Package-oriented programming treats mass-market applications as large components to build sophisticated software development tools.

Giancarlo Succi, Witold Pedrycz, Eric Liu, and Jason Yip

Package-Oriented Software Engineering: A Generic Architecture

New methodologies and better techniques are the rule in software engineering, and users of large and complex methodologies benefit greatly from specialized software support tools. However, developing such tools is both difficult and expensive, because developers must implement a lot of functionality in a short time.

A promising solution is component-based software development (CBSD), in particular, a CBSD specialization called *package-oriented programming* (Kevin Sullivan and colleagues, "Package-Oriented Programming of Engineering Tools," *Proc. 19th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1997.) POP treats regular mass-market applications (such as Microsoft Office or Rational Rose) as large components, taking advantage of the abundant functionality and user familiarity that come with them. Often, the right tool for a particular job exists as a full-fledged application, but developers or managers do not consider it a component in the traditional sense. POP does away with this restriction.

POP fails, however, to satisfy all the requirements of large, complex software engineering tasks. First, it does not support multiuser operation, and fields such as domain analysis and requirements engineering frequently involve more than one user. Second, systems using POP have so far relied only on architecturally compatible components, especially those of Microsoft's COM specification. POP-based systems like Galileo and the ISI design editor generator use COM because applications that fit their needs implement COM. Integrating with COM applications is attractive, because the communication mechanism is relatively easy to implement and you can achieve a very tight integration. However, such a focus restricts integration to applications primarily compatible with Microsoft Windows; they must also be architecturally compatible with COM.

Third, traditional POP also seems to lack platform integration. Today, a typical network is heterogeneous, consisting of a mix of Windows, Linux, and other Unix machines. Each platform provides different tools to accomplish different tasks, and these tools must be integrated in a way that lets them work on the same data remotely.

A more generic POP architecture would better serve the development of software engineering environments for large and complex methodologies. Such an architecture emerged from our development experiences with two software engineering research tools:

- Holmes, a domain analysis support tool; and
- Egidio, a unified-modeling-language-based business modeling tool.

We found this particular architecture simple to understand, easy to implement, and a natural candidate for a generic POP architecture.

Inside

1520-9202/01/$10.00 © 2001 IEEE

Critiquing-Tool Basics

Certain domains are more suitable for critiquing than others; these include those with

- **several alternative solutions,**
- **several risks and benefits associated with the various solutions, and**
- **new solutions and knowledge that periodically alter the field.**

Critiquing systems employ two strategies—active and passive—to interact with users. An active critiquing system continuously monitors user actions. As soon as it detects a problem, the system makes suggestions. A passive system requires users to explicitly invoke it to evaluate a partially completed design. Studies show that users don't activate passive systems early enough to prevent designers from spending time on design solutions that are known to be suboptimal or have problems.

Some existing tools incorporate critiquing systems. Lisp-Critic lets Lisp programmers request improvements for their code. This critiquing system suggests code transformations that make the code easier to read or maintain. It also suggests ways to make the code more machine efficient.

Developers have also built critiquing systems for multimedia authoring. The eMMaC system helps casual users harness the power of high-functionality authoring tools. It critiques the use of color combinations and color balance. The rules behind the system come from a community of multimedia authoring tool users.

The following example illustrates the usefulness of a critiquing system in software design. In a domain-modeling situation, a designer works with UML class diagrams and could accidentally introduce circular inheritance in a class hierarchy. The critiquing system notices this flaw, informs the designer, and advises removing the circular inheritance.

On the other hand, the system should not impede seasoned designers in their work. Rather, it should be a passive guide, providing advice that can be followed or ignored as appropriate.

TOOLS FOR LARGE AND COMPLEX SOFTWARE ENGINEERING TASKS

There are several large and complex software engineering fields that can benefit from some form of automated support. Tools for such software must support

- multiple users and a distributed mode of operation,
- many varied but interrelated activities,
- traceability of data from different activities,
- customizable support specific to each individual activity, and
- quick integration of existing software packages.

These needs are also applicable to other large software engineering methodologies.

