



Visualizing Flow Diagrams in WebSphere Studio Using SHriMP Views

Derek Rayside^{*,†} and Marin Litoiu

IBM Centre for Advanced Studies, IBM Toronto Laboratory
E-mail: drayside@acm.org
E-mail: marin@ca.ibm.com

Margaret-Anne Storey, Casey Best
and Robert Lintern

Department of Computer Science, University of Victoria
E-mail: mstorey@csr.uvic.ca
E-mail: cbest@csr.uvic.ca
E-mail: rlintern@csr.uvic.ca

Abstract. This paper describes the integration of an information visualization tool, called SHriMP Views, with IBM WebSphere Studio Application Developer Integration Edition, which was developed with Eclipse technology. Although SHriMP was originally developed for visualizing programs, it is content-independent. We have re-targeted SHriMP for visualizing flow diagrams. Flow diagrams, as supported by WebSphere Studio Application Developer Integration Edition, can be hierarchically composed, thus leveraging the key features of SHriMP that allow a user to easily navigate hierarchically composed information spaces. We discuss the differences between programs and flow diagrams, in terms of their semantics and their visual representation. We also report on the main technical challenges we faced, due to the different widget sets used by SHriMP (Swing/AWT) and Eclipse (SWT).

Key Words. integration, software engineering, software visualization, system modeling, flow diagrams, MOF, XMI

1. Introduction

Leveraging industrial tool infrastructure can significantly reduce the amount of effort required to develop tool prototypes. At CASCON'99, Martin (1999) reported that it took him only two weeks to build a complete C++ fact extractor for Rigi (Müller and Klashinsky, 1988) based on IBM® VisualAge® C++, in contrast to the protracted effort that had been required to build the previous fact extractor (Martin, 1999). Moreover, his new fact extractor was more robust, easier to understand, and had 75% less code

(Martin, 1999). Leveraging industrial tool infrastructure allows researchers to focus on research.

Eclipse presents some exciting opportunities for researchers who build software engineering tools. At its core, Eclipse is a platform of common tool infrastructure, including: an XML-based plug-in architecture for interoperability; frameworks for structured text and graphics editing; a common debugging infrastructure; and a complete Java™ development environment. Furthermore, because the Eclipse core is open source, it can be modified by researchers and freely downloaded by research tool users. Leveraging the Eclipse infrastructure can significantly reduce the amount of effort required to develop research tools.

The new IBM offering, *WebSphere Studio Application Developer Integration Edition* (IBM, <http://www.ibm.com/software/ad/studiointegration>)—WebSphere Studio for short—presents new opportunities for researchers interested in leveraging standard technologies. WebSphere Studio is based on open-source technology from the Eclipse Project (www.eclipse.org). This paper reports on the integration of WebSphere Studio with the SHriMP (Simple Hierarchical Multi-Perspective) information visualization

*To whom correspondence should be addressed.

†Present address: Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, Room 532, Cambridge, MA, 02139-3578, USA.

tool from the University of Victoria. Specifically, we want to use SHriMP to visualize *flow diagrams*.

Flow diagrams, as implemented by WebSphere Studio Application Developer Integration Edition, are used in an e-business project to model the dynamic aspects of a system, such as the main activities and the movement of information in a business process. The main motivation for our project is to apply SHriMP's advanced visualization features to flow diagrams, which may be hierarchically composed.

SHriMP was originally developed for program visualization; therefore, some minor modifications were required to adapt it to this new domain. One of our research objectives is to transfer knowledge from program understanding to flow understanding—and in the process to see if we can gain any new insights to improve program visualization.

Fig. 1 shows the architecture of our project and the three main foci of our integration efforts. The column on the left represents SHriMP, and the column on the right represents the flow diagram editor; both of these tools are presented within the context of the Eclipse infrastructure. Project-application-specific components are shown in a darker shade of gray, and project-independent reusable components are shown in a lighter shade of gray. The three labeled, dashed lines represent the points of interaction between SHriMP and Eclipse: Line 1 represents file-based data exchange between the RSF (Rigi Standard Form) format used by SHriMP and the XML-based format used by the flow

diagram tool in Eclipse; line 2 represents API-based data exchange between the flow diagram tool and SHriMP; line 3 represents the control and user interface integration between the tools.

The MOF (Meta-Object Facility) frameworks for working with XMI-encoded data, such as flow diagrams, made the mechanics of the data integration very easy. MOF and XMI are described in the next section. Section 3 introduces flows and flow diagrams in detail. Section 4 discusses the data and domain integration aspects of rendering flow diagrams in SHriMP: these are largely to do with terminals.

We faced some interesting challenges in the control integration because SHriMP and Eclipse are based on different widget toolkits: Swing/AWT (Abstract Window Toolkit) and SWT (Standard Widget Toolkit), respectively. SWT is described in more detail in the next section. We used an experimental mechanism for integrating tools based on these different widget toolkits, and report our experience for the research community in Section 5 (commercial developers should use the standard control integration mechanisms, which are also described briefly). Section 6 concludes the paper.

2. WebSphere Studio

In this paper we use the term *WebSphere Studio* to refer to *WebSphere Studio Application Developer*

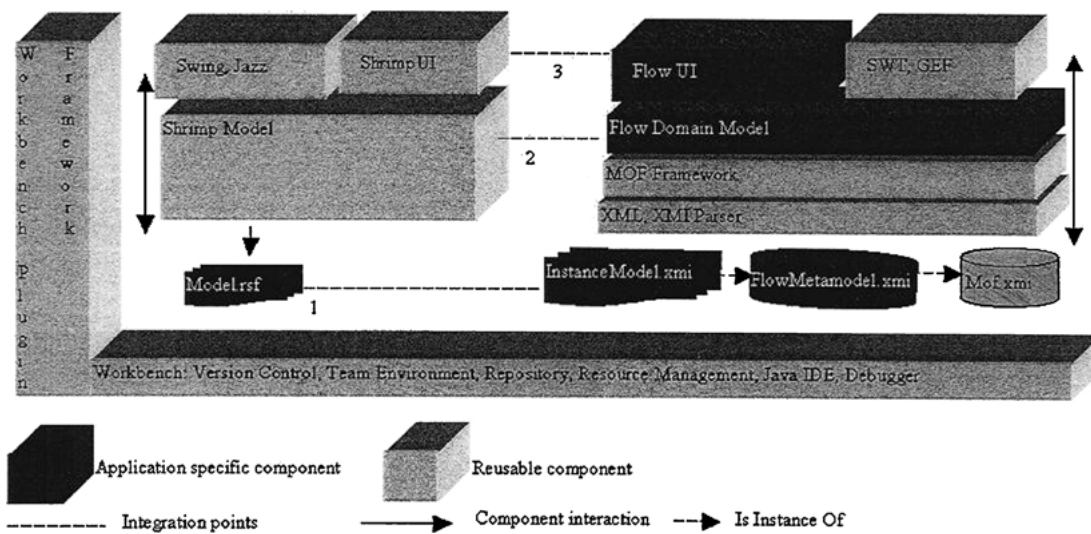


Fig. 1. Eclipse and SHriMP architecture.

