2nd International Symposium on Constructing Software Engineering Tools (CoSET'2000), Limerick, Ireland, pp. 51-61.

Construction of an Integrated and Extensible Software Architecture Modelling Environment

John Grundy[1]

[1] *Department of Computer Science, University of Auckland,*
Private Bag 92019, Auckland, New Zealand,
Phone: +64-9-3737-599 ext 8761, Fax: +64-9-3737-453, Email: john-g@cs.auckland.ac.nz

Abstract

Constructing complex software engineering tools and integrating them with other tools to form an effective development environment is a very challenging task. Difficulties are exacerbated when the tool under construction needs to be extensible, flexible and enhanceable by end users. We describe the construction of SoftArch, a novel software architecture modelling and analysis tool, which needs to support an extensible set of architecture abstractions and processes, a flexible modelling notation and editing tools, a user-controllable and extensible set of analysis agents and integration with OOA/D CASE tools and programming environments. We developed solutions to these problems using an extensible meta-model, user-tailorable notation editors, event-driven analysis agents, and component-based integration with process support, OOA/D, code generation and reverse engineering tools.

Keywords: software engineering tools, software architecture, modelling notations, analysis agents, tool integration

1. Introduction

Building complex software development tools and integrating these tools with existing 3rd party tools is very challenging [20, 7, 17]. We have been developing a novel software architecture modelling and analysis tool, SoftArch, which presents a number of challenges in its construction. SoftArch needs to support an extensible set of architecture modelling abstractions, visual notations and editing tools. It also needs a user-controllable and extensible collection of model analysis agents to assist with validating an architectural model. Import of OOA specifications and export of OOD models and code fragments is needed, to make use of the tool organisationally feasible.

These requirements are a challenge to meet with conventional tool construction approaches, such as those provided by MetaEDIT+ [12], MOOT [16], KOGGE [3], JComposer [7], and MetaMOOSE [4]. This is because such approaches either produce inflexible, difficult to integrate, configure and extend tools, or provide inappropriate abstractions for building tools like SoftArch.

We describe the implementation of SoftArch using the JComposer meta-CASE toolset and focus on various adaptations we had to make to JComposer's tool development approaches in order to successfully realise SoftArch. We developed an extensible meta-model with its own visual programming language, enabling developers to extend SoftArch's architecture modelling abstractions. Editing tools and notational symbols with a high degree of user-customisability give developers a degree of freedom when representing model abstractions. User controllable and extensible analysis agents were developed using event-driven components, along with a visual end user programming language. The Serendipity-II process management environment [5] provides this event-based end user programming language, plus process and work co-ordination agent support. JComposer itself provides OOA and D model import/export for SoftArch, along with code generation and reverse-engineering support. These tools are integrated using a component-based software architecture. In addition, we have prototyped OOA and D model interchange between SoftArch and Argo/UML [17] using UML models encoded in an XML-based data interchange format. A proposed approach to dynamic architecture visualisation using SoftArch is briefly discussed. We briefly compare and contrast the implementation of SoftArch with other approaches.

2. Overview of SoftArch

There has been a growing need for support for software architecture modelling and analysis tools as systems grow more complex and require more complex architectures [1, 10, 13, 19]. We developed the SoftArch environment to address this need [10]. SoftArch supports the modelling and analysis of large, complex system architectures using primarily multiple views of visual representations of architectural abstractions. SoftArch uses a concept of successive refinements of architecture abstractions, from high-level component characterisations to detailed architectural implementation strategies.

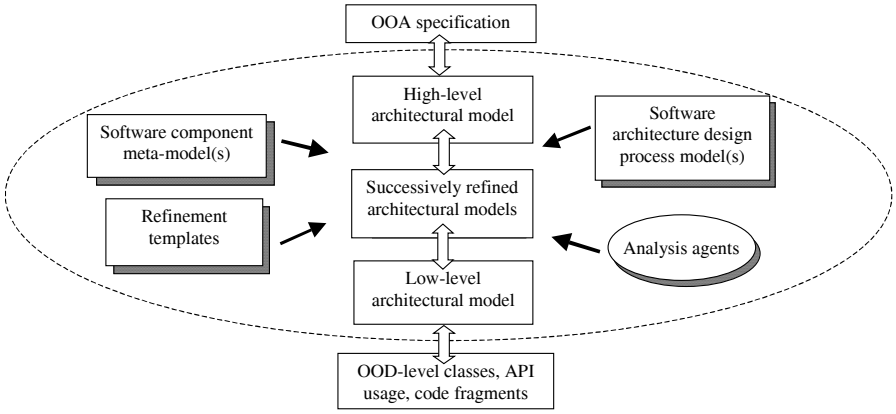


Figure 1. Overview of the SoftArch modelling and analysis approach.

Figure 1 illustrates this concept. An OOA specification (codified functional and non-functional requirements) is imported into SoftArch, typically from a CASE tool. Architects then build an initial high-level architecture for the system that will satisfy these specifications. This high-level model captures the essence of the organisation of the system's software components. It includes information about the non-functional properties of parts of the system, and links architectural components to parts of the OOA specification they are derived from. Architects then refine this high-level model to add more detail, making various architectural design decisions and trade-offs, and ensure the refined architectural models meet constraints imposed by the high-level model. Eventually architects develop OOD-level classes which will be used to realise the architecture, and export these to CASE tools and/or programming environments for further refinement and implementation.

Figure 2 shows an example of SoftArch being used to model the architecture of an e-commerce application (a collaborative travel itinerary planner [8]). The travel planner system is made up of a set of client applications/applets (shown in view (1) at the top). These communicate via the internet to a set of servers, in this example comprising a chat server, itinerary data manager and RDBMS. View (2) shows a more detailed view of the itinerary management part of this system. This includes the itinerary editor client and its connection to the itinerary management server, a client map visualisation, and a map visualisation agent, which updates the map to show a travel path when the itinerary editor client is updated by the user. Architecture components can be refined by creating a subview containing their refinements, by enclosing their refinements (like for "server apps" in view (1)), or using explicit refinement links. OOA and D-level classes and services can also be modelling in SoftArch, and refined to/from appropriate architecture abstractions.

View (3) shows an analysis agent reporting dialogue. A collection of user-controllable analysis agents monitor the state of the architecture model under development. They report inconsistencies, problems or suggested improvements to the user non-obtrusively via this dialogue, are run on-demand by the developer, or act as "constraints" that validate modelling operations as they are performed. SoftArch OOA level abstractions can be sourced from a CASE tool, and OOD-level classes exported to a CASE tool or programming environment (by generating class stubs). Reverse engineering of existing applications is also supported, with OOD-level abstractions able to be imported from a CASE tool and grouped by reverse-refinement into higher-level architectural abstractions.