Today, large software development projects involve a team of developers, managers, domain experts, and market experts. All these stakeholders have distributed and concurrent interactions with the system. The issues of data and change consistency are critical to successfully support such a distributed, multiuser operation.

Most methodologies involve multiple, different yet interrelated, activities. For instance, a typical domain analysis methodology would include activities involving domain definition, scoping, and some type of modeling. Users of such a methodology would perform different activities at different times. And, for each of these activities, a suitable or superior tool is probably already available. Thus, support for integrating several software packages is crucial. As a minimum, tool integration should support data-level integration. This helps the system keep data and changes consistent.

Users also would like a sense of traceability between the activities—how does changing data in one activity affect data presented in another? This is especially important for a large and complex methodology. For instance, a requirements engineer might be interested in knowing whether changing a particular requirement will affect the test engineers' functional tests.

Users would also benefit greatly from support for semantic correctness. The system should offer advice based on what the user is doing. If users are inexperienced (or just careless), such a system could guide them away from common pitfalls, even offering suggestions on what they should do instead.

A typical example comes from system design. Designers sometimes get carried away with a class inheritance hierarchy. At some point, the tool should advise designers to limit the depth of such a hierarchy because historical data suggests that defects are more likely to occur with a deeper hierarchy. Designers could change the design and avoid potential problems down the road.

By using existing components, POP leads to quicker integration.

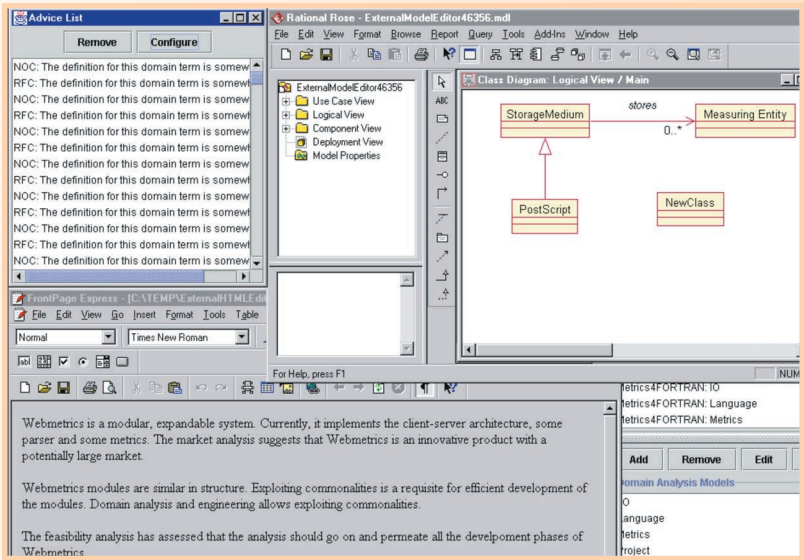
DEVELOPING SOFTWARE ENGINEERING ENVIRONMENTS: OUR EXPERIENCE

There are two ways in which to use packages as components; either

- use one package as the platform on which to build the new system or
- integrate multiple applications tightly into a single system.

We used the second approach to develop two systems using the same architecture, each supporting a different software engineering methodology. Those systems and their respective methodologies illustrate the importance of the requirements we've just outlined.

Figure 1. Holmes integrates our design-critiquing system (Design Critic), FrontPage Express, and Rational Rose.



Holmes

Holmes is a domain analysis tool that supports Sherlock, a software engineering methodology. Sherlock has multiple participants and many related activities, divided into five phases: domain definition, characterization, scoping, modeling, and framework development.

Sherlock has some activities that involve large designs and difficult decisions, making semantic support an important aspect of Holmes. For instance, Holmes will warn users about adding terms that are duplicated, possibly with other meanings in another activity. Holmes will also assist users in determining a suitable product strategy to make a product more competitive in a reasonable development time.

Domain objects for Sherlock are statically stored in XML. This opens up possibilities for tools that work with data offline. Furthermore, if the Holmes system should somehow become obsolete or unusable in the future, the data would remain usable because XML preserves structure.

