Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development

Matthias Merdes
EML Research gGmbH
Villa Bosch
Schloss-Wolfsbrunnenweg 33
D-69118 Heidelberg, Germany
<firstname.lastname>@eml-r.villa-bosch.de

Dirk Dorsch
EML Research gGmbH
Villa Bosch
Schloss-Wolfsbrunnenweg 33
D-69118 Heidelberg, Germany
<firstname.lastname>@eml-r.villa-bosch.de

ABSTRACT

The development of a tool for reconstructing UML sequence diagrams from executing Java programs is a challenging task. We implemented such a tool designed to analyze any kind of Java program. Its implementation relies heavily on several advanced features of the Java platform. Although there are a number of research projects in this area usually little information on implementation-related questions or the rationale behind implementation decisions is provided. In this paper we present a thorough study of technological options for the relevant concerns in such a system. The various options are explained and the trade-offs involved are analyzed. We focus on practical aspects of data collection, data representation and meta-model, visualization, editing, and export concerns. Apart from analyzing the available options, we report our own experience in developing a prototype of such a tool in this study. It is of special interest to investigate systematically in what ways the Java platform facilitates (or hinders) the construction of the described reverse engineering tool.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *object-oriented design methods*, D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *reverse engineering, documentation.*

General Terms

Algorithms, Documentation, Design, Experimentation

Keywords

UML models, sequence diagrams, reverse engineering, Java technology

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPPJ 2006, August 30 – September 1, 2006, Mannheim, Germany.
Copyright 2006 ACM 3-939352-05-5/06/08…$5.00.

1. INTRODUCTION

Due to the increasing size and complexity of software applications the understanding of their structure and behavior has become more and more important. Proper specification and design activities are known to be important in producing understandable software. If such specification and design artifacts are unavailable or of poor quality reverse engineering technologies can significantly improve understanding of the design of an existing deployed software system and in general support debugging and maintenance. While modern CASE tools usually support the reconstruction of static structures, the reverse engineering of dynamic behavior is still a topic of on-going research [20], [25].

The development of a tool supporting the reconstruction of the behavior of a running software system must address the major areas of data collection from a (running) system, representation of this data in a suitable meta-model, export of the meta-model's information or its graphical representation as well as post-processing and visualization aspects. These core areas and their mutual dependencies are shown in Figure 1. Clearly, all conceptual components depend on the meta-model. In addition, a visualization mechanism can be based on a suitable export format as discussed in sections 4 and 5. While this figure illustrates the main conceptual components of our sequence diagram reengineering tool a symbolic view of its primary use can be seen in Figure 2: The main purpose of such a tool is to provide a mapping from a Java program to a UML sequence diagram. The various relevant options will be discussed in detail in the following sections. Recurrent technical topics include meta-model engineering, aspect-oriented technologies, XML technologies – especially in the areas of serialization and transformation – and vector graphics.

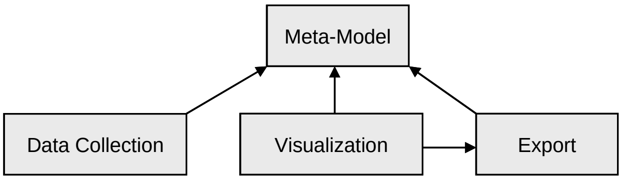
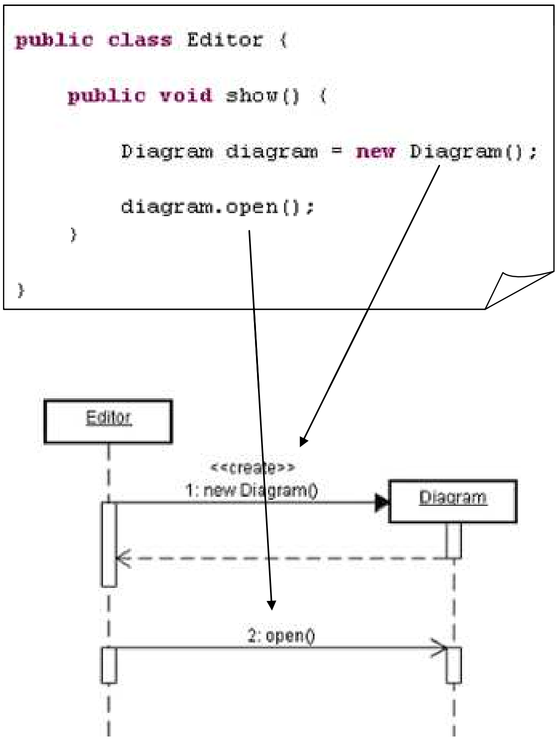


Figure 1. Conceptual components with dependencies

UML sequence diagrams are among the most widely used diagrams of the Unified Model Language (UML) [32]. The UML is now considered the lingua franca of software modeling supporting both structural (static) and behavioral (dynamic) models and their representation as diagrams. Behavioral diagrams include activity, communication, and sequence diagrams. Such sequence diagrams are a popular form to illustrate participants of an interaction and the messages between these participants. They are widely used in specification documents and testing activities [24] as well as in the scientific and technical literature on software engineering.

