code" process activity. These changes are forwarded by the "send to john's receiver" action to the "receive bill's code changes" action in the agent running in John's environment. John's agent stores the changes in a JViews history component "Bill's code changes" (left branch). It also notifies John by message if any change is made to the Shape class (right branch).



(a) Simple, local software agent for tool integration.



(b) Distributed software agent for notification.

Figure 8. Specifying simple task automation agents.

The component-based architecture of JViews-based environments allows such task automation and tool integration facilities to be straightforwardly built using our event filtering and actioning model. JViews components generate events which can be filtered or used to produce a wide variety of "actions" (notify user, abort operation, communicate with another tool/component, open/close views, store change event, etc.). We have built a variety of reusable JViews components to help facilitate the construction of software agents like those in Figure 7. These include components to: perform parameterised filtering of events; inter-machine event passing and operation invocation; store events; notify users of events and support user communication; and provide interfaces to various third party tools (e.g. MS Word™ and Excel™, Eudora™, Netscape™ and WinEdit™).

8. Future Research Opportunities

Our experiences with component-based software engineering tools have so far been very positive. Environments like JComposer and Serendipity-II were easier to build than comparable earlier systems we developed without using components [15]. This has been due to the high degree of reusability of our JViews components, the useful set of middleware and user interface abstractions embodied by JViews components, and the use of the JComposer and BuildByWire meta-CASE tools. We have been able to extend environments like Serendipity-II using our visual event filtering and actioning facility. This has allowed us to easily develop and deploy new reusable components for tool integration and environment extension, without having to substantially modify existing structures and behaviour.

As more software tools begin to utilise component-based architectures, like JavaBeans [32] and COM [36], and distributed object management facilities like CORBA [31], it becomes easier to effectively integrate such tools with JViews-based environments. Better middleware components to support collaborative work, data persistency, distributed and parallel processing and remote notification will allow better performing software tools to be built. If appropriate component interface designs have been used for such middleware capabilities, upgrading environment infrastructures also becomes easier and more successful.

A problem we encountered with some JViews and BuildByWire components was poor extensibility of their user interfaces. Components need to be designed so that their user interfaces can be appropriately extended by other components, to ensure a consistent look-and-feel and to tailor the user interface to suits the needs of subsets of users.

This becomes more difficult as highly reusable components are developed whose application domains are not fully known during their design, and when third-party component-based tools are reused. Similar issues arise with middleware component interface and capability design, requiring the development of better component-based system interface standards and design techniques.

Multiple view and tool integration support necessarily involves mapping operations and/or events from components in one view/tool into another. Support for complex inter-component event and operation mapping is lacking in most component-based systems, making development of multiple views and tools more difficult. This is an area we have attempted to address in our work, but additional work is needed.

Software developers are tending to require more control over the configuration of their tools, composition of their environments and behaviour of their tools. Appropriate end-user configuration facilities for component-based systems are thus essential to ensure they are effective. Similarly, the use of software agents to automate tasks and the effective cataloguing and retrieval of reusable components remain issues requiring further research.

9. Conclusions

Our experience in the development of complex component-based software engineering tools has convinced us of the value of a component-based approach. The modularity provided by component interfaces and the flexibility provided by the "plug and play" event-based composition of components have proven to be of considerable value in the both the development of such environments, and the provision of end users with the capability to tailor and extend the environments. Several issues remain to be solved to enable software components in general to be used on a large scale, and these also impact on our work on the generation of software engineering tools.

