

Integrating SHriMP with the IBM WebSphere Studio Workbench*

Derek Rayside and Marin Litoiu
IBM Centre for Advanced Studies
IBM Toronto Laboratory
{drayside, marin}@ca.ibm.com

Margaret-Anne Storey and Casey Best
Department of Computer Science
University of Victoria
{mstorey, cbest}@csr.uvic.ca

Abstract

This paper provides an experience report for researchers who are interested in integrating their tools with the new IBM WebSphere Studio Workbench. The Workbench (open source at www.eclipse.org) provides an open framework for building integrated development environments. We report on our experience integrating an information visualization tool (called SHriMP Views) with the IBM Workbench. Although SHriMP was originally developed for visualizing programs, it is content independent. We have re-targeted SHriMP for visualizing *flow diagrams*. Flow diagrams 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 both in terms of their semantics and in their visual representation. *Terminals*, which are a first-class entity that mediate between nodes and arcs in flow diagrams, presented the main challenges here. We also report on the main technical challenges we faced, due to the different widget sets used by SHriMP (SWING/AWT) and the Workbench (SWT).

Keywords: integration, software engineering, software visualization, system modeling, flow diagrams, MOF, XMI

*WebSphere Studio Workbench is an IBM product based upon technology from the Eclipse Project. For the purposes of this paper, the reader may consider these terms as logically, but not legally, synonymous.

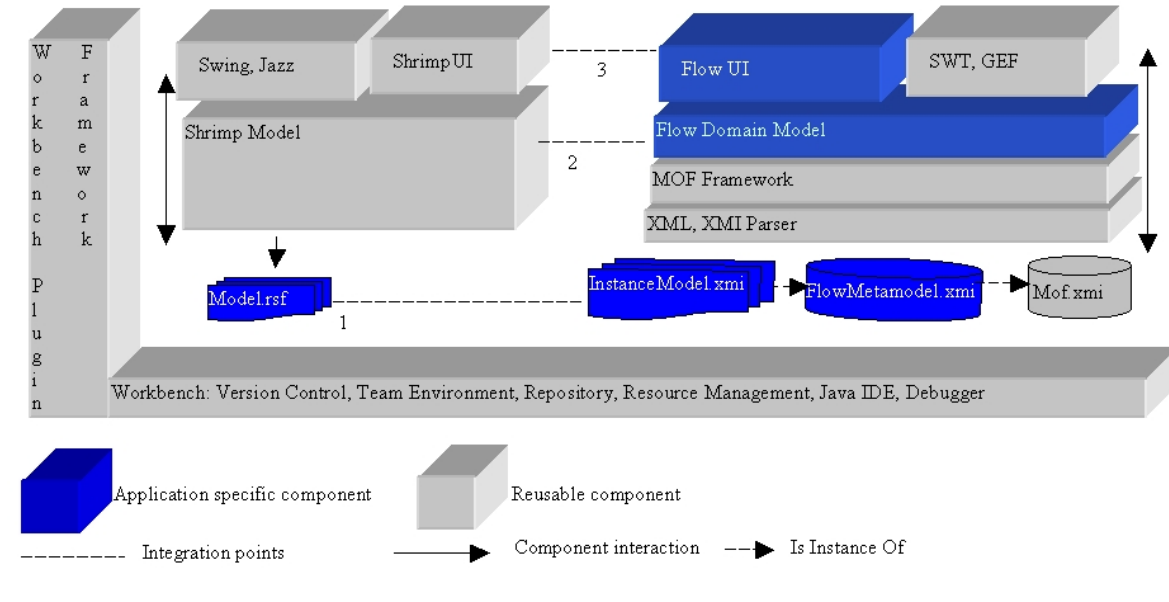
1 Introduction

Leveraging industrial tool infrastructure can significantly reduce the amount of effort required to develop tool prototypes. At CASCON'99, Martin reported that it took him only two weeks to build a complete C++ fact extractor for Rigi [23] based on IBM's VISUALAGE C++, in contrast to the protracted effort that had been required to build the previous fact extractor [22]. Moreover, his new fact extractor was more robust, easier to understand, and 75% less code [22]. Leveraging industrial tool infrastructure allows researchers to focus on research.

IBM's new WebSphere Studio Workbench (based on open source technology from the Eclipse Project) presents some exciting opportunities for researchers who build software engineering tools. 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.

This paper reports our experience integrating a pre-release version of Eclipse with the SHriMP information visualization tool from the University of Victoria [26]. Specifically, we want to use SHriMP to visualize *flow diagrams*.