Sequence diagrams [32] are composed of a few basic and a number of more advanced elements. The basic ingredients of a sequence diagram are illustrated in a very simple example in the right part of Figure 2 along with their respective counterparts in the Java source code on the left-hand side. In such a diagram participants are shown along the horizontal dimension of the diagram as so-called 'life-lines'. In the example, the two participants are 'Editor' and 'Diagram'. These life-lines are connected by arrows symbolizing the messages exchanged between participants. The messages are ordered chronologically along the vertical dimension. In the example, two messages from Editor to Diagram are depicted, namely the constructor message 'new Diagram()' and the 'open()' message. More advanced concepts (not shown in the figure) such as modeling alternatives, loops, and concurrent behavior, can be factored out into so-called 'fragments' for modularization and better readability.



```
public class Editor {

    public void show() {

        Diagram diagram = new Diagram();

        diagram.open();
    }

}
```

Figure 2. Behavior as Java source code and sequence diagram

The reconstruction of the behavior of a software system has been studied extensively both in the static case (from source or byte code) [36], [37], [38] and in the dynamic case (from tracing running systems) [6], [33], [34]. [42] and [7] focus more on interaction with and understanding of sequence diagrams, respectively. An overview of approaches is provided by [25] and [20]. Despite this considerable amount of work there is often little information on implementation-centric questions or the rationale behind implementation decisions. Our study is intended to remedy this lack of such a systematic investigation and is motivated by our experiences in implementing our own sequence diagram reengineering tool. This paper has two main purposes. Firstly, we describe and analyze the possible technological options for the required areas. We also report the lessons learned by our implementation. In this way, the more abstract analysis based on theoretical considerations and the technical and scientific literature is verified and complemented by our own practical experience.

The remainder of this paper is organized as follows. Section 2 explores methods to collect relevant data and section 3 describes the choices for representation of this data using a suitable meta-model. We describe options for visualization and model or graphics export in section 4 and 5, respectively.

2. Data Collection

In this section we will discuss technologies for retrieving information from Java software systems with the purpose of generating instances of a meta-model for UML sequence diagrams. We focus on dynamic (or execution-time) methods but cover static (or development-time) methods as well for the sake of completeness. Static methods gather information from a non-running, (source or byte) code-represented system. Dynamic methods on the other hand record the interaction by observing a system in execution. Data collection requires a mechanism for filtering relevant execution-time events which supports a fine-grained selection of method invocations.

2.1 Development-time Methods

2.1.1 Source Code Based

Using the source code for collecting information about the interaction within an application will have at least one disadvantage: one must have access to the source code. Nevertheless source code analysis is a common practice in the reverse engineering of software systems and supported by most of the available modeling tools. It should be mentioned that the analysis of source code will provide satisfactory results for static diagrams (e.g., class diagrams), but the suitability for the dynamic behavior of an application is limited. If one is interested in a sequence diagram in the form of a common forward engineered diagram (i.e., a visualization of all possible branches of the control flow in the so-called CombinedFragment [32] of the UML), source code analysis will fulfill this requirement. In [37] Rountev, Volgin, and Reddoch introduce an algorithm which maps the control flow to these CombinedFragments. If the intention of the reverse engineering is to visualize the actual interaction any approach of static code analysis is doomed to fail, since it is inherently not possible to completely deduce the state of a system in execution by examining the source code only without actually running the system. Obvious problems include conditional behavior, late binding, and sensor or interactive user input.