Integration Edition. IBM markets a number of WebSphere Studio products, most of which are based on the open-source Eclipse technology. This section describes some of the core functionality of Eclipse, as well as some of the extra technology in WebSphere Studio Application Developer Integration Edition that we used for this project.

2.1. Eclipse

The core of Eclipse is an infrastructure for building integrated development tools. The main integration mechanism is an XML-based mechanism for defining *plug-ins*. Plug-ins contribute functionality by hooking into *extension points* defined by other plug-ins, and may also define new extension points. The Eclipse core provides all of the infrastructure necessary for the dynamic discovery, linking, and execution of plug-ins.

Eclipse also includes infrastructure for resource management, version control, and debugging. One of the other useful frameworks included in Eclipse is SEF (source editing framework). SEF can be used to build customized editors for various forms of structured text, such as Java programs and XML documents.

Eclipse comes with three standard sets of plug-ins: Java development tools, Web development tools, and plug-in development tools. Each set is built with the Eclipse core infrastructure, and interoperates with the other sets. The Java development tools in Eclipse will eventually supersede the VisualAge for Java functionality (IBM, <http://www7.software.ibm.com/vad.nsf/data/document2020>).

2.1.1. Standard Widget Toolkit (SWT). Standard Widget Toolkit (SWT) is the operating-system-independent GUI widget toolkit used by the Eclipse platform. SWT is analogous to Swing/AWT: the difference is in the implementation strategy. SWT uses native widgets wherever possible for three main advantages: *performance*, *look-and-feel*, and *debugging*. If no native widget is available on a particular platform, then a Java version is implemented for that platform. For example, Microsoft® Windows® contains a native tree widget and Motif does not: on a Windows platform, SWT uses the native widget, and a Java equivalent is provided in the Motif environment. This is, of course, all transparent to the Java application programmer.

Consequently, SWT-based applications always look, feel and perform like native applications, and any problems with the widgets can be directly replicated in C code (which makes debugging easier for the SWT

implementor). Another benefit of using native widgets is that SWT-based applications can interact with platform-specific features, such as ActiveX controls in Windows.

AWT follows a ‘lowest common denominator’ strategy: it provides only those widgets that are available on all platforms. Swing compensates for this by building higher-level Java widgets on top of AWT’s lowest common denominator. So, there is only one implementation of Swing that works across all Java-supported windowing environments. However, these Swing widgets will always look and feel distinct from their native counterparts and perform differently.

The Eclipse Web site contains white papers on SWT (Northover, 2001), integrating with ActiveX and OLE on Windows (Irvine, 2001), and creating new widgets with SWT (Northover and MacLeod, 2001).

While SWT provides Eclipse with superior performance and look-and-feel, it presents an integration challenge to us because SHriMP is based on Swing/AWT. Ideally, we would have the resources to rewrite SHriMP in SWT, but this is not feasible given SHriMP’s advanced visual nature and the limited resources of university research. Consequently, we experimented with an undocumented mechanism for integrating Swing-based program with SWT-based programs, as discussed in Section 5. Commercial tool developers, and those starting from scratch, should use one of the other options, also discussed in Section 5.

2.2. WebSphere studio

On top of Eclipse plug-ins and architecture, WebSphere Studio Application Developer Integration Edition adds more plug-ins and frameworks, such as:

- MOF and XML Tools
- GEF: Graphical Editor Framework
- Web services tools
- Flow diagram editors

2.2.1. Meta-Object Facility (MOF). The *Meta-Object Facility* (MOF) (OMG, 2000) is an Object Management Group (OMG) standard for describing metamodels, models, and instance data. XMI (OMG, 2000) is an associated OMG standard for serializing MOF in XML.

Flow diagrams can be modelled with MOF and serialized in XMI, as previously reported by Litoiu, Starkey, and Schmidt (2001). We leverage the WebSphere


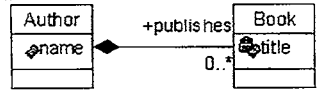

UML	MOF	XMI
	M3: Level 3 (MOF-Core)	<pre><mof:Class ... <mof:Attribute ... </mof:Class ... <mof:Association ...</pre>
	M2: Level 2 (Application Model)	<pre><mof:Class id="Author" ... <Association id="publishes" ... </mof:Class></pre>
	M1: Level 1 (Application Instance Model)	<pre><Author name="Plato"> <Author.Publishes> <Book title="The Republic"/> </Author.Publishes> </Author></pre>

Fig. 2. The relation between MOF, UML, and XMI.

Studio Application Developer Integration Edition infrastructure for working with MOF to take advantage of their results.

Fig. 2 shows the correlation among the expressions of model information in UML, MOF, and XMI. The first level of meta-modeling is M3 of MOF-core. MOF-core is a subset of UML and contains definitions of *Class*, *Attribute*, *Association*, and so on Booch, Rumbaugh, and Jacobson (1999), OMG (2000). The second level of meta-modeling is M2—the application domain model. This is the level on which most people operate when they build ordinary class diagrams. Any M2 model is an instance of an M3 level, being constructed with instances of *Class*, *Attribute*, *Association*, and so on.

As an example, **Author** and **Book** may be considered as M2 instances of *Class*, and **name** and **title** may be considered as instances of *Attribute*. **Author** and **Book** are related to each other by a relationship called *Aggregation* that is an instance of M3 *Association*. A meta-model from the M2 level can have many instances in the M1 level. For example, **Plato** and **Shakespeare** are instances of *Author*; **The Republic** and **As You Like It** are instances of *Book*.

The *instance-of* relation distinguishes the three levels in MOF: each level instantiates the level above it and, conversely, is instantiated by the level below it. One of the main advantages, for integration, of having an explicit meta-model is that an API written for the meta-model can be used to work with any model that instantiates that meta-model and any data that instantiates those models.

Use of MOF in our project. Fig. 1 illustrates the role that MOF and XMI play in our project. *FlowMetaModel.xmi* represents our application domain metamodel (taken from Litoiu, Starkey, and Schmidt, 2001). The WebSphere Studio Application Developer Integration Edition infrastructure can automatically derive a Java API for working with models that instantiate this meta-model. This API is named. *Flow Domain Model* in Fig. 1, and can be used to work with flow diagrams encoded in XMI. We can use this automatically generated API without knowledge of the particular details of the XML encoding.