Figure 1 Eclipse and SHriMP Architecture



Flow diagrams are used in an eBusiness 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.

Figure 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 grey, and project independent re-usable components are shown in a lighter shade of grey. The three labeled, dashed lines represent the points of interaction between SHriMP and Eclipse:

1. represents file-based data exchange between the RSF format used by SHriMP and the XML-based format used by the flow diagram tool in Eclipse; **2.** represents API-based data exchange between the flow diagram tool and SHriMP; **3.** represents the control and user interface integration between the tools.

The MOF 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, *Overview of Eclipse*. 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 connected with terminals.

We faced some interesting challenges in the control integration because SHriMP and Eclipse are based on different widget toolkits: SWING/AWT and SWT, 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 §5 (commercial developers should use the standard control integration mechanisms, which are also described briefly).

Section 6 concludes the paper.

2 Overview of Eclipse

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

Libraries Because the Workbench is specifically intended for building integrated development tools, instead of just integration in general, it also includes infrastructure for resource management, version control, and debugging. The Workbench core also includes a number of useful libraries:

- MOF & XML Tools
- SEF: Source Editing Framework
- GEF: Graphics Editing Framework
- JFACE: high level user interface constructs
- SWT: Standard Widget Toolkit

MOF and SWT are discussed in more detail later in this section. SEF is a framework for building editors of structured text, such as Java programs. GEF is a framework for building editors of structured graphics, such as flow diagrams. JFACE is a library of high level user interface constructs, such as wizards, etc.

Standard Plug-ins The Workbench also comes with three standard sets of plug-ins: Java development tools, web development tools, and plug-in development tools. Each of these are built with the Workbench core infrastructure, and each interoperates with the others. The Java development tools in Workbench will supersede the VisualAge for Java functionality [14].

Flow Diagrams in XML Flow diagrams can be modelled with MOF and serialized in XMI, as previously reported by (Litoiu, Starkey & Schmidt [19]). We intend to leverage the Workbench infrastructure for working with MOF to take advantage of Litoiu et al's results.

2.1 Overview of MOF

MOF [27] is an OMG standard for describing meta-models, models, and instance data. XMI [28] is an associated OMG standard for serializing MOF in XML.


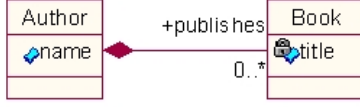

Figure 2 shows the correlation between expressing model information in UML, MOF, and XMI. The first level of meta-modeling is M3 or MOF-core. MOF-core is a subset of UML and contains definitions of *Class*, *Attribute*, *Association* and so on [6, 27]. The second level of meta-modeling is M2 — the application 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 differentiates 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 Figure 1 illustrates the role that MOF and XMI play in our project. `FlowMetaModel.xmi` represents our application domain meta-model (taken from [19]). The Workbench infrastructure can automatically derive a Java API for working with models that instantiate this meta-model. This API is named **Flow Domain Model** in Figure 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.

Figure 2 The relation between MOF, UML, and XMI

<i>UML</i>	<i>MOF</i>	<i>XMI</i>
	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>

2.2 Overview of SWT

Standard Widget Toolkit (SWT) is the software component that delivers native widget functionality for the Eclipse platform in an operating system independent manner. SWT is analogous to SWING/AWT in Java: the difference is in the implementation strategy. SWT uses native widgets wherever possible for three main advantages: *performance*, *look-and-feel*, and *debugging*. SWT-based applications always look and perform like native applications, and any problems with the widgets can be directly replicated in C code. Another benefit of using native widgets is that SWT-based applications can interact with platform specific features, such as ACTIVEEX controls in Windows.

AWT follows a ‘least 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 least common denominator. So, there is only one implementation of SWING that works across all platforms.

SWT takes a different approach: it uses native widgets wherever possible, and only implements a widget in Java if there is no native version available (for example, Windows contains a native tree widget and MOTIF does not). SWT exposes the same public API to applications on all platforms, but provides differ-

ent implementations for each platform. Part of this implementation is a shared library (e.g. a DLL on Windows), that exposes part of the operating system widget API to SWT through the Java native interface (JNI).