SoftArch poses various challenges for the tool developer:

- Architectural abstractions include components, associations and component annotations, each which may have a variety of properties [10]. The modelling abstractions available needs to be extensible by the user of SoftArch, to allow them to capture information about the architectural entities they deal with in useful ways, and to add additional components, component properties, etc. as required.
- The modelling notation and editing tools need to be flexible and preferably extensible, supporting model abstraction enhancement and tailorability of the tool. Users should be able to reconfigure the tool to display architecture abstractions as they prefer.
- Templates, or reusable architectural model fragments, are required to assist developers in reusing common architectural styles and patterns. Thus SoftArch must support abstraction of views to templates, instantiation of templates, and ideally support keeping templates and derived model components consistent when either changes.
- Analysis tools that constrain how a model is built and/or check model validity on demand must be user-controllable and extensible. When doing exploratory modelling, modelling alternatives or changing a model dramatically, we have found architects prefer to relax constraints. They then successively re-activate checks as they need them.

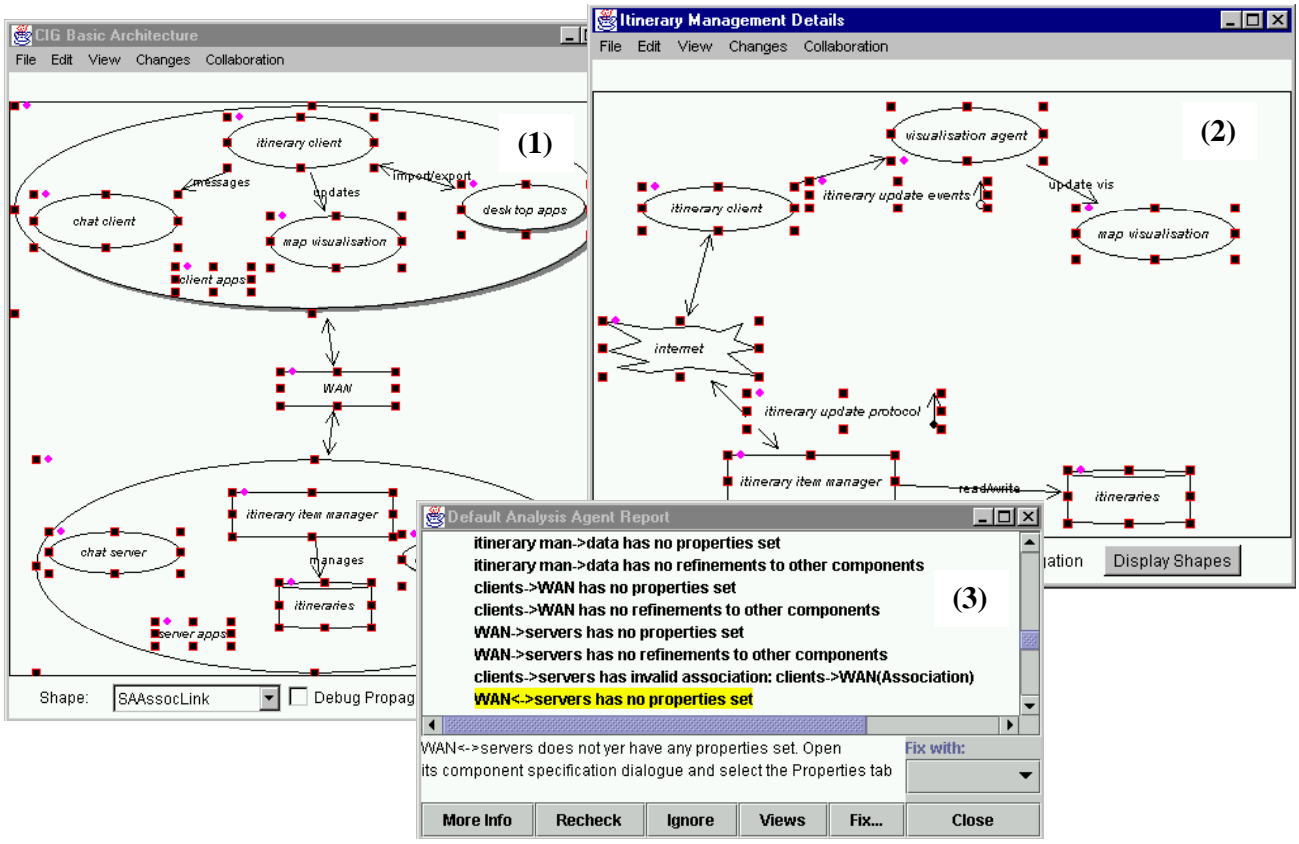


Figure 2. Examples of architecture modelling and analysis in SoftArch.

- The architecture development process should be definable and process management tool support provided to developers. This should not just guide development but also support automated analysis tool activation/deactivation, and configure available modelling abstractions appropriate to the development process stage being worked on.
- Import/export support between CASE tools and SoftArch should leverage existing support within CASE tools where possible. For example, using a CASE tool or programming environment API, using XML-based encoding, or using source code files.

3. SoftArch Architecture and Implementation

The basic architecture of SoftArch is illustrated in Figure 3. SoftArch maintains a collection of meta-model entities, specifying available architectural abstractions and basic syntactic and semantic constraints. A collection of reusable refinement templates supports reuse of common architectural styles and patterns. A collection of analysis agents monitor the changing architectural model and inform the developer of problems. An architecture model holds the current system architectural model (repository, multiple views, refinement links etc.).

We integrated SoftArch with the Serendipity-II process management environment. Serendipity-II provides architecture development process models, work co-ordination agents based on these processes, and user defined analysis agents used to check the validity of SoftArch models. SoftArch was also integrated with the JComposer component engineerint tool. JComposer provides OOA-level class components for SoftArch and SoftArch generates OOD-level class components in JComposer. SoftArch also uses JComposer's code generation facilities to generate Java classes based on OOD-level architectural abstractions and middleware and database component properties described in SoftArch. Generated Java classes can be modified in tools like JBuilder and JDK, and changes reverse-engineered back in JComposer and then into SoftArch. We have prototyped simple XML-based import/export tools, which exchange OOA and D models with Argo/UML.

We implemented SoftArch with the JViews multi-view, multi-user software tool framework, using the JComposer meta-CASE and component engineering toolset [7]. Our JComposer tool also provides a component engineering environment for JViews.

We encountered a number of challenges when using JViews and JComposer to engineer SoftArch. JComposer does not directly support extensible meta-models for CASE tools, however, and its notation tailoring tool enables users to inappropriately modify notation-implementing editors and icons.