3. Flow Diagrams

There are many specific kinds of technical flow diagrams, such as: Petri nets, data flow diagrams (Ross, 1977), statecharts (Harel and Gery, 1997), process dependency diagrams (Martin, 1989; Martin and Odell, 1992), UML activity diagrams (Booch, 1999), and workflows (Leyman and Roller, 1997; Casatti et al., 2000). IBM produces a number of middleware products that also have specific kinds of flow diagrams associated with them. In these products, the programmer, business analyst or system integrator, draws flow diagrams that are executed by associated runtime environments. The flows in these products are classified as *nano-flows*, *micro-flows* and *macro-flows* (also known as *work-flows*) (Litoiu, Starkey, and Schmidt, 2001).

Macro-flows use large-scale granular activities and are deployed by large-scale software components such

as applications. Workflow products that model and implement business processes are examples of macro-flows. Nano-flows are flows of data and control that take place inside an object or object method. In IBM products, nano-flows model and implement the wiring of Java classes into programs that access legacy applications. Nano-flow, micro-flow, and macro-flow diagrams can be modeled with an extension of MOF, called Flow Composition Model (FCM), as was previously reported by Litoiu, Starkey, and Schmidt (2001). FCM is in the process of becoming an OMG standard (Object Management Group). MOF and FCM are typically serialized in the XML-based XMI format (OMG, 2000). As was discussed in the previous section, and shown in Fig. 1, WebSphere Studio Application Developer Integration Edition provides substantial infrastructure for working with MOF models encoded in XMI.

The WebSphereMQ Integrator[®] (IBM, <http://www-3.ibm.com/software/ts/mqseries/integrator/>) product (formerly known as Message Queue System Integrator[®], or MQIntegrator[®]) provides the analyst with tools to draw message-flow diagrams that describe the movement of data between information systems. The associated runtime environment executes these diagrams: that is, combines, transforms, and delivers the messages. WebSphere MQIntegrator is typically used to integrate information systems when large corporations merge. Much of IBM's flow-related middleware is developed in Toronto.

3.1. An example flow diagram

Fig. 4 describes a (simplified) business process for processing a credit request, involving activities such as **Risk:Assess**, **Risk:Reassess** and, eventually, **Send:Money** or **Send:Rejection**.

Fig. 4(c) shows the main **Credit Approval** process, which is composed of two sub-processes: **Risk** (Fig. 4(a)) and **Send** (Fig. 4(b)). Fig. 5 shows the credit approval section (i.e., Fig. 4(c)) as it looks in WebSphere MQIntegrator's Flow Composition Builder. Notice that the two sub-flows (i.e., Fig. 4(a) and (b)) are not immediately evident: the user must open up separate editor tabs for each sub-flow. This obfuscates the hierarchical composition of the flows, and makes it more difficult for the user to understand the diagrams. We want to use SHriMP's advanced features for visualizing these kinds of hierarchical composition relationships.

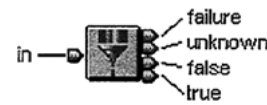


Fig. 3. A filter node from Message-Queue system integrator.

3.2. Terminals

One of the important characteristics of nodes in flow diagrams is that they have specific *terminals* on which arcs may be attached to them. Which terminal an arc is attached to is fundamental to the meaning of the diagram, as discussed below, and so these terminals must be explicitly represented in some fashion.

A simple example. Fig. 3 shows an example of a 'filter' node from the MQIntegrator (IBM, <http://www-4.ibm.com/software/ts/mqseries/>) flow editor with its terminals labeled. A filter node is a kind of primitive node that is analogous to a simple conditional test in an imperative programming language. Filter nodes have one input terminal and four output terminals: **failure**, **unknown**, **true** and **false**. When a message arrives at a filter node, its condition is evaluated, and a message is sent from one of the output terminals according to the result of the evaluation.

Other kinds of nodes may accept multiple inputs and send multiple outputs.

Terminals have distinct identity. Terminals have distinct identity: that is, they cannot be identified with arcs as arrowheads can, nor can they be subsumed by the nodes they are attached to. Terminals are different from arrowheads in two ways: terminals are still present on a node even if there are no arcs connected to them; and multiple arcs may be connected to a single terminal.

3.3. Flows and programs

The relationship between flows and programs has been approached from many angles. Researchers in program analysis, compilers and reverse engineering often use controlflow graphs and data-flow graphs to describe certain aspects of programs. Böhm and Jacopini (1966) consider the relation of flow diagrams to the fundamental notion of computation in their classic paper *Flow Diagrams, Turing Machines and Languages with only Two Formation Rules*.

Litoiu, Starkey, and Schmidt (2001) consider that there is an analogy between flows and classes (in the object-oriented programming sense of the word) when modeling flows with MOF. From their perspective, the

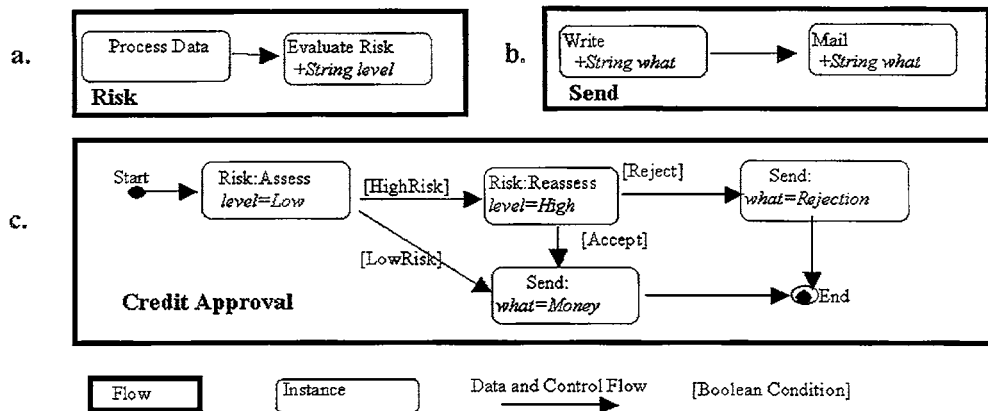


Fig. 4. Expanded flow diagram of a credit approval process.

Risk:Assess and **Risk:Reassess** boxes in Fig. 4(c) represent *instances* of the flow **Risk** shown in Fig. 4(a).

We are interested in the relationship between flows and programs because SHriMP was originally developed for program visualization and we are now using it for flow visualization.

4. Visualizing Flow Diagrams with SHriMP

This section describes how SHriMP's advanced visualization features are useful for flow diagrams, describes some of the detailed visual characteristics of flow diagrams, and outlines how we added terminals to SHriMP (the main change required for visualizing flow diagrams). Fig. 7 shows how SHriMP renders the credit approval example that was used in Figs. 4 and 5. We will discuss SHriMP's features with respect to this example.