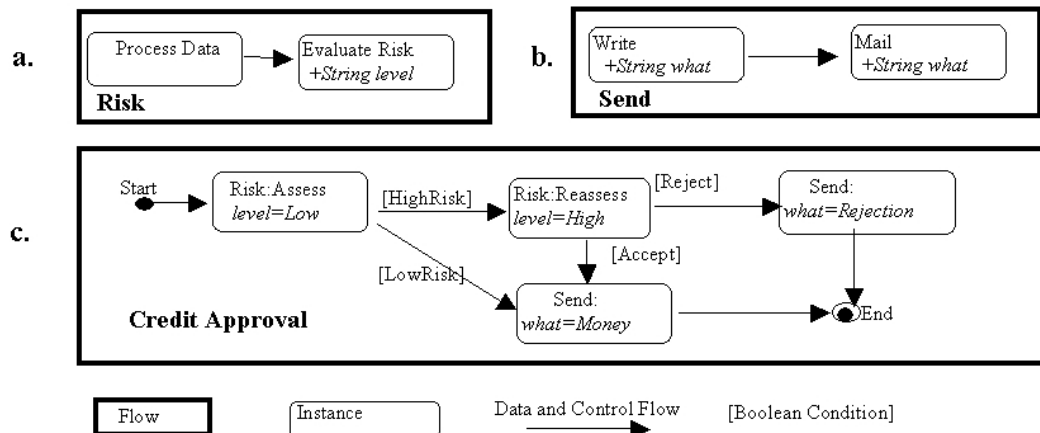
The Eclipse website contains white papers on SWT [24], integrating with ACTIVEEX and OLE on Windows [16], and creating new widgets with SWT [25].

Challenges for Our Project 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. We describe our experience using an experimental mechanism to integrate AWT-based and SWT-based applications in §6.

3 Flow Diagram Overview

There are many specific kinds of technical flow diagrams, such as: Petri nets, data flow diagrams [30], statecharts [12], process dependency diagrams [20, 21], UML activity diagrams [6], and workflows [18, 7]. IBM produces a number of middle-ware products [15] 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

Figure 3 An example flow diagram



by associated runtime environments. The flows in these products are classified as *nano-flow*, *micro-flow* and *macro-flow* (also known as *work-flow*) [19].

An example of micro-flow is the MQSI (Message Queue System Integrator) product, that allows the programmer to draw message-flow diagrams that describe the movement of data between information systems. The associated runtime environment executes these diagrams: i.e. combines, transforms, and delivers the messages. MQSI is typically used to integrate information systems when large corporations merge with former competitors. Much of IBM’s flow related middle-ware is developed in Toronto. Macro-flows use large-scale granular activities and are deployed by large-scale software components such as applications. Workflows 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 modelled with an extension of MOF, called Flow Composition Model (FCM), as was previously reported by Litoiu et al [19]. FCM is in the process of becoming an OMG standard [11]. MOF and FCM are typically serialized in the XML-based XMI format [28]. As was discussed in the previ-

ous section, and shown in Figure 1, the Eclipse platform provides substantial infrastructure for working with MOF models encoded in XMI.

3.1 An Example Flow Diagram

Business processes, or *work-flows*, are a kind of flow diagram that may be more familiar to the business analyst. Figure 3 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**.

Figure 3c shows the main **Credit Approval** process, which is composed of two sub-processes: **Risk** (Figure 3a) and **Send** (Figure 3b). We want to use SHriMP’s advanced features for visualizing these kinds of composition relations. For the simplicity of the presentation, the remainder of the paper will use MQSI to illustrate flow diagrams.

3.2 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 control-flow graphs and data-flow graphs to describe certain aspects of programs. Böhm and Jacopini 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* [5].

Litoiu, Starkey & Schmidt consider that there is an analogy between flows and classes (in the object-oriented programming sense of the word) when modeling flows with MOF [19]. From their perspective, the **Risk:Assess** and **Risk:Reassess** boxes in Figure 3c represent *instances* of the flow **Risk** shown in Figure 3a.

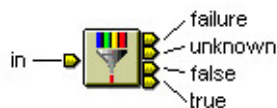
We are interested in the relation between flows and programs because SHriMP was originally developed for program visualization and we are now using it for flow visualization.

3.3 Visual Presentation of Flows

One of the important characteristics of nodes in flow diagrams is that they have specific *terminals* where 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 Figure 4 shows an example of a ‘filter’ node from the MQSI [15] flow editor with its terminals labelled. A filter node is a kind of primitive node that is analogous to a simple conditional test in an imperative programming language.

Figure 4 A Filter Node from MQSI



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 depending on 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 than 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 into a single terminal.