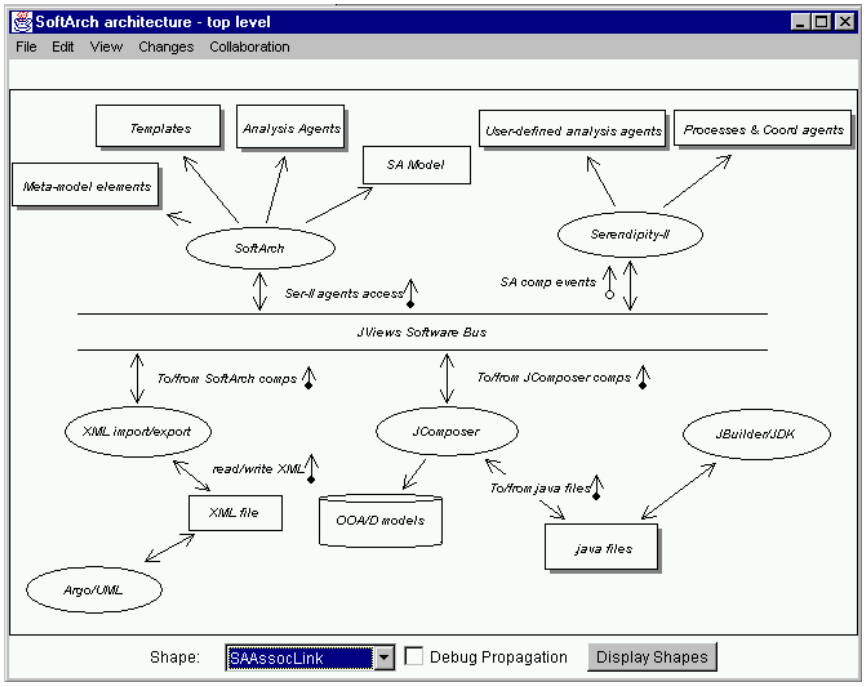


Figure 3. Basic architecture of SoftArch.

Flexible and extensible analysis tools can be built, but no direct abstractions are provided by JViews, and some Java programming is required to achieve these. Tool integration is supported directly via component interface-based mechanisms, but indirectly via components implementing 3rd party tool communication protocols and data exchange format parsing and generation.

The following sections examine the construction of various SoftArch facilities in further detail, focusing on the approaches we used to satisfy some of these more challenging requirements of the tool. As JViews and JComposer did not directly support many of these capabilities, we discuss how we overcame these short-comings. We then discuss the various lessons we learned from developing SoftArch, and summarise some directions for future software tool construction approaches we have been exploring because of this work.

4. Architecture Modelling

5.1. Meta-model Support

SoftArch uses a basic model of architecture components, inter-component associations and component and association annotation to describe architectural models [10]. Each of these architectural entities has a set of properties associated with it. Property values can be simple numbers or strings, or a collection of value ranges. JComposer, like most meta-CASE tools, assumes a tool developer would have a fixed set of tool repository component and relationship types e.g. process stages, in/out ports, filters and actions in Serendipity-II, and components, association, generalisation, aspects etc. in JComposer itself [5, 7]. Thus with SoftArch there might be a fixed set of different architecture component, association and annotation types, each with a fixed set of properties, which could each be modelling as appropriate JViews repository component specialisations.

However, in order to support user-extension of SoftArch's software architecture modelling capabilities, we had to develop a meta-model for SoftArch in JComposer, as well as the component/association/annotation architecture model repository representation. Figure 4 (a) illustrates the basic components of this meta-model. SoftArch components, associations and annotations must each have a type, with the meta-model allowing the specification of valid component associations and annotations. Each different type has a set of properties, which have property type and value constraints. For example, component types include "SA Entity", "OOA class", "Process", "Server", "Client Process", "RDBMS" etc. Association types include "dependency", "data usage", "event subscribe/notify", "message passing" etc. Annotations include "cached data", "data exchanged", "events exchanged", "replicated data", "process synchronisation", etc.

Component (and association and annotation) types also specify valid refinements allowed. For example, the most general "SA Entity" component can be refined to any other kind of architecture component when modelling. The "Client Process" type cannot, however, be refined to "Server Process" or "RDBMS" components, as such a refinement does not make any sense.

Unlike most CASE tools, SoftArch does not inherently enforce constraints like valid associations/annotations, valid refinements or valid properties/property values for components. A set of

analysis agents does this, and selected agents can be turned on and off to allow architects greater or lesser flexibility to model and change architectures (see Section 6). We found this facility to be very useful when architects dramatically change an architecture, or are doing alternative or exploratory modelling of parts of an architecture. Relaxing some constraints makes it easier for architects to morph or revise parts of the model through partially inconsistent states, than if meta-model typing constraints are always rigidly enforced.

We allow users of SoftArch to open JViews projects, which contain partial meta-model specifications. Meta-model components in different projects build upon one another to construct a complete set of component and other types available when modelling architectures with SoftArch. Users can extend the meta-model using a simple visual specification tool, illustrated in Figure 4 (b). Using multiple meta-model projects allows architects to package domain-specific meta-models e.g. "basic abstractions", "real-time systems", "e-commerce systems" etc., each with specialised architecture modelling abstractions.

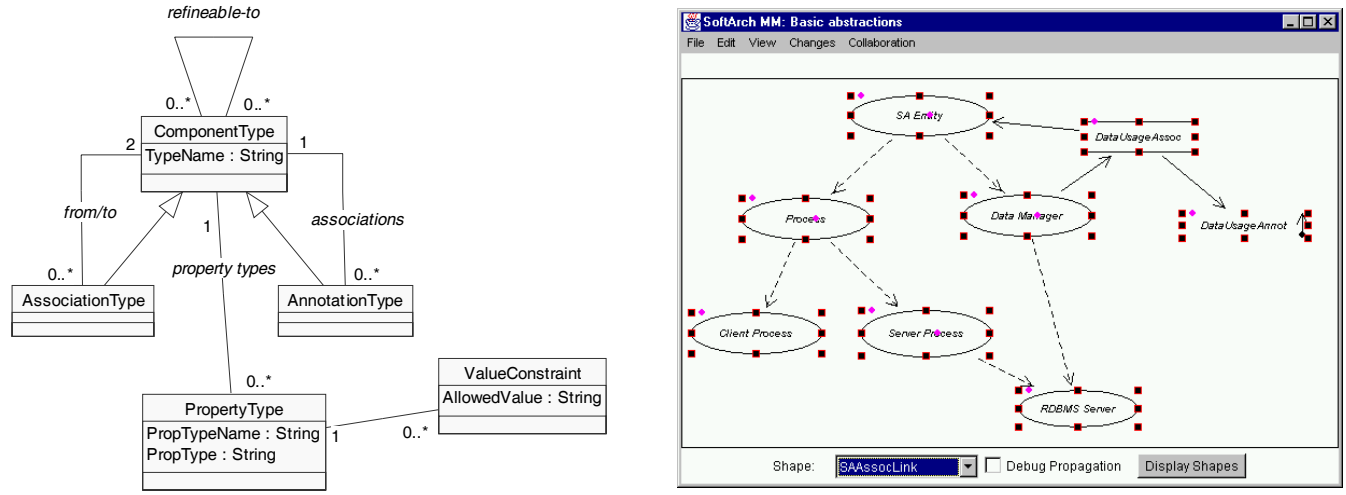


Figure 4. (a) SoftArch meta-model; (b) visually viewing and programming the meta-model.