In addition, some major activities in the methodology deal with rich text and UML (Unified Modeling Language) modeling. Creating custom text editing and UML modeling tools would waste time. So Holmes is integrated with Microsoft FrontPage Express (an HTML editor) and Rational Rose (a UML modeling tool) to reduce development time. Figure 1 shows how the Holmes interface displays these tools. We implemented the tools for the remaining activities as Java Swing components.

Egidio

Egidio is a specialized tool for software business process modeling and human resource management. Its methodology uses various types of diagrams, such as matrices and UML class diagrams, to provide different views of the business process. There are also multiple activities in which users specify employee roles, activities, assignments, skills, and so on.

Since we had to develop several custom views for this tool, including the view shown in Figure 2, we used the Holmes architecture to save time. In this way, our development effort concentrated on developing the views and widgets, instead of on the cross-communication among tools. This strategy worked well, and we completed the system in a short time.

A GENERIC ARCHITECTURE

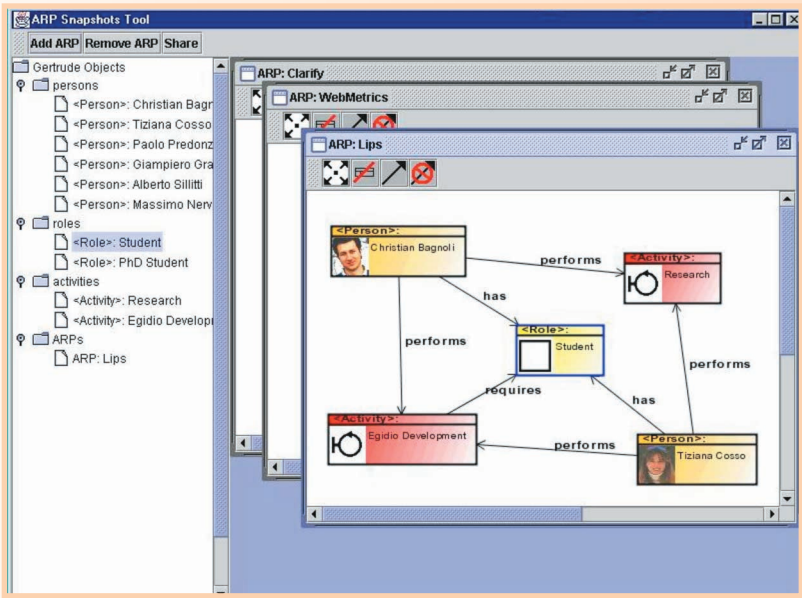
The architecture used in our projects emphasizes JavaSpaces, the Sun specification of a Java-based tuple space (associative shared memory) based on Linda, a tool to support blackboard architectures on distributed memory multiprocessors. A space is essentially a shared blackboard of objects. The concept of putting, matching, and getting entries (the JavaSpaces equivalent of tuples) is simple, easy to program, and naturally fulfills the requirements for multiuser, distributed operation. In addition, Sun provides a reference JavaSpace implementation that developers can use immediately.

JavaSpaces provides a general messaging mechanism via event queues. Two classes of programs interact with the space: repositories and tools. Design critiquing systems provide support for semantic correctness.

Distributed event queues

Programmers can construct numerous distributed data structures in the space. For our systems, we chose to implement distributed event queues, as shown in Figure 3. Since a system consists of multiple types of data, each data type roughly corresponds to an event queue.

Figure 2. Custom Java tool for Egidio.



Tools communicate state changes by posting and listening to an appropriate event queue. This decoupled communication mechanism lets a tool communicate anonymously with every other tool. In this way, the architecture supports multiuser operation simply, since each user's clients can anonymously connect to the space and communicate state changes. It is also very easy to add new tools to the system, since all you have to do is attach the tool to the desired event queues.

We chose event queues because they preserve the order of event occurrences in the context of the system. They are also easy to implement using collaborating entries in the space. A tail entry tracks the queue's current length, and each element in the queue is an event entry in the space. Figure 4 shows a simplified view of one such event queue.

For a tool to track changes to a particular data type, it should first read the tail entry and note the current length (in this case, n). From then on, it simply has to read event entries $n + 1$, $n + 2$, ... and so on to be notified of future events. Such a scheme is very easy to implement since JavaSpaces supports blocking reads and entry matching by attributes (the event positions, in this case).