4 Integration with Flows

This project also involved a fair amount of data and domain integration, as well as the control integration discussed in the next section. The *terminals* on nodes posed the main challenge for both data and domain integration. Since it is easier to understand some of the issues by first seeing the ‘whole picture’, so to speak, we will first show screen-captures of our working prototype and then discuss the details. But first, a brief background on SHriMP.

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 [31, 33]. 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.

SHriMP’s advanced visualization features, namely its smooth zooming capabilities within a nested graph and advanced layout algorithms, provide us with an opportunity to re-examine some of the visual conventions used in the current flow diagram browser/editor.

4.1 Rendering Flows in SHriMP

The top picture of Figure 5 shows SHriMP rendering the simple flow diagram `multi1.messageflow` inside the Eclipse

Workbench (`multi1.messageflow` is encoded in XMI using the MOF frameworks discussed previously). The diagram has three nodes: **Input** (top-left), **Output** (top-right) and **DataUpdate** (bottom-centre). The **Input** node has one output terminal (green); the **Output** has one input terminal (blue); the **DataUpdate** node has three terminals: input (blue), regular output (green), and error output (red). Some of the important visual features of the flow diagram rendered in the top picture of Figure 5 are:

- nodes are represented by images
- terminals are represented by images
- terminals are colour-coded
- arcs are connected to terminals
- some terminals have no arcs attached
- some terminals have > 1 arc attached
- terminals are placed to reduce arc length

All of these visual features have had some impact on the data and domain integration effort. Some of these features differ from how flow diagrams are rendered in MQSI: for example, visually differentiated terminals (colour-coding) and terminals automatically placed to reduce arc length.

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 Figures 5 and 6. Figure 5 simulates the user ‘opening’ up and zooming in on the **DataUpdate** node, which shows an HTML table of the node’s property/value pairs. Similarly, in Figure 6, the **SubFlow** node can be viewed as closed, or it can be opened so that the user can interact with and explore the subflow. The nested interchangeable view mechanism more easily allows users to explore information at the required levels of detail to suit different needs and tasks.

Hierarchical Composition Figure 6 illustrates the main motivation for our project: to visualize hierarchically composed flow diagrams. The main flow diagram has three nodes:

Input, **Output** and **SubFlow**. Figure 6 simulates the user opening up the **SubFlow** node and zooming in to see that it is also composed of three nodes: **Input**, **Output** and **Filter**.

4.2 Data Integration

Our project had two data integration aspects of note: reading the flow diagrams from the XMI-based format and representing flow diagrams within SHriMP’s data-model. The former task was fairly straightforward given the MOF frameworks in WebSphere Studio Workbench; the latter task required substantially more work.

SHriMP uses a graph-based data-model, similar to the one embodied by RSF [23] or GXL [13]. 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 mediate between arcs and nodes.

It would be possible to create a new data-model that explicitly incorporates the notion of terminals. However, such an approach would have required a substantial programming effort given SHriMP’s existing code-base, as we would have had to completely replace its data-model. We think that this approach would be worth investigating if one was writing a program to deal with flow diagrams from scratch.

There are two main possibilities for representing terminals within SHriMP’s existing data-model: as *distinct nodes*, or as *attributes* on existing nodes and arcs. We tried both approaches, and our experience was that the latter required less overall effort.

Representing terminals as attributes on existing nodes and arcs requires adding two attributes to every arc (source terminal and target terminal), as well as n attributes to every node, where n 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 $2n$ attributes on every node: the second attribute indicates the image to use for the terminal.

Model-View-Controller (MVC) SHriMP may be considered from the standpoint of the well-known *model-view-controller* architecture. The discussion above describes the changes we