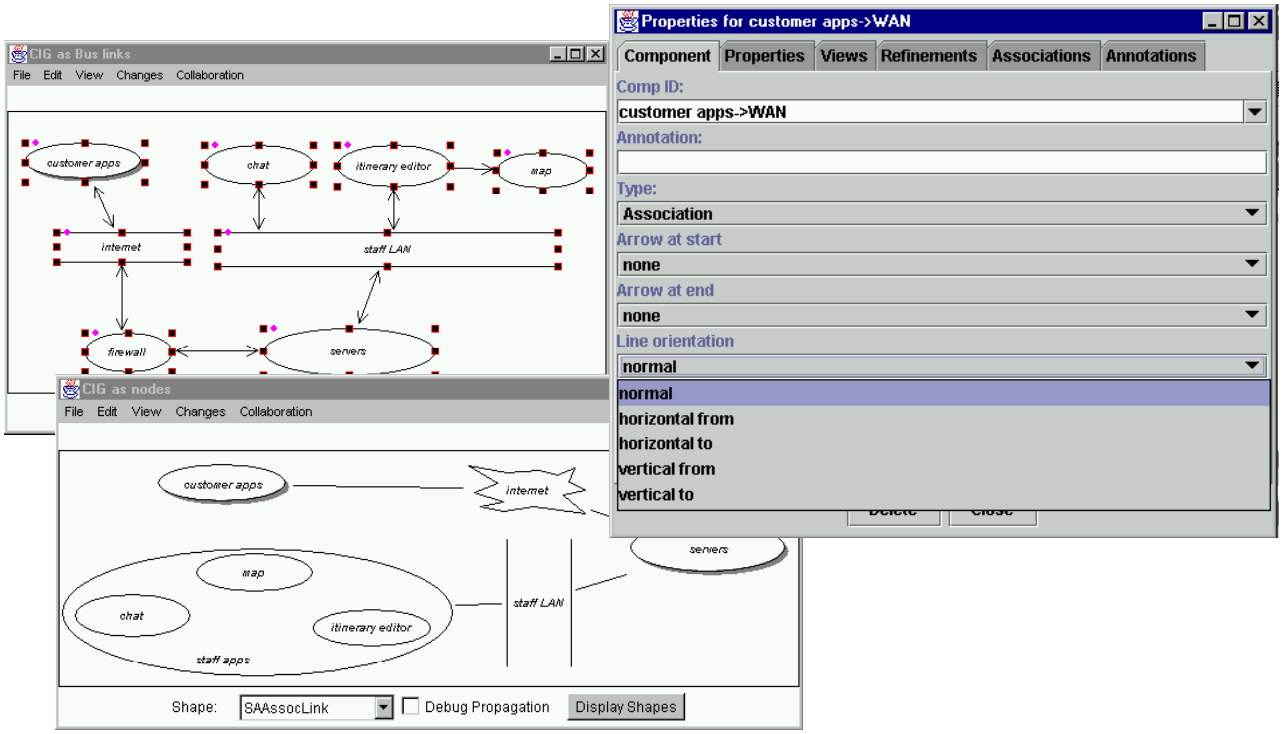


Figure 5. Examples of SoftArch Notation Usage.

5.2. *Flexible Modelling Notation*

JComposer provides a notation icon editor, BuildByWire, which can be used by tool users to reconfigure their icon appearance in certain ways [7]. With SoftArch, we decided to take an alternative approach and provide users with a range of icon appearances that they could tailor as they required via the same dialogue used to specify and view architecture component properties. For example, Figure 5 shows two examples of modelling the same information in SoftArch, the top view using bus-style associations between client and server components and the bottom node-style connectors and enclosure of clients running on the same host. The dialogue shown provides configuration capabilities allowing users to tailor the appearance of component, association and annotation icons as they require. Automated tailoring can be achieved using Serendipity-II task automation agents (see Section 6).

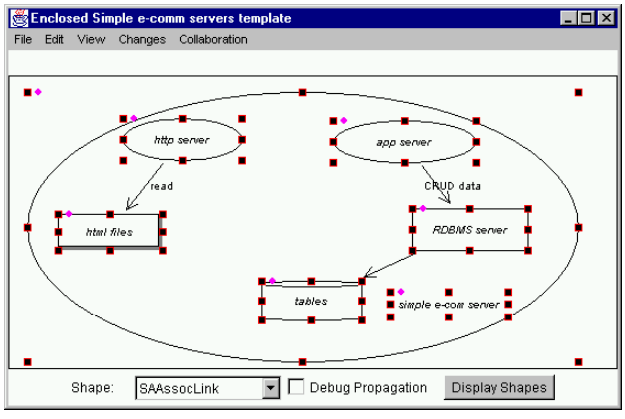
We adopted the customisable icon appearance approach over having end users use BuildByWire directly as it is much easier and quicker for them to tailor icons, and they do not need to learn to use the meta-CASE tool. They also can not make errors and cause SoftArch to fail, which is possible using BuildByWire directly. Users can, however, use the BuildByWire meta-CASE tool to extend the possible icon appearances if no pre-defined ones suit their needs.

5.3. *Refinement Templates*

In order to support reuse of common architectural styles and patterns, we developed reusable refinement templates for SoftArch. A view in SoftArch which specifies the refinement of one (or more) architectural components into more detailed architectural model components can be copied and packaged for reuse. For example, Figure 6 (a) shows a packaged refinement template commonly used in simple e-commerce applications. The high-level component "simple e-com server" encloses (and thus is refined to) several parts: an http server with html and other files, an application server, and an RDBMS server with tables. Figure 6 (b) shows how the user of SoftArch has reused this refinement template when developing part of the travel itinerary system's architecture. SoftArch allows users to reuse refinement templates by creating subviews for a specified component or by automatically copying the template components into their model (as in this example).

JViews does not explicitly support the concept of templates. When developing Serendipity-II's process templates we built a complex mechanism for copying and instantiating template process models [5].