To write changes to a particular data type, a tool takes the tail entry (to prevent another participant from writing to the queue at the same time), notes current length n, and writes the tail entry back into the space. The tool then writes a new event entry with the data change information. It also writes a new position (n) into the space for other parties to read.

To prevent possible deadlock (which could happen if the tool dies before writing the tail entry back into the space), the tool can perform all the operations described under a transaction. If the tool dies, the transaction will eventually time out and abort, and the system would restore all entries. JavaSpaces supports such a transaction mechanism by default.

JavaSpace entries also incorporate the concept of expiring leases, which prevent event queue entries from taking up more and more space as the queue grows. Since a queue's purpose is simply to notify observers of events, it is reasonable to give each entry a fairly short lease time. After the lease expires, the system can perform garbage collection for a queue entry as required.

Repositories

Repositories track the state of certain types of data by "listening" to the proper event queues. The system uses the information gathered from the queues to update the local data state in the repository. Periodically, a repository writes the data state to an XML document with a custom DTD (document type definition) for long-term storage.

Using XML has numerous advantages. The data is human-readable with an XML viewer or just a plain-text viewer. XML is standardized, so developing software to work with it takes minimal effort, especially since XML parsers (such as the one at http://www.xmlsoftware.com/parsers/) are available for many programming languages. This permits offline, file-level integration, if necessary.

Holmes repositories use the Holmes Markup Language (HML) format for storage. This format is just XML with a DTD specifically targeting Sherlock's data types. We used HML to develop a quick-and-dirty prototype for a browser that displays selected types of domain information in a hyperbolic tree view (J. Lamping and R. Rao, "A Focus + Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies," *Proc. ACM SigCHI Conf. Human Factors in Computing Systems*, ACM Press, New York, 1995). This view lets users see how closely other entities are related to an entity of interest based on the entity's size and proximity in the view; it also gives a sense of traceability among entities. We parsed the XML data with a standard Java-based XML parser, transforming the desired data into a tree data model that the browser could understand.

Tools

Tools present and change one or more types of data. A tool can connect to the space from anywhere and request

the current state for a certain type of data by posting a state request entry in the space. The corresponding repository will notice the request and post the current state as an entry in the space. The tool then picks up this current state to initialize its own local state. It can then post and monitor changes to this particular type of data on the corresponding event queue.

A tool can also be an aggregate of other tools, launching specific tools only when necessary. In Holmes, the domain definition tool handles all the activities in that phase. It invokes custom views and external applications (and their corresponding adapters) on demand.

With JavaSpaces support for data-level sharing, you can integrate a variety of applications. To be integrated, a tool must

- communicate with the JavaSpace and
- transfer data via specified event queues.

Java-based tools easily satisfy these requirements. For non-Java applications, programmers must develop a tool adapter using Java. Adapters serve two purposes: They handle the JavaSpace and event queue interactions, and map data from the application-specific format to the Java object format that the system expects. Usually, the application and its adapter use the Observer pattern (Eric Gamma and colleagues, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994) to communicate data changes.

The difficulties in building a tool adapter vary with how tight the integration has to be, and what language or environment the developer uses. For very high-level integration (through a shared file, for example), the effort and difficulty are minimal.

For lower levels of integration, the effort depends on the target application. If the application uses a language that can communicate with Java classes, developers can build adapters fairly easily, provided the application's source code is available. Examples of these situations include using JNI (Java Native Interface) with C++ and using TclBlend with Tcl.

For very low-level integration, developers must devote extra effort to interpreting fine-grained communication. For example, integrating a COM component requires an adapter that can understand, interpret, and translate the component's fired events into equivalent JavaSpace event queue interactions.

The use of Java is key because it allows tools on any platform supporting a Java virtual machine to participate in the system. For instance, a user could want to use the Emacs code editor during Holmes' domain framework development phase. A DFD adapter tracks the state of the shared source files by attaching itself to the appropriate event queues in the JavaSpace. When a user edits a file, the adapter asks Emacs to load it via an Emacs Lisp (Elisp) function call. The user can now make code changes within Emacs. Once the user finishes editing and is ready to commit the changes, he invokes another Elisp function to notify the adapter. The adapter will then post the code changes to the appropriate event queue(s).