Figure 5 Screen-capture of SHrIMP rendering a flow diagram inside Eclipse

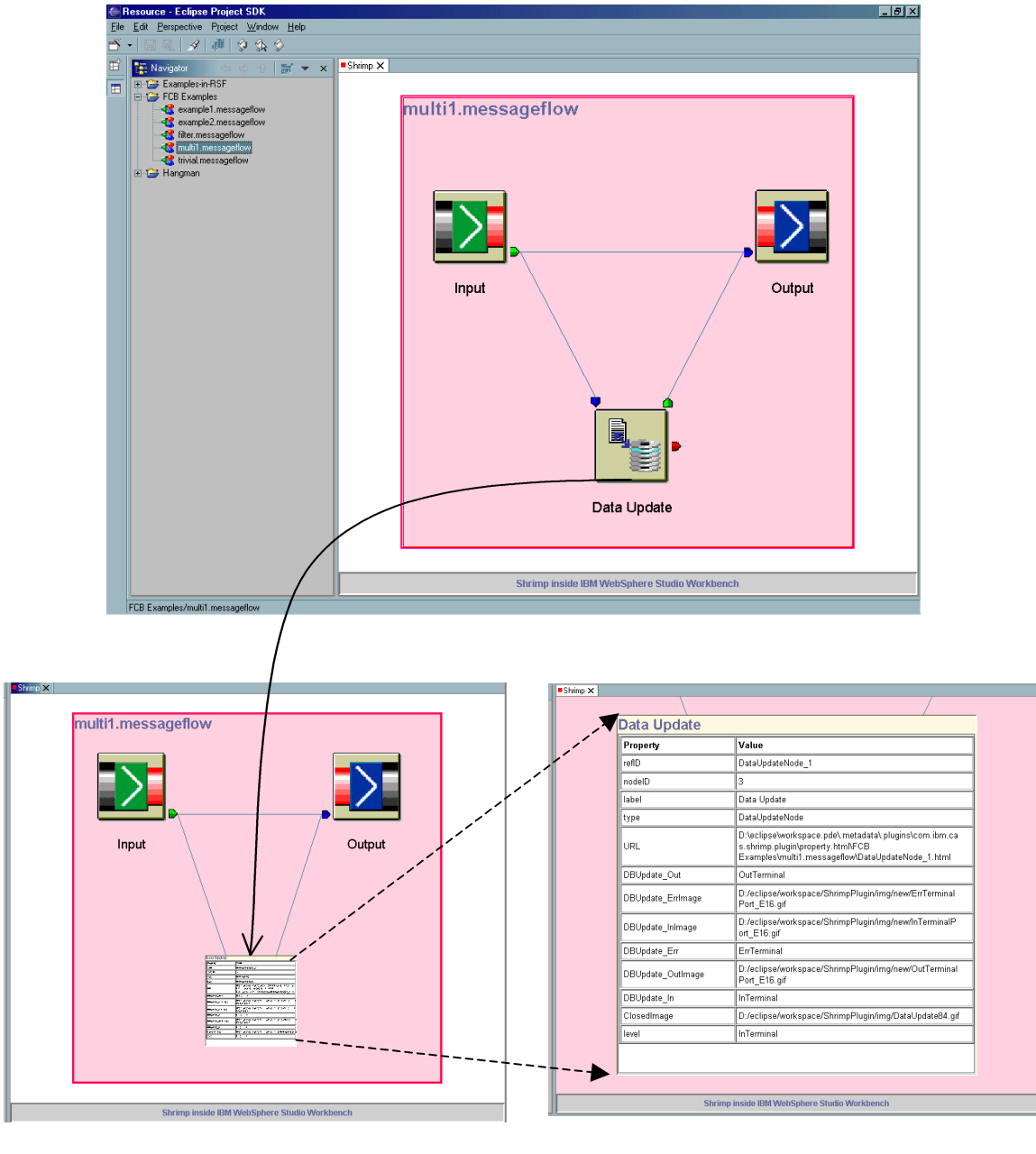
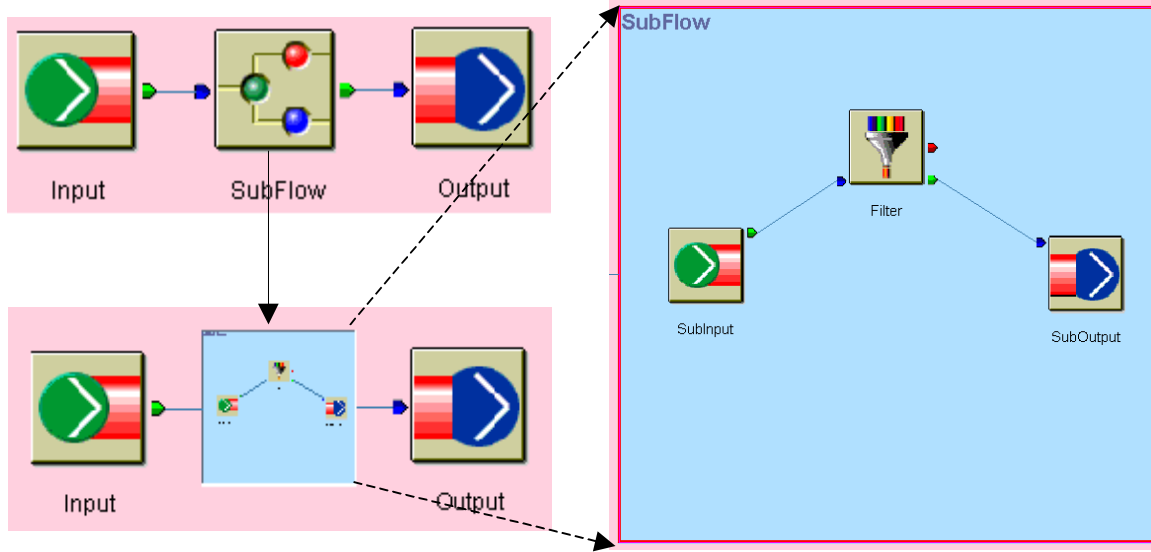