SHriMP. The SHriMP (Simple Hierarchical Multi-Perspective) visualization technique was designed to enhance how people browse and explore complex information spaces. It was originally designed to enhance how programmers understand programs (Storey, Müller, and Wong, 1996; Wu and Storey, 2000). SHriMP presents a nested graph view of a software architecture. Program source code and documentation are presented by embedding marked up text fragments within the nodes of a nested graph. Finer connections among these fragments are represented by a network that is navigated using a hypertext-link-following metaphor. SHriMP combines this hypertext metaphor

with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user.

Smooth zooming. SHriMP's most visually striking feature is its smooth zooming, which enables the analyst to 'zoom-in' to the detail, or 'zoom-out' to the big picture in an easy and intuitive manner. Fig. 6 attempts to convey this dynamic nature with a simple example diagram: the top-left portion shows the high-level flow diagram; the bottom-left portion opens up the sub-flow that it is composed of; and the right side zooms into the sub-flow. SHriMP's zooming feature is based on the JAZZ zooming library from the University of Maryland (www.cs.umd.edu/hcil/jazz.)

By contrast, if using the Flow Composition Builder pictured in Fig. 5, the user would have to open up a separate window to view the sub-flow, and its relationship to the super-flow would thereby be obfuscated.

Nested interchangeable views. One of the key features of SHriMP is its ability to show *nested interchangeable views*. This feature means that there are different views possible for nodes at different levels in the hierarchy. This feature is illustrated in Fig. 7, which shows nodes in three different views; *open* (i.e., showing their subnodes), *closed* (i.e., showing their own icon), and *property sheet*. The open and closed views are common across all domains. Other domains may also have other views: for example, when visualizing Java programs SHriMP supports both source code and Java-doc views. The user can change the currently displayed view of each node on an individual basis.

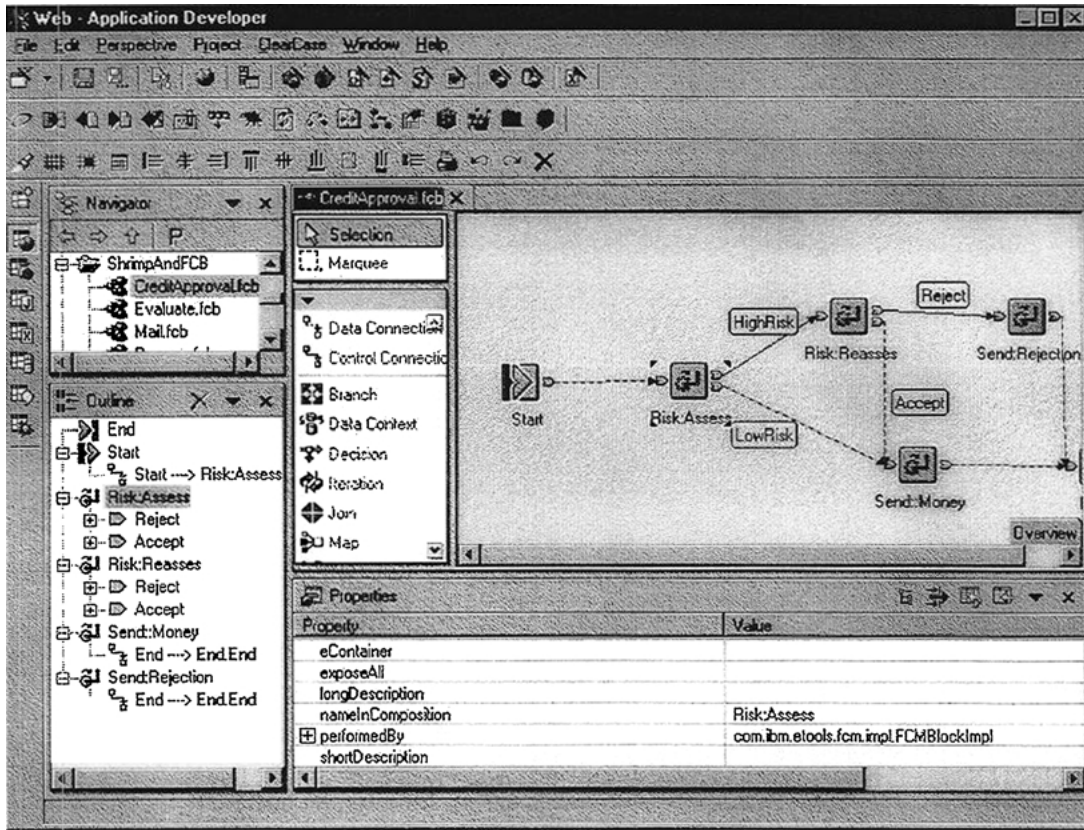


Fig. 5. CreditApproval.fcb in flow composition builder inside WebSphere studio.

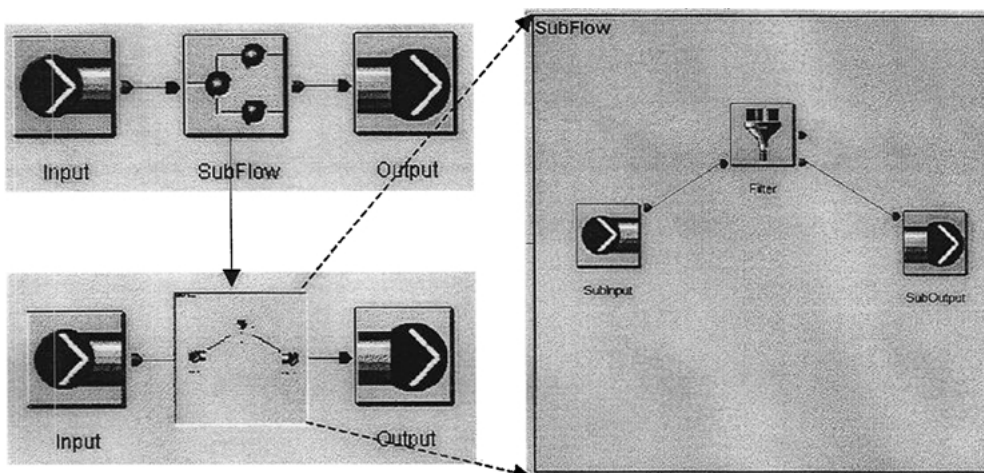


Fig. 6. SHriMP zooming in on a hierarchically composed flow diagram.