In this approach, if there is no existing application, you can build an application in Java. Such an application can take advantage of the tightest possible integration with the adapter, which is also written in Java.

Using Holmes as an example, we integrated a variety of applications—Web browsers for HTML viewing, FrontPage Express for HTML editing, Rational Rose for use-case and class diagram editing, and custom Java applications for displaying structured data in tree formats.

Figure 3. Distributed event queues in JavaSpaces provide decoupled communication.

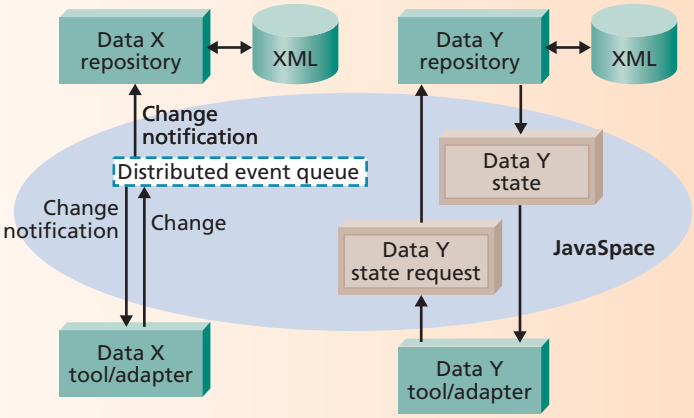


Figure 4. JavaSpace-based event queue.

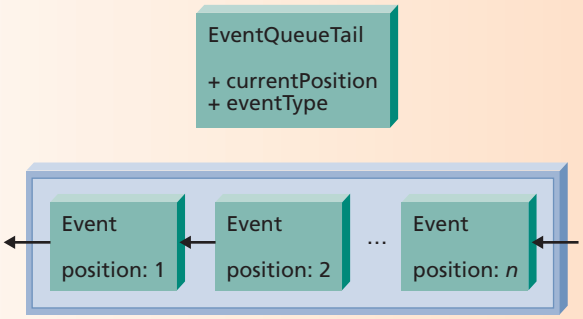


Table 1. How our generic architecture satisfies essential tool requirements.

Requirement	Architectural feature fulfilling this requirement
Multiuser support and distributed operation to maintain data and change consistency	JavaSpaces and distributed event queues
Traceability	Indirectly by integration of hyperbolic browser
Fast development via software package integration	Java-based tool adapters to handle JavaSpace and event queue interactions
Semantic support	Design critics to build into the system just like any other tool

Design critiquing system

Large and complex methodologies can be difficult to fully grasp and work with. So users could likely benefit from some sort of semantic support. Such support could come in the form of a critiquing system, which answers queries about whether a proposed solution is acceptable. Such a system takes a problem description and a proposed solution, and produces a critique of that proposed solution. The critique explains the proposed solution's risks(s) and proposes alternate solutions if possible.

Semantic support can also be built by using the idea of design critics—simple software agents that monitor user actions (data changes) and offer advice when appropriate. Design critics can easily play an important part in our generic architecture.

In our architecture, these critics are simply just another type of tool that participates in the event queue interactions. They detect changes in a specific type of data, evaluate the action against some preset rules, and when necessary, post advice on another dedicated event queue. A separate tool presents the advice to users in a list format.

There are many choices for implementing the critiquing logic. In our implementation, we used the Prolog language because Prolog clauses describe relationships very well. Relationships can describe all our data models; for example, a domain term has a definition, a product has a certain strategy, or a developer has procedural programming skills.

The critiquing system adds substantial value to our proposed architecture. Since the critiquing system is immediately available for any application built on top of the architecture, it can basically critique anything. In addition, we implemented the critics in Prolog, a well-known high-level language. This makes the power of critiquing available to more users.