Figure 6 SHriMP zooming in on a hierarchically composed flow diagram



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 idea worth investigating.

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 in order to render an image on the screen: 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 set 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 things in actually getting the terminals to display and zoom properly.

5 Integration with Eclipse

The Eclipse platform is designed to facilitate tool integration, both between a tool and the Workbench, and between tools. 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, etc. The entire Workbench is built using this plug-in architecture.

Since this plug-in architecture is one of the main features of Eclipse, it is well documented in the technical documentation as well as in a number of introductory articles on Eclipse, such as [2]. The focus of this paper is to report

aspects of our experience that are not documented elsewhere. We have used an experimental, undocumented mechanism for seamless interface integration of SWING-based tools with the Workbench, and the majority of this section discusses our experience with this mechanism. First a summary of different mechanisms of integration in Eclipse is provided.

5.1 Control & UI Integration

Control and user interface integration of tools such as SHriMP with the Workbench can be pursued independently of other aspects of integration, such as data integration. The reason is that both SHriMP and the Workbench are fairly content independent: each tool can be used for a wide variety of data. In our project, we actually worked on the control and user interface integration before working on the data integration.

The Workbench can integrate with tools based on four different user interface technologies [1]: SWT, SWING, OLE, and other (i.e. launching an external tool in a separate window and process). SWT is the tightest interface integration and 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 the Workbench: 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 the Workbench itself. These tools can achieve seamless functional integration if the Workbench API's are used.

In-place Java tools can also use ACTIVE X controls (on Windows only). Eclipse provides a mechanism for using ACTIVE X controls from within the Java code [16].

This is the tightest and most seamless form of interface integration.

External Tools Any tool written in any language can be launched from the Workbench with its own separate process. These tools open as separate windows and are not visually integrated with the Workbench. The platform

provides automatic launching of external editors based on file type associations provided to the Workbench or obtained from the underlying operating system.

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 the Workbench, but functionally they are the same as any external-launch tool: all they do is open on an artifact, edit it, and save it.

Swing-based Tools SWING-based tools can be integrated with the Workbench interface in two ways: in a separate window or within the Workbench. 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 launch in a separate process, whereas SWING-based tools can run within the same VM and class libraries as the Workbench.

There is an experimental mechanism for running SWING-based tools in-place within the Workbench, 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.

The main restrictions on this experimental mechanism are [17]:

- It only works on Windows.
- The keyboard doesn't work for all widgets.
- There may be possible deadlock problems.
- The API is internal and subject to change.

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`).

5.2 Our Experience with Swing/SWT Interaction

We have experienced three main problems with this rudimentary SWING/SWT integration mechanism: keyboard events do not work for certain widgets, some complicated repaint operations do not work reliably, and there are some issues in the interaction of the SWING and SWT event queues. The keyboard problem is known to the Eclipse developers, and we are currently waiting for more information from them.

The repaint problems that we have experienced are related to SHriMP's 'hot-box' or 'control-box' feature. The control-box is a user interface feature similar to context sensitive right mouse-button menus. There are two differences: the control-box is a group of widgets that is not limited to a menu, and it is requested by holding down the control key instead of pressing the right mouse button. (In SHriMP, the right mouse button is used to control the animated zooming feature.) The problem is that the control-box is drawn over top of the main window (rather than in it), and so sometimes one (or part of one) gets painted rather than the other. SHriMP issues repaint events for the control-box as long as the user has the control key pressed, so the display changes with time as well as with mouse movement. We are investigating the SWING mechanism that SHriMP uses to display the control-box, and hope to find out how to work around this problem.