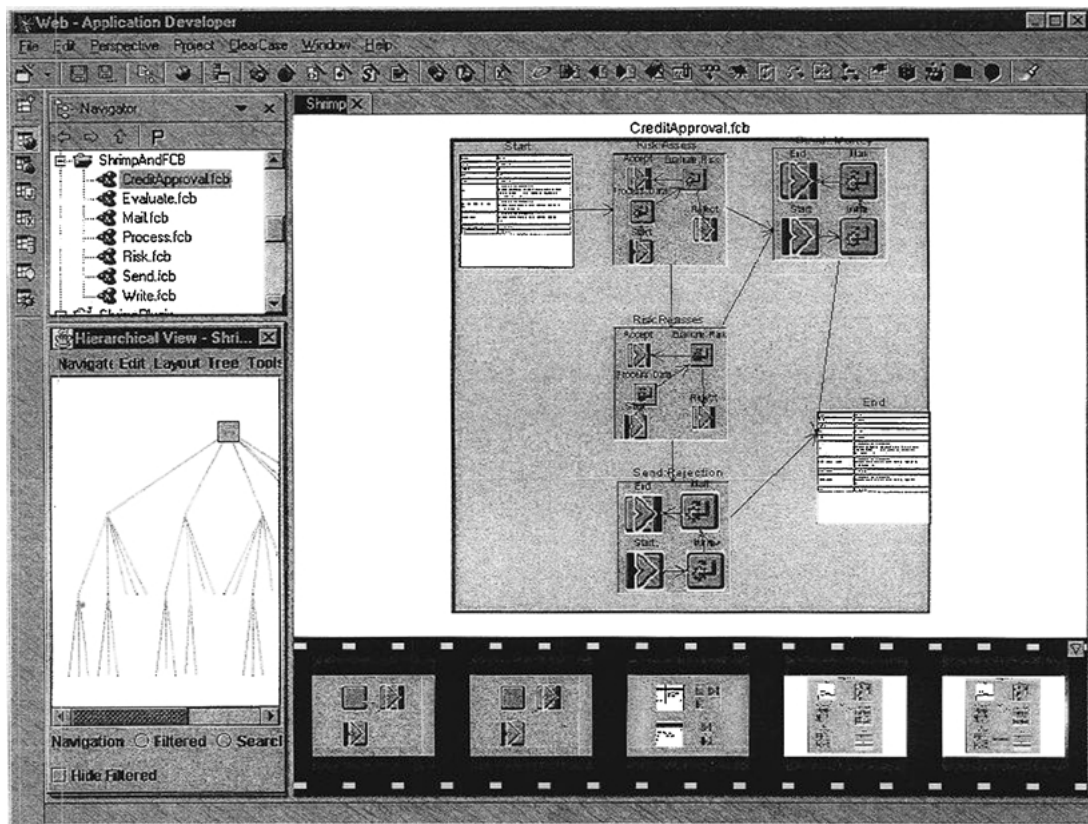


Fig. 7. *CreditApproval.fcb* in SHriMP inside WebSphere studio.

Hierarchy view. The left side of Fig. 7 shows a hierarchical overview (vertical slice) of the nested graph layout of the flow diagram shown on the right side. This view serves purposes. It provides a “gestalt” picture of the depth of the hierarchy in the flow diagram and the approximate number of instances it contains. In addition, it acts as a powerful navigation aid by increasing the size of the node that is currently being visited (in other words it provides a “you are here” map). Furthermore, since the user can search for nodes in a flow diagram, the hierarchical overview can be used to highlight the results from a search query (thus providing important clues on the location of the results in the overall flow diagram). Also, it can be used to show which nodes have been filtered. The user can either manually filter selected nodes, or filter nodes of a certain type. The user can use the filtered hierarchical overview to “unfilter” nodes using direct manipulation. The user can pan, zoom or select any node in this view and zoom to it in the main SHriMP view.

Filmstrip. The bottom of Fig. 7 shows a “filmstrip” of saved “snapshots” or “bookmarks” that capture interesting views to be reloaded by the user as desired. Each snapshot records all of the layout and filter information as well as which node was zoomed in the view. The thumbnail sized images provide an important recognition aid for users as they explore a hierarchical flow diagram. The user can (and indeed should) annotate each of the snapshots. A tooltip reveals each of the annotations as the user moves the mouse over snapshots in the filmstrip.

4.1. Adding terminals to SHriMP

There were two main aspects to adding terminals to SHriMP: the visual aspects and the data-modeling concerns.

4.1.1. Visual aspects. The main visual aspects of terminals are:

- Arcs are connected to terminals.
- Terminals ‘belong to’ nodes.
- Some terminals have no arcs attached.
- Some terminals have >1 arc attached.
- Terminals are placed to reduce arc length.
- Terminals are represented by images.
- Terminals are colour-coded.

Some of these aspects are illustrated in Fig. 6. In the old Message Queue System Integrator flow diagram editor, terminals are all visually identical, and so they are identified by their position with respect to the node. This limits the layout possibilities, and so in SHriMP we have decided to visually distinguish the terminals such that the arcs can be drawn according to a shortest-path heuristic.

4.1.2. Data-modeling concerns. SHriMP uses a graph-based data model, similar to the one embodied by RSF (Rigi Standard Form, Müller and Klashinsky, 1988) or GXL (Graph Exchange Language, Holt et al., 2000) (the two exchange formats that SHriMP uses). This kind of data model is not particularly well suited to the concept of terminals, because terminals are a new kind of first-class entity that mediates between arcs and nodes.

We considered two main possibilities for representing terminals within SHriMP’s existing data model: as *distinct nodes*, or as *attributes* on existing nodes and arcs. In either approach, the view (in the sense of MVC (Model-View-Controller), see below) must be modified to recognize and render the terminal information. We tried both approaches and found that the attribute approach was better.

Representing terminals as attributes on existing nodes and arcs requires adding two attributes to every arc (source terminal and target terminal), as well as k attributes to every node, where k is the number of terminals on that node. The attributes on nodes are necessary to ensure that terminals with no arcs attached to them are represented. In practice we used $2k$ attributes on every node: the second attribute indicates the image to use for the terminal.

Another possibility, which we did not try, would be to create an entirely new data model for SHriMP that explicitly incorporated the notion of terminals. SHriMP’s architecture does provide a generic data model that can be specialized for each new domain. However, this would have involved significantly more effort. Integration with SHriMP is

further discussed in Best, Storey, and Michaud (2002).

Model-view-controller. SHriMP is designed around the well-known *model-view-controller* (MVC) architecture. The discussion above describes the changes we made to the *model*. However, to render the diagram properly the *view* has to interpret the attributes pertaining to terminals in a special fashion. Note that other attributes are simply regarded as arbitrary strings associated with the nodes or arcs, and have no special implications for the view. While it was not difficult to change the view to interpret the terminal-related attributes, it obviously required introducing special cases, which always complicates code over the long-term. This is one of the reasons that designing a new data model to explicitly incorporate terminals is an approach that we will likely follow in the future.

Two views. Typically, the *view*, in the MVC sense of the word, consists of the set of objects manipulated by the GUI. SHriMP’s advanced zooming features, however, require that it have two views: one for Swing (i.e., the ‘normal’ view), and one for the zooming library JAZZ (i.e., the ‘extra’ view). In other words, both Swing and JAZZ maintain their own sets of objects representing the visual elements on the screen.