When developing SoftArch refinement templates we instead extended the versioning and import/export mechanisms JViews supports. A template is created by exporting a view to a file then importing it and using JViews' component identifier (ID) mapping mechanism to create a template. When instantiating a template, we export the template to a file then import it, using the same ID mapping mechanism to create new components with unique IDs in the software architecture model. Refinement links are created automatically by SoftArch for subviews, and are created automatically for imported enclosed components and explicit refinement links.



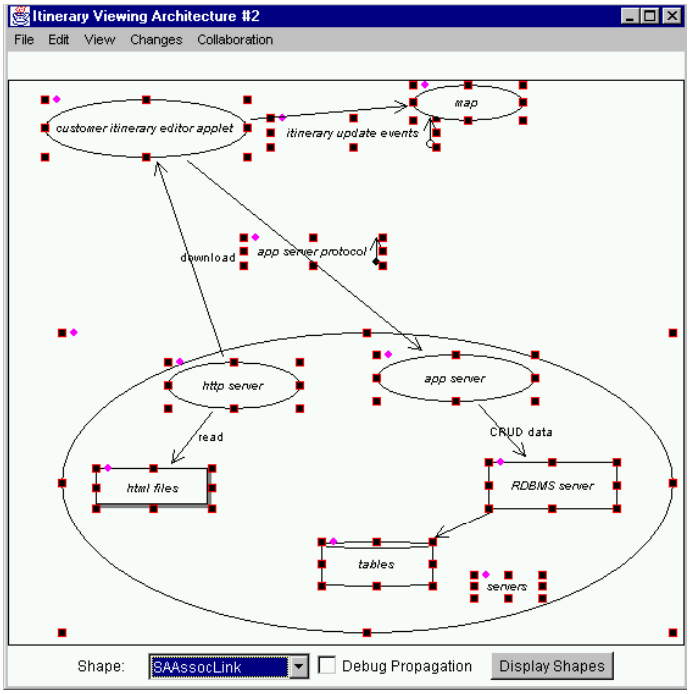


Figure 6. (a) Example of SoftArch template; (b) reused template.

This approach proved to be a much simpler solution than that used for Serendipity-II, but provides almost identical template support. JViews' version merging abstractions [7] can even be used to reconcile changes made to the template or components copied from the template into the architecture model.

5. Process and Analysis Support

6.1. Process Management

We wanted to provide SoftArch users with integrated process management support to allow them to use enacted process models to both guide and track their work. It would also automate tedious tasks like enabling/disabling analysis agents and configuring allowable component types and notation appearance during different stages of architecture model development. Rather than building process support into SoftArch, as done in Argo/UML [17], using CAME tools like MetaEdit+ with very limited automation support [12], or forcing developers to configure the tool themselves, as in Rational Rose [15], we reused the Serendipity-II process management environment.

Figure 7 shows a simple architecture development process in Serendipity-II, along with a task automation agent which enables and disables groups of analysis agents when a particular process stage is enacted or finished. Serendipity-II detects changes made to SoftArch models and records these against process stages, allowing developers to track work associated with different process tasks/subtasks. The task automation agent illustrated here detects process activation/deactivation (the left-hand square icons, or "filters), then uses two actions (shaded ovals) to enable and disable named SoftArch analysis agents (right-hand side rectangles). The actions send events to the SoftArch analysis agent manager to enable or disable the named SoftArch analysis agents. The filters and actions used here are reused from a library of such event-driven components. Others can be implemented using JComposer and Java and added to this library as required.

This integration is achieved by Serendipity-II using JViews' component event propagation mechanism to listen to SoftArch component events and to record these. The task automation agents, like the one shown here, send events to SoftArch which configure analysis agents, configure display of notational symbols and configure available meta-model abstractions. This produces what seems to the developer to be a more or less seamlessly integrated process management and task automation tool support for SoftArch.

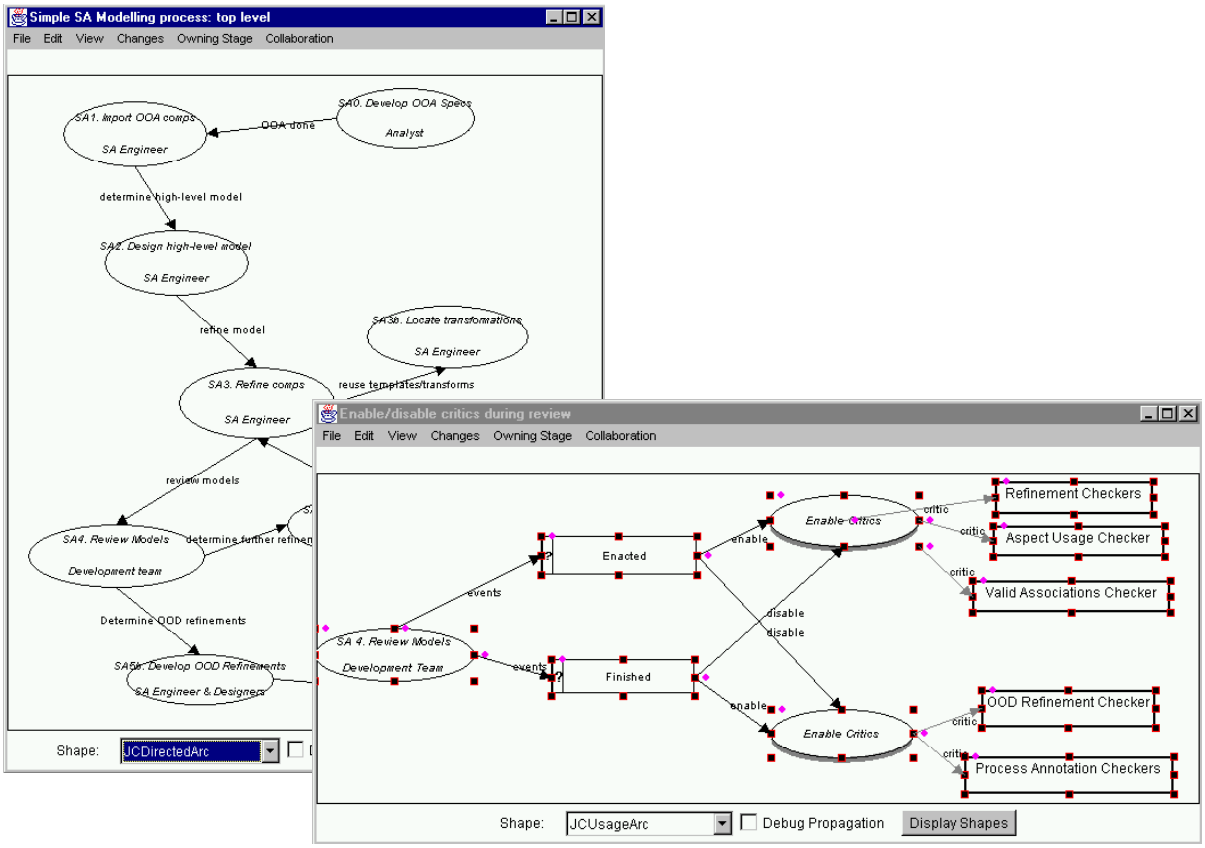


Figure 7. (a) Simple software process; (b) simple analysis co-ordination agent.