We have successfully resolved some of the issues involved in working with two interface event dispatching queues (SWING and SWT). 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 (i.e. moves the mouse, 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: i.e. 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 the graph is displayed using the `invokeLater()` method as shown by the code snippet in Figure 7.

Figure 7 Defer focus event (Swing)

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

However, when running SHriMP inside the Workbench, 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; which, 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

Figure 8 Defer focus event (SWING in SWT)

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

the code snippet in Figure 8.

The code in Figure 8 essentially says the following: after all SWT interface events have been processed (i.e. including the `SWT.Resize`), place a `FocusOnRoots` event at the end of the SWING event queue.

Blocking the SWT UI from a Swing Dialog Box Another event queue interaction work-around has been described on the Eclipse Corner Newsgroup [32]; we report it here as it may be of interest to the reader, although we haven't had to use this technique as of yet.

The objective is to launch a SWING dialog box from within an SWT based application, and to block the SWT execution until the user dismisses the SWING dialog box. The solution given in [32] 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 Figure 9.

6 Discussion

This paper documents our experience integrating SHriMP, an information visualization tool, with IBM's WebSphere Studio Workbench, 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 Eclipse by transforming SHriMP into an Eclipse plugin. The creation of a plugin is straight-forward: Eclipse provides full support for developing, testing and debugging.

One of the main technical challenges was integrating SWING/AWT-based SHriMP with

Figure 9 SWING Dialog blocking SWT UI

```
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();
}
```

SWT-based Eclipse. We reported our experience using an experimental mechanism for integrating programs written in these different GUI frameworks, including some problems with repaint events, keyboard events, and thread interaction. 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.

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 MQSI-style message-flow diagrams, but the issues generalize to other kinds of flow diagrams.

SHriMP's data-model is very similar to RSF [23] and GXL [13], all of which are 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) archi-

ture. The easiest part of the data integration was reading the XMI-encoded flow diagrams with the MOF frameworks.

SHriMP's advanced visualization features prompted us to render flow diagrams differently in SHriMP than is done MQSI. 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 [22] 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 difficult 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 [13].

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* [9, 10] and *Rapid Type Analysis* [3, 4]). 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: e.g. 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 Figure 4).

Acknowledgements

We thank Grant Taylor, Mike Beltzner, and Evan Mamas at the IBM Toronto Laboratory for their time and assistance. Participants on the Eclipse Corner Newsgroup have also been helpful, especially: Greg Adams, Veronika Irvine, Jeff McAffer, David Whiteman, and Mike Wilson.

About the Authors

Derek Rayside: Derek has just completed his MASc (Master of Applied Science) in Electrical & Computer Engineering at the University of Waterloo under the supervision of Professor Kostas Kontogiannis. Derek's BASc is also from the University of Waterloo, in Systems Design Engineering. This year Derek is

pursuing an MA at the University of Toronto's Institute for the History and Philosophy of Science and Technology. He expects to earn a PhD in computer science after the MA.

Derek also intends to leverage Eclipse to build a prototype of a Java program browser designed for understanding polymorphic method invocations [29].

Marin Litoiu: Marin is a Research Associate at the IBM Centre for Advanced Studies in the IBM Toronto Laboratory. Marin holds two PhD degrees: Systems and Computer Engineering (Carleton University, 1999); and Control Systems (University Politehnica of Bucharest, 1995). He has been with IBM since 1997, where he held leading roles in the design of middleware tools for technologies such as RMI, IIOP and MQSERIES. Prior to joining IBM, he was a faculty with the Department of Computers and Control Systems of the UPB and held research visiting positions with Polytechnic of Turin, Italy and Polytechnic University of Catalonia (Spain), European Centre for Parallelism.

Margaret-Anne Storey: Dr. Margaret-Anne (Peggy) Storey is currently a visiting professor at the Department of Aeronautics and Astronautics at MIT, on leave from the Department of Computer Science at the University of Victoria. She is a fellow of the British Columbia Advanced Systems Institute (ASI) and collaborates with IBM on Human-Computer Interaction and eBusiness. She is one of the investigators for CSER (Centre for Software Engineering Research) developing and evaluating software migration technology and an adjunct research scientist at the New Media Innovation Centre in Vancouver, BC. Her research interests include experimental software engineering, program understanding, human-computer interaction, information visualization, eCommerce and graph drawing.

Casey Best: Casey is a MSc student with Professor Margaret-Anne Storey at the University of Victoria. Casey is currently the chief programmer on the SHriMP project.