The distinction between these two views is important for a variety of reasons; in our particular context, each view deals with terminals differently. In the Swing view, terminals are first-class entities: that is, there is a Swing object associated with every terminal. This Swing object is necessary to render the terminal. However, terminals are not represented in the JAZZ view because they zoom with the nodes that they are attached to.

Terminals are the only things that are treated differently by the Swing and JAZZ views: nodes and arcs are explicitly represented in both views. This subtle issue was one of the most important factors that enabled us to get the terminals to display and zoom properly.

5. Control and User Interface Integration

The Eclipse platform is designed around a ‘plug-in’ architecture in order to achieve tight control and user interface integration between tools. All of the standard tools that come with Eclipse, such as the Java development environment, are plug-ins. The core of the Eclipse platform is a mechanism for discovering, registering

and integrating plug-ins. The details of integration are specified in a `plugin.xml` file, which describes how one tool ‘plugs-in’ to the extension points of another tool, and what libraries it depends on, and so forth. How to build new plug-ins is well documented (see, for example, Amsden, 2001).

The most seamless user-interface integration is achieved for plug-ins that are written with SWT: these plug-ins seamlessly merge with the Eclipse UI in the same way that the standard plug-ins do. Such plug-ins also run in the same VM (virtual machine) that Eclipse does, and can take advantage of platform-specific features, such as using ActiveX controls on Windows (Irvine, 2001) (of course, this limits the portability of the plug-in).

SHriMP, however, is written with Swing and JAZZ (a third-party visual zooming library, www.cs.umd.edu/hcil/jazz/). Ideally, we would rewrite SHriMP with SWT, but this is not feasible at the present given SHriMP’s sophisticated visual nature and the limited resources of university research. So, we experimented with an undocumented mechanism for integrating Swing applications into Eclipse: this approach is acceptable for our research prototyping purposes, but is not recommended for commercial tool builders, or for researchers starting from scratch.

We were able to pursue the user interface integration aspect of the project independently of the data integration aspect because both SHriMP and Eclipse are fairly content-independent.

5.1. Options for integrating with SWT

Eclipse can be integrated with tools that are based on four different user interface technologies (Adams, 2001): SWT, Swing, OLE (object linking and embedding), and ‘other’ (i.e., launching an external tool in a separate window and process). SWT is the tightest interface integration while arbitrary external tool integration is the loosest form of integration. More details on the various integration mechanisms are given below.

SWT-based tools. Java tools based on the SWT can run *in-place* within Eclipse: that is, they are visually indistinguishable from the built-in tools. These tools also run *in-process*: that is, within the same VM and class libraries as Eclipse itself. These tools can achieve seamless functional integration if the Eclipse API’s are used. This is the tightest and most seamless form of interface integration.

In-place Java tools can also use ActiveX controls (on Windows platforms only). Eclipse provides a mechanism for using ActiveX controls from within the Java code (Irvine, 2001).

External tools. Any tool written in any language can be launched from Eclipse with its own separate process. These tools open as separate windows and are not visually integrated with Eclipse. The platform provides automatic launching of external editors based on file type associations, which may be obtained from the underlying operating system or specified by the user. This is the loosest form of integration.

OLE-based tools. On the Windows platform, OLE-based tools can be run in-place. These tools appear to be tightly integrated with Eclipse, but functionally they are the same as any external-launch tool. For example, the Eclipse help system on Windows uses Microsoft Internet Explorer in this way.

Swing-based tools. Swing-based tools can be integrated with the Eclipse user interface in two ways: in a separate window or within Eclipse. If the Swing-based tool is launched in a separate window, it appears to the user much the same as any external tool would. One important technical difference is that arbitrary external tools are launched in a separate process, whereas Swing-based tools can run within the same VM and class libraries as Eclipse does.

There is an experimental mechanism for running Swing-based tools in-place within Eclipse on Windows platforms, but its use is not officially supported at this time. We have experimented with this mechanism and report on our experience here for the research community. Commercial tool developers, or anyone else requiring officially supported functionality, should run Swing-based tools in a separate window (or, better yet, rewrite the interface in SWT). The main restrictions on this experimental mechanism are (Irvine, 2001):

- It only works on Windows.
- It does not work with JDK 1.4.
- The API is internal and subject to change.
- There may be possible deadlock problems.

The experimental mechanism is basically just the `new_Panel` method in the class: `org.eclipse.swt.internal.awt.win32.SWT_AWT`. The `new_Panel` method has one parameter, an SWT Composite object, and it returns an AWT Panel object. The AWT Panel is contained within the SWT Composite, and is constructed using a

WEmbeddedFrame from Sun's internal package `sun.awt.windows`. Sun has changed this internal code in JDK 1.4, and so this undocumented Eclipse mechanism does not work with that version.

5.2. Swing/SWT interaction

The main problem that we had with Swing/SWT interaction is with the interaction of the event queues—which we resolved, as described below. We previously reported (Rayside et al., 2001) some problems with keyboard events and complicated repaint operations with earlier versions of Eclipse, but these have since been resolved by the Eclipse team. We sometimes experience freezes, and suspect that this is due to some threading problem between the two GUI systems.

The execution of any widget toolkit is controlled by an event queue: the operating system adds events to the queue as the user manipulates the machine (such as moves the mouse or presses a key, etc.); the dispatching loop pops events off the front of the queue and executes them. Sometimes it is important to ensure that certain events happen in a certain order, or that some computation is performed before or after certain interface events. For example, an incremental progress bar requires that the user interface is periodically updated while a computation proceeds.

Swing provides two methods in the `SwingUtilities` class for the programmer to specify such temporal ordering: `invokeLater()` and `invokeAndWait()`. Both methods take a parameter of type `Runnable` that contains the code to be executed in the event dispatching thread. `invokeLater()` places the `Runnable` at the end of the event queue and returns immediately (i.e., it makes an asynchronous request). `invokeAndWait()` is a synchronous request: it blocks the requesting thread until the `Runnable` has been executed in the event dispatching thread. SWT provides two equivalent methods in the `Display` class: `asyncExec()` and `syncExec()`.

The first thing that SHriMP does when it opens a graph is to focus on a specific node (usually the root). Focusing on a node entails panning and zooming the display: that is, programmatic manipulation of the interface. This programmatic manipulation of the graph display cannot occur until the graph is actually displayed because all of the visual elements need to have positions and sizes before those attributes can be manipulated. When executing SHriMP within its own VM, the programmatic manipulation is deferred until after

```
SwingUtilities.invokeLater(
    new FocusOnRoots(_shrimpView)
);
```

Fig. 8. Defer focus event (Swing).

the graph is displayed using the `invokeLater()` method, as shown by the code snippet in Fig. 8.