5.2. Design Constraints, Critics and Analysis Agents

SoftArch's meta-models have a set of analysis agents (implemented by event-driven JViews components) which monitor the state of the architecture model being developed. Agents may be fired immediate an invalid action is made e.g. incorrect association type specified between two architecture components, and the editing operation reversed and an error dialogue shown. Alternatively, they can monitor changes and unobtrusively add messages to an analysis report dialogue (like the one shown in Figure 2), or can be run on-demand by developers and their error messages displayed as a group. Users can control the way an analysis agent behaves using a control panel dialogue e.g. change an agent from running as a constraint to a critic, enable or disable agents etc. As show in Figure 7, Serendipity-II visually-specified task automation agents can also be used to control analysis agents.

Users can also extend the set of analysis agents being applied to a SoftArch model by using Serendipity-II's task automation agent specification tool. Figure 8 shows a user-defined analysis agent that checks to see if a component has associations (either from it to other components, or to it from other components). The top "guard" filters are fired when a component has been changed, and then following filters determine if the component has associations to/from it. If neither, an action (bottom oval icon) generates an error event which the analysis agent manager displays in an error dialogue (if this agent is run as a constraint) or displays in an analysis agent report.

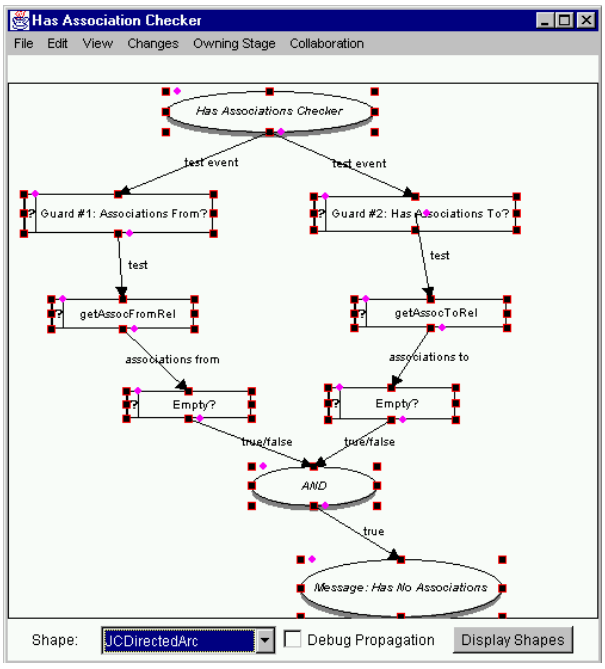


Figure 8. Simple visual analysis agent specification.

6. Tool Integration

7.1. OOA/D Import & Export

Many tools exist which provide object-oriented analysis and design capabilities. Our own JComposer is one such example, but others include CASE tools like Rational Rose [15] and Argo/UML [17]. We originally planned SoftArch as an extension to JComposer, but decided it would be more useful as a stand-alone tool, that could ultimately be used in conjunction with other, 3rd party CASE tools.

SoftArch requires constraints from an OOA model, particularly non-functional constraints like performance parameters, robustness requirements, data integrity and security needs and so on. These constrain the software architecture model properties that needs to be developed in order to realise the specification. These also influence the particular architecture-related design decisions and trade-offs software architects need to make. Similarly, a SoftArch architectural model is little use on its own, but needs to be exported to a CASE tool and/or programming environment for further refinement and implementation. Some code generation can even be done based on a SoftArch model description e.g. appropriate middleware and data management code generated. When reverse engineering an application, an OOD model will need to be imported into SoftArch and a higher-level system architecture model derived from it. Ultimately an OOA specification may be exported from SoftArch to a CASE tool. Thus SoftArch must support OOA and D model exchange with other tools, and ideally some code generation support.

We initially used a JComposer component model as the source for SoftArch OOA-level specification information. JComposer allows not only functional requirements to be captured, but has the additional benefit of requirements and design-level component "aspects", which are used to capture various non-functional requirements [9]. We developed a component that supports basic component and aspect import into SoftArch from a JComposer model, using JViews' inter-component communication facilities to link SoftArch and JComposer.

Rather than add OOD and code generation support to SoftArch itself, we leveraged existing support for these in JComposer. SoftArch uses JComposer's component API to create OOD-level components (classes) in JComposer, and instructs JComposer to generate code for these to produce .java files. JComposer supports a concept of code fragments, which SoftArch uses to generate some basic Java component configuration, communications and data access code for generated classes. Figure 9 illustrates the interaction of JComposer and SoftArch to achieve OOA import and OOD/P export for SoftArch.

JComposer was reasonably straightforward to integrate with SoftArch as JComposer provides a JViews-implemented, component-based API. Other CASE tools and programming environments do not generally provide

such open, flexible integration mechanisms. Generated .java class source code files can be used in tools like JDK and JBuilder, and changes reverse engineered back into JComposer and then into SoftArch. We have prototyped a data interchange mechanism to enable SoftArch to exchange OOA and D models with Argo/UML using an XML-based encoding of UML models. This is a less tightly integrated mechanism than that used by SoftArch and JComposer, but allows other tools using the XML exchange format for UML models to be integrated with SoftArch in the future.