COMPARING OUR ARCHITECTURE TO OTHER FRAMEWORKS

There are several existing tool integration frameworks. PCTE (Portable Common Tool Environment) is a specification from the ECMA standardization body (http://www.ecma.ch), which is based on the monolithic Stoneman model. It has strong support for data integration, and provides a message service for tool communication as well. However, this service is a low-level, Unix-like mechanism, which makes sophisticated messaging passing more difficult.

Hewlett-Packard's SoftBench is a commercial tool integration platform. The broadcast message server (BMS) routes requests and event notifications between tools (such as a development environment's compilers, debuggers, and editors). The messages follow an abstract tool protocol, each of which has its own set of operations. When the BMS receives a request, it checks to see if any tools are registered to handle that request. If none is registered, the BMS will start a tool to handle this request, if one is available.

Our proposed generic architecture is a federation of tools. Unlike PCTE, it requires no central server—a system can function with just tools communicating changes with each other through the event queues. This makes the system more open and flexible to work with.

The event queue mechanism in our architecture is similar to that of SoftBench. All requests and event notifications are communicated through "software buses." However, because Java-based event queues are object oriented, programmers have more flexibility when they need to extend the communication mechanisms. Specializing an event queue entry does not affect the rest of the system—the JavaSpace and the tools can continue working with the entries as before.

Putting all the concepts together yields the proposed architecture, which satisfies the essential requirements listed earlier. Table 1 summarizes these requirements.

SAMPLE IMPLEMENTATION

Galileo is a POP-based fault tree analysis tool. It lets users edit fault trees graphically, using an integrated Visio component, or textually, using an integrated Microsoft Word component. The application offers online help by using an integrated Internet Explorer component.

To use the generic architecture, we first determined the

granularity of the data that tools had to communicate. Since a fault tree has basic events, gates, and connections, we can naively pick these as the fault tree's building blocks. Event queues report changes at the level of these building blocks. We also wrote a suitable DTD to correspond with the object model chosen for the fault tree.

Next, we determined the flow of objects necessary for tools to communicate fault tree changes. When a user changes a fault tree, the tool posts that change on an *unchecked-element change queue*. A main engine reads this information from the queue and checks for consistency with the rest of the model. If the change is allowed, the tool places it onto a *checked-element change queue*. Other tools would be monitoring this queue, updating their local states as needed. Galileo also needed a repository to track the current state of the fault tree model.

With this information, we can start building the underlying support. Each type of fault tree building block becomes a subclass of a JavaSpace entry. We then develop classes for initializing and interacting with the two main event queues. If the main consistency engine is implemented in Java, we simply use the classes previously developed. Otherwise, we write a consistency engine adapter that handles the queues and entries, and also integrates with the concrete consistency engine. We also need the repository, which will periodically spawn a new thread to write the data to XML.

Next, we have to integrate Visio and Word; each requires its own tool adapter. As an example, the Visio tool adapter has several tasks:

- Initialize, customize, and present a Visio component through COM using JNI.
- Request the fault tree's current state on start up, passing this information to Visio.

Resources

Package-oriented programming

- "Package-Oriented Programming of Engineering Tools," K.J. Sullivan and colleagues, *Proc. 19th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 1997.
- "Multiple Mass-Market Applications as Components," D. Coppit and K.J. Sullivan, *Proc. 19th Int'l Conf. Software Eng.*, IEEE CS Press, Los Alamitos, Calif., 2000.

POP-based tool

- Holmes: *A Domain Oriented Approach to Software Production*, P. Predonzani, G. Succi, and T. Vernazza, Artech House Inc., Norwood, Mass., 2000.
- Egidio: "Business Process Modeling with Objects, Costs, and Human Resources," G. Succi, P. Predonzani, and T. Vernazza, in *System Modeling for Business Process Improvement*, D. Bustard, P. Kawalek, and M. Norris, eds.; Artech House Inc., Norwood, Mass., 2000.
- ISI Visual Design Editor: "The ISI Visual Design Editor Generator," N.M. Goldman and R.M. Balzer, *Proc. IEEE Symp. Visual Languages*, IEEE CS Press, Los Alamitos, Calif., 1999.