However, when running SHriMP inside Eclipse, the initial rendering of the graph does not occur until the SWT `Composite` that contains SHriMP is initialized. This whole process is triggered by an `SWT.Resize` event that gives the `Composite` its initial size; when, in turn, resizes the `AWT.Panel` it contains; this causes the SHriMP graph to be displayed for the first time. So, for things to work properly the order of events must be: `SWT.Resize`, initial rendering of SHriMP graph, focus on initial node (the programmatic manipulation of the graph). Both the SWT method `asyncExec()` and the Swing method `invokeLater()` must be used to accomplish this order, as detailed by the code snippet in Fig. 9.

The code in Fig. 9 essentially says the following: after all pending SWT interface events have been processed (including the `SWT.Resize`), place a `FocusOnRoots` event at the end of the Swing event queue. Then, after all of the pending Swing events have been processed (including SHriMP initialization), the `FocusOnRoots` event can programmatically manipulate the SHriMP display to focus on the root of the graph.

Blocking the SWT UI from a swing dialog box. Another event queue interaction work-around has been described on the Eclipse Corner Newsgroup (Wilson, 2001); we report it here as it may be of interest to the reader, although we have not yet had to use this technique.

The objective is to launch a Swing dialog box from within an SWT-based application, and to block the SWT

```
Display display = Display.getCurrent();
display.asyncExec(new Runnable() {
    public void run() {
        SwingUtilities.invokeLater(
            new FocusOnRoots(_shrimpView));
    }
});
```

Fig. 9. Defer focus event (Swing in SWT).

```

import javax.swing.JOptionPane;
import org.eclipse.swt.widgets.Display;

// flag for inter-thread communication
final boolean[] done = new boolean[1];

// create new thread for Swing dialog box
new Thread() {
    public void run() {
        JOptionPane.showMessageDialog(
            null, "alert", "alert",
            JOptionPane.ERROR_MESSAGE);
        done[0] = true;
    }
}.start();

// wait for the Swing thread to finish task
while (!done[0]) {
    Display display = Display.getCurrent();
    if (!display.readAndDispatch())
        display.sleep();
}

```

Fig. 10. Swing dialog blocking SWT UI.

execution until the user dismisses the Swing dialog box. The solution given in Wilson (2001) is to use a Boolean object to communicate between the SWT application thread and the Swing dialog box thread. A code snippet detailing this solution is given in Fig. 10.

6. Discussion

This paper documents our experience integrating SHriMP, an information visualization tool, with IBM WebSphere Studio Application Developer Integration Edition, which is based on open source technology from the Eclipse Project. Our work involved control, data, and domain integration.

Control integration. From the control perspective, we integrated SHriMP with WebSphere Studio Application Developer Integration Edition by transforming SHriMP into an Eclipse plug-in. The creation of a plug-in is straightforward: Eclipse provides full support for developing, testing and debugging.

One of the main technical challenges was integrating Swing/AWT-based SHriMP with SWT-based Eclipse. We reported our experience using an experimental mechanism for integrating programs written in these

different GUI frameworks, including our solution to the thread interaction problem we discovered. Researchers and other prototype developers with existing Swing-based code may be interested in trying this experimental mechanism, despite its limitations. Commercial tool developers should use the standard, supported mechanisms for better performance and more consistent look-and-feel. For the tightest interface integration, one should write plug-ins using SWT.

Data and domain integration. Terminals were the main challenge for the data and domain integration aspects of our project. Throughout the paper we discussed terminals in the context of Message Queue System Integrator style message-flow diagrams, but the issues extrapolate to other kinds of flow diagrams.

SHriMP's data model is very similar to RSF (Rigi Standard Form, Müller and Klashinsky, 1988) and GXL (Graph Exchange Language, Holt et al., 2000), both of which are also based on the idea of a graph composed of nodes and arcs. These data models are not designed to deal with terminals, which are another kind of first-class entity that mediates between nodes and arcs. We tried two approaches for expressing terminals within SHriMP's data model, and reported the implications that this had for SHriMP's model-view-controller (MVC) architecture. The easiest part of the data integration was reading the XML-encoded flow diagrams with the MOF frameworks.

SHriMP's advanced visualization features prompted us to render flow diagrams in a different way in SHriMP from that used in MQIntegrator. We discussed these changes with UCD and flow composition specialists at IBM.

6.1. Conclusions

We have drawn three main conclusions from our experience:

First, we confirm Martin's findings (Martin, 1999) that leveraging industrial tool infrastructure can significantly reduce the amount of effort required to develop tool prototypes. In our case, the MOF frameworks essentially eliminated most of the mechanics of the data integration work, and let us focus on more interesting problems.

Second, tight UI integration between Swing-based tools and SWT-based Eclipse seems possible, but there are some important limitations. Loose UI integration with Swing-based tools is fully supported, as is tight UI integration with SWT-based tools. SHriMP's advanced

visualization features made this part of the project more challenging than it might be for other tools.

Third, terminals seem to be a very useful and natural concept, and they should be better supported by exchange formats such as GXL (Holt et al., 2000).

6.2. Future work

Our anticipated future work involves adapting SHriMP to new data domains within Eclipse, and applying the idea of terminals to program visualization.

New domains. The two domains that we are most interested in are MOF and Java. Flow diagrams are one example of data that may be modeled in MOF (using the FCM framework). However, MOF can be used for modeling in many other domains. We hope to generalize the work presented in this paper to visualize any MOF-based models.

Eclipse also includes a full Java development environment, and we would like to integrate SHriMP into this environment for visualizing Java programs. SHriMP has already been used to visualize Java programs, but we would like to build a new fact extractor that performs some static analysis of polymorphic method invocations (such as *Class Hierarchy Analysis* Dean, Grove, and Chambers, 1995; Diwan et al., 1996) and *Rapid Type Analysis* (Bacon, 1997; Bacon and Sweeney, 1996). Determining the targets of polymorphic method invocations is one of the most important tasks in understanding object-oriented programs, and there is relatively little support for it in most program-understanding tools.

Using terminals for program visualization. We think that the idea of terminals presents an opportunity to add extra visual semantics to the standard box-and-arrow diagrams used by many program visualization tools (such as SHriMP). We are not aware of any current program-understanding tool that uses terminals.

One simple possibility is to use terminals to categorize arcs: for example, there could be a terminal for data-related arcs and another for control-related arcs. In flow diagrams, terminals have semantics that are independent of the kind of arcs connected to them. One way to use terminals and retain this independence in program visualization is to have terminals representing normal and exceptional exit paths, as is done with filter nodes in flow diagrams (see Fig. 3).

Using terminals for software architecture. The new ArchJava (Aldrich, Chambers, and Notkin, 2002a, 2002b) architectural description language from the

University of Washington includes terminals, and we are discussing the possibility of using SHriMP to visualize ArchJava architectures with them.