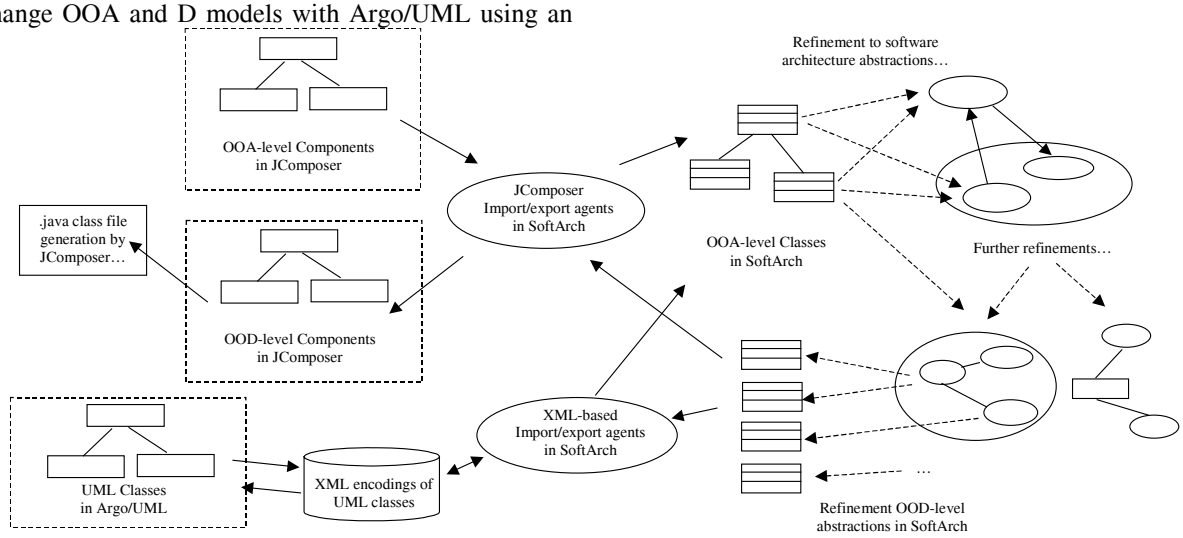


Figure 9. Import/export approaches in SoftArch.

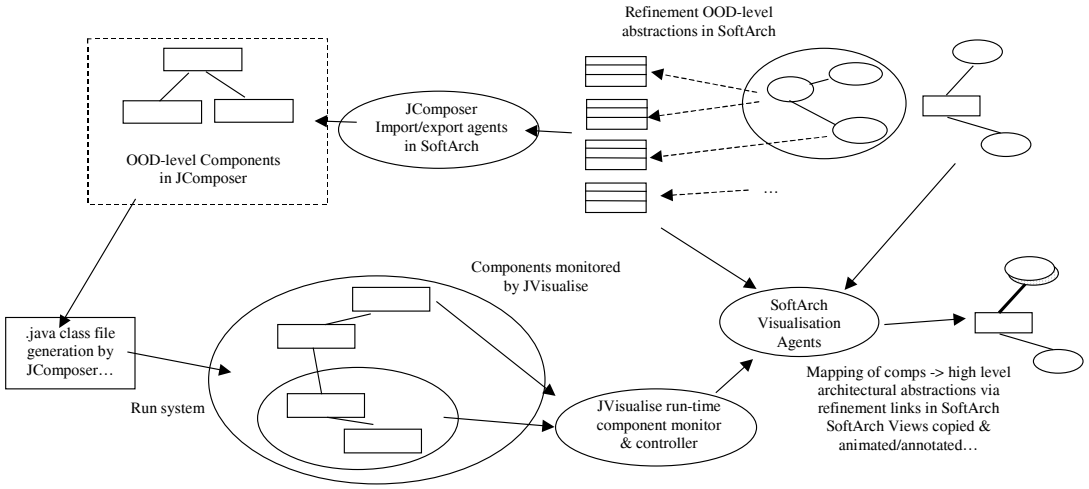


Figure 10. Planned dynamic architecture visualisation using SoftArch views.

7.2. Runtime Architecture Visualisation

So far we have discussed static architecture modelling, analysis and OOD/code generation support with SoftArch. Ultimately we would like to extend SoftArch's support for architecture modelling to include dynamic architecture visualisation and configuration i.e. run-time visualisation and manipulation of software architecture abstractions using SoftArch-style views. We are beginning work to achieve this by making use of our JVisualise component monitoring and configuration tool [7]. JVisualise allows running JViews-based systems to be viewed using JComposer-style visual languages. Users can also manipulate visualised components – changing their properties, adding or removing inter-component relationships, and creating new component instances. We intend to enhance JVisualise to enable any JavaBean-based application to be thus monitored and controlled.

Figure 10 illustrates how SoftArch will be used to visualise and configure running software architectures. JVisualise will request running components send it messages when they generate events, and will create proxies to enable it to intercept operation invocations on components. SoftArch will instruct JVisualise to send it these low-level component monitoring events, which will be mapped onto SoftArch OOD components using the JComposer-generated Java class names. SoftArch will then allow users to view information about running components using higher-level SoftArch views, as OOD-level components will have refinement relationships to

higher-level architecture components in these views. For example, when components implementing a server are created and the server initialised, SoftArch can show a single server component has started in a high-level SoftArch view. Similarly, when the server component receives a message from a client, SoftArch can annotate a high-level association link to indicate this. The user may add a client component to this dynamic visualisation view and connect it to the server. SoftArch can instruct JVisualise to create appropriate components which implement the client and initialise them.

JComposer-generated OOD models and code may be extended if necessary to include additional monitoring components and wrappers to intercept data and communication messages. JVisualise would use these to provide improved event and message monitoring and control support.

7. Discussion

A wide variety of tools and approaches exist with which to build a system like SoftArch. General-purpose programming languages and frameworks, such as Java and JFC, Borland Delphi, Smalltalk, or similar, can be used to implement such a tool "from scratch". However, many tool facilities required by SoftArch, including multiple views with consistency management, multi-user support, version control, persistency and distribution, and so on, are time-consuming to build using such approaches. In addition, building tools with extensible meta-models, visual languages and tool integration mechanisms with these low-level abstractions is extremely difficult.

General purpose drawing editor frameworks, such as Unidraw [21] and Hotdraw [2], could be used to provide editing support, and middleware architectures like CORBA [14], DCOM [18] and Xanth [11] used to support distribution and transparent persistency. Again, these technologies assist tool developers but still lack appropriately focused software tool building abstractions. An existing CASE tool, such as JComposer [7], MOOSE [4] or Argo/UML [17] could be extended to add SoftArch-style support. However, such an approach would make an already very complex tool more monolithic, the existing CASE tool infrastructure may not support some desired characteristics of SoftArch, and the resultant tool may not be usable with other 3rd party tools.

A variety of meta-CASE and CAME tools exist which might be usefully employed. Examples include KOGGE [3], MetaEDIT+ [12], MetaMOOSE [4], MOOT [16], and JComposer [7]. Tools like MetaEDIT+ and KOGGE provide a range of abstractions and tools enabling quick development of conventional CASE tools. Unfortunately they do not support well the need for users of SoftArch to extend architecture model abstractions and notations, do not provide adequate model analysis tool building support. MOOT and MetaMOOSE provide better support for extensible meta-models for software tools, and reasonably tailorable notations. However, they do not support template reuse well, and their analysis tool and tool integration capabilities are limited.