Other integration environments

- "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," M.R. Cagan, *HP Journal*, June 1990, pp 36-47.
- "An Overview of PCTE: A Basis for a Portable Common Tool Environment," F. Long and E. Morris, Tech. Report CMU/SEI-93-TR-1, Carnegie Mellon Univ., 1993.

Critiquing systems

- "A Critic for LISP," G. Fischer, *Proc. 10th Int'l Joint Conf. Artificial Intelligence*, Morgan Kaufmann Publishers, San Francisco, 1987.
- "Embedding Computer-Based Critics in the Contexts of Design," G. Fischer and colleagues, *Proc. ACM InterCHI*, ACM Press, New York, 1993.
- "Critiquing: Effective Decision Support in Time-Critical Domains," A. Gertner, Tech. Report MS-CIS-94-35, Univ. of Pennsylvania, 1994.
- *Expert Critiquing Systems*, Perry Miller, Springer-Verlag, New York, 1986.
- "eMMaC: Knowledge-Based Color Critiquing Support for Novice Multimedia Authors," K. Nakakoji and colleagues, *Proc. ACM Multimedia*, ACM Press, New York, 1995.
- "Design Critiquing Systems," J. Robbins, Tech. Report UCI-98-41, Univ. of California, Irvine, 1998.

JavaSpaces

- *JavaSpaces: Principles, Patterns, and Practice,* E. Freeman, Addison-Wesley, Reading, Mass., 1999.
- "JavaSpaces Specification, Revision 1.0 Beta," Sun Microsystems, Palo Alto, Calif., 1998; http://java.sun.com/products/javaspaces/specs/.

- Monitor the checked-element change queue for data changes. When change events occur, the adapter transforms the information into COM calls to Visio.
- Write changes from Visio to the event queue. When the Visio component commits changes, the adapter transforms these changes into JavaSpace entries that go into the unchecked-element change queue.

Adapting and assembling both components into the architecture give us a fully functional system. To integrate each additional tool, we follow a procedure similar to that for Visio.

Our generic architecture satisfies the additional requirements we deem important for larger, more complex software engineering activities, but at a cost. Because application integration occurs at the data level, it is much looser than what is possible with Microsoft COM. COM-compatible applications can be integrated on a procedural-call or event level. Furthermore, COM-compatible applications are built so that some customization can take place (to hide irrelevant program features, for instance).

On the other hand, loosening the integration level has the advantage of being more general. Relying on COM means that you can integrate only certain applications running on specific platforms into the system (although DCOM is now available on some other platforms; see http://www.softwareag.com/entirex/download/free_download.htm). Using data-level integration means that more applications for different platforms can be integrated, as long as a tool adapter is available. The effort involved in writing the adapter is much less than that of writing a production-quality tool from scratch.

There are also other issues to consider when integrating applications in this generic manner. For example, what happens when changes take place while a user interacts with an application that cannot respond to events? The tool adapter has to implement a suitable strategy for resolving the resulting differences.

In the end, our experiences show that the strength of this architecture lies in its simplicity and ability to work with multiple users and quickly integrate a wide variety of applications. It is definitely not perfect, but we present it as a first step toward a more general package-oriented architecture to encourage further research in this area. ■

Giancarlo Succi *is a professor in the Department of Electrical and Computer Engineering, University of Alberta. Contact him at giancarlo.succi@ee.ualberta.ca.*

Witold Pedrycz *is a professor and Director of Computer Engineering in the Department of Electrical and Computer Engineering, University of Alberta. Contact him at pedrycz@ee.ualberta.ca.*

Eric Liu *completed his MSc at the Department of Electrical and Computer Engineering, University of Calgary, and is currently a software engineer at ThoughtWorks. Contact him at eliu@thoughtworks.com.*

Jason Yip *is a software engineer at ThoughtWorks. Contact him at jcyip@thoughtworks.com.*

This research has been partly supported by the Canadian Natural Sciences and Engineering Research Council, the Government of Alberta, the University of Alberta, the University of Calgary, and the Alberta Software Engineering Research Consortium. We also thank Milorad Stefanovic and Raymond Wong for reviewing early drafts.