Acknowledgments

We thank Grant Taylor, Mike Beltzner, and Evan Mamas at the IBM Toronto Laboratory for their time and assistance. Discussions with Mike Beltzner contributed directly to our decision to visually distinguish the terminals such that the arcs can be drawn according to a shortest-path heuristic. Randy Giffen and Veronika Irvine from OTI also gave us useful comments on an earlier version of this paper. Participants on the Eclipse Newsgroup have also been helpful, especially: Greg Adams, Veronika Irvine, Jeff McAffer, David Whiteman, and Mike Wilson. Finally, Anne R. James proof-read the article.

This work was funded by the IBM Centre for Advanced Studies at the IBM Toronto Laboratory.

IBM, WebSphere, MQIntegrator, and VisualAge are registered trademarks of International Business Machines Corporation in the United States, other countries, or both. Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both. Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the United States, other countries, or both.

References

- Adams G. External tool interoperability. *Eclipse Newsgroups*, June 2001. Available on <http://www.eclipse.org/newsgroups/>.
- Aldrich J, Chambers C, Notkin D. Architectural reasoning in ArchJava. In: *ECOOP'02*, 2002a, to appear.
- Aldrich J, Chambers C, Notkin D. ArchJava: Connecting software architecture to implementation. In: *ICSE'02*, 2002b, to appear.
- Amsden J. Your first plug-in. *Eclipse Article*, June 2001. Available at <http://www.eclipse.org/articles/>.
- Bacon DF. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD Thesis, UCB/CSD-98-1017, University of California at Berkeley, December 1997.
- Bacon DF, Sweeney PF. Fast static analysis of C++ virtual function calls. In: Coplien J, ed. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Jose, CA, 1996:324–341.
- Best C, Storey M-A, Michaud J. Designing a component-based framework for visualization in software engineering and knowledge engineering. In: Ferrucci F, Vitiello G, eds. *Proceedings*

- of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy, July 2002, submitted.
- Böhm C, Jacopini G. Flow diagrams, turing machines and languages with only two formation rules. *CACM* 1966;9(5):366–371. Also Reprinted in Yourdon EN, ed. *Classics in Software Engineering*. Yourdon Press, 1979.
- Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language—User Guide*. Reading, MA: Addison-Wesley, 1999.
- Casatti F, Ceri S, Pernici B, Pozzi G. Conceptual modeling of workflows. In: *Advances in Object-Oriented Data Modeling*. Cambridge, MA: MIT Press, 2000.
- Coplien J (ed.). In: *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Jose, CA, October 1996.
- Dean J, Grove D, Chambers C. Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff W, ed. *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Århus, Denmark, LNCS 952. Berlin: Springer-Verlag, 1995.
- Diwan A, Eliot J, Moss B, McKinley KS. Simple and effective analysis of statically-typed object-oriented programs. In: Coplien J, ed. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Jose, CA, 1996:292–305.
- Harel D, Gery E. Executable object modeling with statecharts. *IEEE Computer* 1997;30(7):31–42.
- Holt R, Winter A, Schürr A, Sim S. GXL: Towards a standard exchange format. In: Cifuentes C, Kontogiannis K, Balmas F, eds. *WCRE'00*, Brisbane, Australia, November 2000.
- IBM. FAQ about IBM's new tooling strategy and the future of VisualAge for Java. Available at <http://www7.software.ibm.com/vad.nsf/data/document2020>.
- IBM. MQSI: Message queue system integrator. Available at <http://www-4.ibm.com/software/ts/mqseries/>.
- IBM. WebSphere MQ integrator. Available at <http://www-3.ibm.com/software/ts/mqseries/integrator/>.
- IBM. WebSphere studio application developer integration edition. Available at <http://www.ibm.com/software/ad/studiointegration>.
- Irvine V. ActiveX support in SWT. *Eclipse Article*, March 2001. Available at <http://www.eclipse.org/articles/>.
- Irvine V. Limitations of Swing/SWT experimental integration mechanism. *Eclipse Newsgroups*, July 2001. Available at <http://www.eclipse.org/newsgroups/>.
- Leyman F, Roller D. Work-flow based applications. *IBM Systems Journal* 1997;36(1):102–122.
- Litoiu M, Starkey M, Schmidt MT. Flow composition modeling with MOF. In: *Proceedings of ICEIS'01*, Setubal, July 2001.
- Martin J. *Information Engineering Book III: Design and Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
- Martin J. Leveraging IBM VisualAge for C++ for reverse engineering tasks. In: MacKay SA, Howard Johnson J, eds. *Proceedings of the 9th NRC/IBM Centre for Advanced Studies Conference (CASCON)*, Toronto, 1999:83–95.
- Martin J, Odell J. *Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
- Müller HA, Klashinsky K. Rigi—A System for Programming-in-the-large. In: *Proceedings of the 10th ACM/IEEE International Conference on Software Engineering (ICSE)*, Raffles City, Singapore, 1988:80–86.
- Northover S. SWT: The standard widget toolkit. *Eclipse Article*, March 2001. Available on <http://www.eclipse.org/articles/>.
- Northover S, MacLeod C. Creating your own widgets using SWT. *Eclipse Article*, March 2001. Available on <http://www.eclipse.org/articles/>.
- Object Management Group. Flow composition model. Available at <ftp://ftp.omg.org/pub/docs/ad/01-06-09.pdf>.
- Object Management Group (OMG). Meta object facility, 2000. Available on <http://www.omg.org>.
- Object Management Group (OMG). XML metadata interchange (XMI), 2000. Available on <http://www.omg.org>.
- Rayside D, Litoiu M, Storey M-A, Best C. Integrating SHriMP with the IBM WebSphere studio workbench. In: Howard Johnson J, Stewart DA, eds. *Proceedings of the 11th NRC/IBM Centre for Advanced Studies Conference (CASCON)*, Toronto, 2001:79–93.
- Ross D. Structured analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering* 1977;3(1):16–36.
- Storey M-A, Müller HA, Wong K. *Manipulating and Documenting Software Structures*, Singapore: World Scientific, 1996:244–263. Vol. 7 of the Series on Software Engineering and Knowledge Engineering.
- University of Maryland Human Computer Interaction Laboratory. JAZZ Zooming Library. Available on www.cs.umd.edu/hcil/jazz/.
- University of Victoria. SHriMP Views Visualization Tool.
- Wilson M. Blocking SWT from a Swing dialog box. *Eclipse Newsgroups*, July 2001. Available on <http://www.eclipse.org/newsgroups/>.
- Wu J, Storey M-A. A multi-perspective software visualization environment. In: *Proceedings of the 10th NRC/IBM Centre for Advanced Studies Conference (CASCON)*, Toronto, 2000:41–50.
- Yourdon EN (ed.) *Classics in Software Engineering*. Yourdon Press, 1979.