References

1. Altmann, R.A. and Hawke, A.N. and Marlin, C.D., An Integrated Programming Environment Based on Multiple Concurrent Views, Australian Computer Journal 20 (2), May 1988, 65-72.
2. Backlund, B. and Hagsand, O. and Pherson, B., Generation of Visual Language-oriented Design Environments, Journal of Visual Languages and Computing 1 (4), 1990, 333-354.
3. Bandinelli, S. and DiNitto, E. and Fuggetta, A., Supporting cooperation in the SPADE-1 environment, IEEE Transactions on Software Engineering 22 (12), December 1996, 841-865.
4. Bird, B., An Open Systems SEE Query Language, Proceedings of 7th Conference on Software Engineering Environments, Noordwijkerhout, Netherlands, April 5-7 1995, IEEE CS Press.
5. Bogia, D.P. and Kaplan, S.M., Flexibility and Control for Dynamic Workflows in the wOrlds Environment, Proceedings of the Conference on Organisational Computing Systems, Milpitas, CA, November 1995, ACM Press.
6. Champine, M.A. A visual user interface for the HP-UX and Domain operating systems, Hewlett-Packard Journal 42 (1), 1991, 88-99.
7. Conradi, R. Hagaseth, M., Larsen, J., Nguyen, M.N., Munch, B.P., Westby, P.H., Zhu, W. and Jaccheri, M.L., EPOS: Object Oriented Coopeartive Process Modeling, In Software Process Modeling & Technology, A.Finkelstein and J. Kramer and B. Nuseibeh Eds, Research Studies Press, 1994.
8. Daberitz, D. and Kelter, U. Rapid Prototyping of Graphical Editors in an Open SDE, Proceedings of 7th Conference on Software Engineering Environments, Noordwijkerhout, Netherlands, April 5-7 1995, IEEE CS Press, pp. 61-73.
9. Di Nitto, E. and Fuggetta, A. Integrating process technology and CSCW, Proceedings of IV European Workshop on Software Process Technology, Leiden, Netherlands, April 1995, LNCS, Springer-Verlage.
10. Ebert, J. and Suttenbach, R. and Uhe, I. Meta-CASE in practice: A Case for KOGGE, Proceedings of CaiSE*97, Barcelona, Spain, June 10-12 1997, LNCS 1250, Springer-Verlage, pp. 203-216.
11. Fernström, C. ProcessWEAVER: Adding process support to UNIX, 2nd International Conference on the Software Process: Continuous Software Process Improvement, Berlin, Germany, February 1993, IEEE CS Press, pp. 12-26.
12. Goldberg, A. and Robson, D. Smalltalk-80: The Language and its Environment, Addison-Wesley, Reading MA, 1984.
13. Grundy, J.C. Human Interaction Issues for User-configurable Collaborative Editing Systems, Proceedings of APCHI'98, Tokyo, Japan, July 15-17 1998, IEEE CS Press, pp. 145-150.
14. Grundy, J.C. and Hosking, J.G., Serendipity: integrated environment support for process modelling, enactment and work coordination, Automated Software Engineering 5 (1), January 1998.
15. Grundy, J.C. and Hosking, J.G. and Fenwick, S. and Mugridge, W.B. Connecting the pieces, Chapter 11 in Visual Object-Oriented Programming, Burnett, M., Goldberg, A., Lewis, T. Eds, Manning/Prentice-Hall, 1995.
16. Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Static and Dynamic Visualisation of Software Architectures for Component-based Systems, Proceedings of SEKE'98, San Francisco, June 18-20 1998, KSI Press.
17. Grundy, J.C., Hosking, J.G. and Mugridge, W.B., Coordinating distributed software development projects with integrated process modelling and enactment environments, Proceedings of 7th IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Palo Alto, June 17-19 1998, IEEE CS Press.
18. Grundy, J.C., Apperley, M.D., Mugridge, W.B. and Hosking, J.G. An architecture for decentralized process modelling, IEEE Internet Computing, September/October 1998.
19. 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, 960-681.
20. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D. Tool integration, collaboration and user interaction issues in component-based software architectures, Proceedings of TOOLS Pacific'98, Melbourne, Australia, Nov 24-26 1998, IEEE CS Press, pp. 289-302.
21. Hart, R.O. and Lupton, G., DECFUSE: Building a graphical software development environment from Unix tools, Digital Tech Journal 7 (2), 1995, 5-19.
22. Kaiser, G.E. and Dossick, S. Workgroup middleware for distributed projects, IEEE WETICE'98, Stanford, June 17-19 1998, IEEE CS Press, pp. 63-68.
23. IBM Corportation, IBM Visual Age for Java™, http://www.software.ibm.com/ad/vajava, 1997.
24. Interactive Software Engineering Inc., Eiffel CASE™, http://www.eiffel.com/products/case.html, 1998.
25. Magnusson, B. and Asklund, U. and Minör, S. Fine-grained Revision Control for Collaborative Software Development, Proceedings of the1993 ACM SIGSOFT Conference on Foundations of Software Engineering, Los Angeles, 1993, pp. 7-10.
26. Marlin, C. and Peuschel, B. and McCarthy, M. and Harvey, J., MultiView-Merlin: An Experiment in Tool Integration, Proceedings of the 6th Conference on Software Engineering Environments, IEEE CS Press, 1993.
27. 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.

28. McWhirter, J.D. and Nutt, G.J., Escalante: An Environment for the Rapid Construction of Visual Language Applications, Proceedings of the 1994 IEEE Symposium on Visual Languages, IEEE CS Press, 1994.
29. Meyers, S. Difficulties in Integrating Multiview Editing Environments, IEEE Software 8 (1), January 1991, 49-57.
30. Mugridge, W.B., Hosking, J.G. and Grundy, J.C. Vixels, CreateThroughs, DragThroughs and AttachmentRegions in BuildByWire, Proceedings of OZCHI98, Adelaide, Australia, November 30-Dec 4 1998, IEEE CS Press, pp. 320-327.
31. Object Management Group, OMG CORBA, http://www.omg.org/, 1998.
32. O'Neil, J. and Schildt, H. Java Beans Programming from the Ground Up, Osborne McGraw-Hill, 1998.
33. Reiss, S.P., PECAN: Program Development Systems that Support Multiple Views, IEEE Transactions on Software Engineering 11 (3), 1985, 276-285.
34. Reiss, S.P., Connecting Tools Using Message Passing in the Field Environment, IEEE Software 7 (7), July 1990, 57-66.
35. Roseman, M. and Greenberg, S., Simplifying Component Development in an Integrated Groupware Environment, Proceedings of the ACM UIST'97 Conference, ACM Press, 1997.
36. Sessions, R. COM and DCOM: Microsoft's vision for distributed objects, John Wiley & Sons, 1998.
37. Shuckman, C., Kirchner, L., Schummer, J. and Haake, J.M. Designing object-oriented synchronous groupware with COAST, Proceedings of the ACM Conference on Computer Supported Cooperative Work, ACM Press, November 1996, pp. 21-29.
38. Ter Hofte, G.H. and van der Lugt, H.J. CoCoDoC: a framework for collaborative compound document editing based on OpenDoc and CORBA, Proceedings of the IFIP/IEEE International Conference on Open Distributed Processing and Distributed Platforms, 26-30 May 1997, Toronto, Canada, pp. 15-33.
39. Thomas, I. and Nejmeh, B. Definitions of tool integration for environments, IEEE Software 9 (3), March 1992, 29-35.
40. Valetto, G. and Kaiser, G.E., Enveloping Sophisticated Tools into Process-centred Environments, Automated Software Engineering 3, 1996, 309-345.