We found our JComposer tool to be of relatively limited usefulness in developing SoftArch. JComposer and its underlying framework, JViews, do not directly support the concept of an extensible tool meta-model, user-configurable icons for visual languages, patterns and templates, model analysis and process co-ordination, and flexible tool integration support. Process co-ordination and tool integration are provided by additional plug-in components (for example, the Serendipity-II process management tool for processes, and various components for database, remote server and XML data encoding use). This support could be improved to make build environments like SoftArch easier.

Allowing users to dynamically extend the meta-model of their enviroments, the visual languages they model with, the analysis tools and incorporate integration mechanisms with third-party tools are all very difficult in general. Our approach with SoftArch has been to build a JViews meta-model that has its own visual programming language, and have SoftArch use this model to validate architecture models. This proved challenging to realise, as JViews components designed for building software tools weren't built with a meta-model in mind, but rather a fixed, JComposer-generated model. Re-architecting both JViews and JComposer is required to provide suitable abstractions that make it easier to build such facilities. Similarly, while we developed the BuildByWire visual tool for iconic specification, this was not intended for use by tool users directly, but for tool developers. We need to modify the architecture of this to better-support end user configuration of iconic appearance, while retaining tool editing semantics.

We have built some reusable components in JViews which can be deployed for use in other environments to support analysis agent specification. We have also developed some basic agents in Serendipity-II that can be deployed by end users to extend the constraint and analysis checking of their tools while in use. However, these require further development to become easier to use by both tool developers and users. Similarly, our tool integration components built for SoftArch could be usefully generalised to make building file and XML-based tool integration easier. We are extending JViews' support for patterns and templates, and also extending JViews and JComposer to provide higher-level dynamic monitoring to better support visualisation of running SoftArch-modelled systems.

Alternative approaches to building SoftArch might have used a meta-CASE tool which allows end users to extend a meta-model and/or visual notation. However, most meta-CASE tools, like JViews, assume tool developers specify such meta-level constructs, rather than tool users. Another approach would be to use tools designed for end user computing, somewhat like Serendipity-II's process modelling and agent specification tools. In fact, we originally explored building most of

SoftArch using Serendipity-II in this fashion. Unfortunately the abstractions supported by such an approach for SoftArch-style notations, architecture models and analysis are very difficult to express in such end user computing tools, and the efficiency and extensibility of the resulting solution likely to be poor.

8. Summary

We have described the construction of the SoftArch software architecture modelling and analysis tool. SoftArch requires a number of facilities that are challenging to build using conventional tool development approaches. We achieved the aim of an extensible set of modelling abstractions and notations by using a user-extensible meta-model and set of user-customisable icons. Reusable refinement templates are supported by SoftArch, leveraging component import/export and version merging capabilities of our tool implementation framework. Process support, including work co-ordination and user-defined analysis agents, are supported by integrating SoftArch with the Serendipity-II process management environment. OOA/D import/export and code generation and reverse engineering support are provided by integrating SoftArch with the JComposer component engineering/meta-CASE environment and the Argo/UML CASE tool.

We are investigating extending our JComposer meta-CASE toolset to better support meta-models for software development tools, and to provide abstractions for template and pattern reuse. In addition, we are investigating other process management tool integration approaches, such as the workflow management coalition's process interchange format. We are also investigating other interchange formats for CASE tools and programming environments, allowing more OOA specification information, especially non-functional requirements codification, to be exchanged, along with improved OOD and code generation facilities. We are beginning to develop an exploratory dynamic architecture visualisation and configuration facility, using SoftArch and the JVisualise component monitoring tool.

References

1. Bass, L., Clements, P. and Kazman, R. Software *Architecture in Practice*, Addison-Wesley, 1998.
2. Beck, K. and Johnson, R. Patterns generate architectures. *Proceedings ECOOP'94*, Bologna, Italy, 1994.
3. 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.
4. 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.
5. Grundy, J.C., Hosking, J.G., Mugridge, W.B. and Apperley, M.D. An architecture for decentralised process modelling and enactment, *IEEE Internet Computing*, Vol. 2, No. 5, September/October 1998, IEEE CS Press.
6. Grundy, J.C. and Hosking, J.G. Directions in modelling large-scale software architectures, In *Proceedings of the 2nd Australasian Workshop on Software Architectures*, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
7. Grundy, J.C., Hosking, J.G. and Mugridge, W.B. Constructing component-based software engineering environments: issues and experiences, *Journal of Information and Software Technology: Special Issue on Constructing Software Engineering Tools*, Vol. 42, No. 2, January 2000, pp. 117-128.
8. Grundy, J.C., Mugridge, W.B., Hosking, J.G., and Apperley, M.D., Tool integration, collaborative work and user interaction issues in component-based software architectures, In *Proceedings of TOOLS Pacific '98*, Melbourne, Australia, 24-26 November, IEEE CS Press.
9. Grundy, J.C. Aspect-oriented Requirements Engineering for Component-based Software Systems, In *Proceedings of the 4th IEEE Symposium on Requirements Engineering*, Limerick, Ireland, June 1999, IEEE CS Press, pp. 84-91.
10. Grundy, J.C. *Software Architecture Modelling, Analysis and Implementation with SoftArch*, Technical Report, Department of Computer Science, University of Auckland, December 1999.
11. Kaiser, G.E. and Dossick, S. Workgroup middleware for distributed projects, *Proceedings of IEEE WETICE'98*, Stanford, June 17-19 1998, IEEE CS Press, pp. 63-68.
12. 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.
13. Leo, J. OO Enterprise Architecture approach using UML, In *Proceedings of the 2nd Australasian Workshop on Software Architectures*, Melbourne 23rd Nov 1999, Monash University Press, pp. 25-40.
14. Mowbray, T.J., Ruh, W.A. *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley, 1997.
15. Quatrani, T. *Visual Modelling With Rational Rose and UML*, Addison-Wesley, 1998.
16. Phillips, C.E. , Adams, S., Page, D. and Mehandjiska, D., Designing the client user interface for a methodology independent OO CASE tool, *Proceedings of TOOLS Pacific'98*, Melbourne, Nov 24-26, IEEE CS Press.
17. Robbins, J.E. and Redmiles, D.F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, *Proceedings of CoSET'99*, Los Angeles, 17-18 May 1999, University of South Australia, pp. 61-70.
18. Sessions, R. *COM and DCOM: Microsoft's vision for distributed objects*, John Wiley & Sons 1998.
19. Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
20. Thomas, I. and Nejmeh, B. Definitions of tool integration for environments, *IEEE Software*, vol. 9, no. 3, March 1992, 29-35.
21. Vlissides, J.M. and Linton, M.A. Unidraw: a framework for building domain-specific graphical editors, *ACM Transactions on Information Systems*, vol. 8, no. 3, July 1990, 237-268.