References

- [1] Greg Adams. External tool interoperability. *Eclipse Corner Newsgroup*, June 2001.
- [2] Jim Amsden. Your First Plug-In. *Eclipse Corner Article*, June 2001.
- [3] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California at Berkeley, December 1997. Supervised by Susan Graham. UCB/CSD-98-1017.
- [4] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Coplien [8], pages 324 – 341.
- [5] C. Böhm and G. Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *CACM*, 9(5):366–371, May 1966. Reprinted in [34].
- [6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language — User Guide*. Addison-Wesley, 1999.
- [7] F. Casatti, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. In *Advances in Object-Oriented Data Modeling*. MIT Press, 2000.
- [8] James Coplien, editor. *Proceedings of ACM/SIGPLAN Conference on Object-Oriented Systems, Languages and Applications (OOPSLA)*, San Jose, California, October 1996.
- [9] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *ECOOP'95*, Århus, Denmark, August 1995. Springer-Verlag. LNCS 952.
- [10] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In Coplien [8], pages 292–305.
- [11] Object Management Group. Flow composition model. <ftp://ftp.omg.org/pub/docs/ad/01-06-09.pdf>.

- [12] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30(7):31–42, 1997.
- [13] R. Holt, A. Winter, A. Schrr, and S. Sim. GXL: Towards a Standard Exchange Format. In Cristina Cifuentes, Kostas Kontogiannis, and Françoise Balmas, editors, *WCRE'00*, Brisbane, Australia, November 2000.
- [14] IBM. *FAQ about IBM's new tooling strategy and the future of VisualAge for Java*. <http://www7.software.ibm.com/vad.nsf/data/document2020>.
- [15] IBM. *MQSI: Message Queue System Integrator*. <http://www-4.ibm.com/software/ts/mqseries/>.
- [16] Veronika Irvine. ActiveX Support in SWT. *Eclipse Corner Article*, March 2001.
- [17] Veronika Irvine. Limitations of Swing/SWT experimental integration mechanism. *Eclipse Corner Newsgroup*, July 2001.
- [18] F. Leyman and D. Roller. Work-flow based applications. *IBM Systems Journal*, 36(1):102–122, 1997.
- [19] M. Litoiu, M. Starkey, and M.T. Schmidt. Flow composition modeling with MOF. In *Proceedings of ICEIS'01*, Setubal, July 2001.
- [20] J. Martin. *Information Engineering Book III: Design and Construction*. Prentice-Hall, 1989.
- [21] J. Martin and J. Odell. *Object-Oriented Analysis and Design*. Prentice-Hall, 1992.
- [22] Johannes Martin. Leveraging IBM VisualAge for C++ for Reverse Engineering Tasks. In Stephen A. MacKay and J. Howard Johnson, editors, *CASCON'99*, pages 83–95, Toronto, November 1999.
- [23] H.A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *ICSE'88*, pages 80–86, Raffles City, Singapore, April 1988.
- [24] Steve Northover. SWT: The Standard Widget Toolkit. *Eclipse Corner Article*, March 2001.
- [25] Steve Northover and Carolyn MacLeod. Creating Your Own Widgets using SWT. *Eclipse Corner Article*, March 2001.
- [26] University of Victoria. Shrimp views visualization tool.
- [27] Object Management Group (OMG). Meta object facility, 2000. <http://www.omg.org>.
- [28] Object Management Group (OMG). Xml metadata interchange (xmi), 2000. <http://www.omg.org>.
- [29] Derek Rayside and Kostas Kontogiannis. On the Syllogistic Structure of Object-Oriented Programming. In Hausi Müller, Mary-Jean Harrold, and Willhelm Schäfer, editors, *ICSE'01*, pages 113–122, Toronto, Canada, May 2001.
- [30] D. Ross. Structured Analysis (SA): A language for communicating ideas. *IEEE Transactions on Software Engineering*, 3(1):16–36, 1977.
- [31] M.-A. Storey, H.A. Müller, and K. Wong. *Manipulating and Documenting Software Structures*, pages 244–263. World Scientific Publishing Co., November 1996. Volume 7 of the Series on Software Engineering and Knowledge Engineering.
- [32] Mike Wilson. Blocking SWT from a Swing dialog box. *Eclipse Corner Newsgroup*, July 2001.
- [33] J. Wu and M.-A. Storey. A multi-perspective software visualization environment. In *CASCON'00*, pages 41–50, Toronto, November 2000.
- [34] Edward Nash Yourdon, editor. *Classics in Software Engineering*. Yourdon Press, 1979.