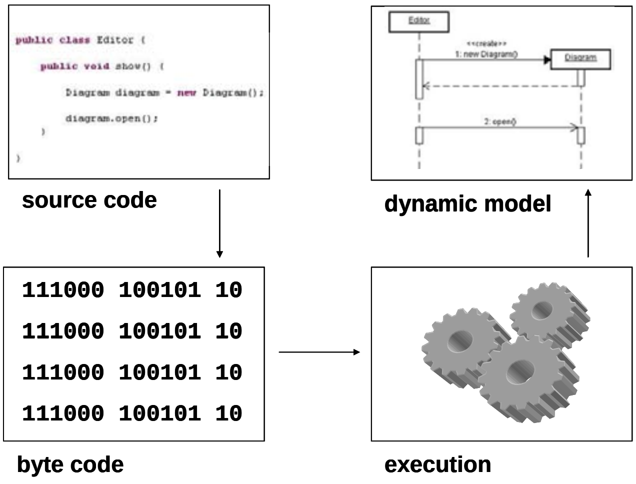
2.1.2 Byte Code Based

The static analysis of code can also be performed with compiled code, i.e., byte code in the case of Java. Such an analysis of byte code basically shares most of the (dis-) advantages of the source code based approach, but it can be applied to compiled systems. One advantage is the fact that processing the byte code must be performed after compilation, separate from the source code, and thus leaves the source code unchanged. This prevents mixing of concerns (application logic and tracing concerns) in the source code and connected maintenance problems.

2.2 Execution-time Methods

The purpose of the dynamic approaches is to record the *effective* flow of control, or more precisely, the sequence of interactions, of a (deployed) system's execution. Any dynamic approach results in a model that represents the actual branches of the application's control flow. In this section we will discuss technologies based on a temporary interception of the program's execution. Basically, we differentiate between the instrumentation of the application itself (i.e., its code) and the instrumentation of its runtime environment.

An overview of the basic workflow from the Java sources to the byte code and on to the UML model and its visualization can be seen in Figure 3. This figure illustrates the more expressive approach of generating the model from dynamic runtime trace information, compared to the static approach described in section 2.1, which relies on source code only.



```
public class Editor {

    public void show() {

        Diagram diagram = new Diagram();

        diagram.open();
    }

}
```

```
111000 100101 10
111000 100101 10
111000 100101 10
111000 100101 10
```

Figure 3. Symbolic steps from source code to sequence diagram model for a Java program (dynamic analysis)

2.2.1 Program Instrumentation

2.2.1.1 Source Code Based

Assuming access to the source code is provided it can be instrumented in a number of ways. Two obvious possibilities are:

1. Modify the source code manually; this is both troublesome and error-prone.

2. Take advantage of aspect-orientation and compile the code with suitable aspects.

Both will finally result in modified source code either explicitly or transparently. Support for filtering can be achieved by a (manual or automatic) manipulation of selected source code fragments. Another related approach is the common logging practice which can be seen as source code manipulation as well. Such an analysis of log-files is discussed in [17].

2.2.1.2 Byte Code Based

Instrumenting the byte code instead of the source code has one advantage: the source code is not manipulated in any way. Again, one could take advantage of aspect-orientation and recompile the byte code with some aspects [5]. In most cases one will have access to the byte code in the form of Java archives (jar files) or raw class files; otherwise this approach will fail. Again, as in the development time case explained in section 2.1.2, byte code manipulation is superior to source code manipulation because of maintenance and versioning issues. In the following section another aspect-oriented approach will be discussed.

2.2.2 Instrumentation of the Runtime Environment

For Java applications the instrumentation of the runtime environment means the instrumentation of the Java Virtual Machine (JVM). When discussing JVM instrumentation the theoretical possibility to develop a customized JVM should be mentioned. Due to the large effort of implementing a new or even modifying an existing virtual machine we won't discuss this approach any further. We prefer to introduce technologies based on virtual machine agents that could be applied to existing JVM implementations. In principle, a custom agent could be developed against the new Java Virtual Machine Tool Interface (JVMTI), which is part of J2SE 5.0. Gadget [16] is an example using an older version of this API for the purpose of extracting the dynamic structure of Java applications. Using the AspectJ or Java-Debug-Interface (JDI) agents as described below allows to focus on a higher level of abstraction compared to the low-level tool interface programming.

2.2.2.1 Java Debug Interface (JDI)

The JDI is part of the Java Platform Debugger Architecture (JPDA) [45]. The JPDA defines three interfaces, namely the Java Virtual Machine Tool Interface (JVMTI, formerly the Java Virtual Machine Debug Interface, JVMDI) which defines the services a virtual machine must provide for debugging purpose, the Java Debug Wire Protocol (JDWP) which defines a protocol allowing the use of different VM implementations and platforms as well as remote debugging, and last but not least the JDI, the Java interface implementation for accessing the JVMTI over JDWP. The debuggee (in our case the observed program) is launched with the JDWP agent, this allows the debugger (in our case the observing application) to receive events from the debuggee by using JDI. For the purpose of reengineering the system's behavior we are mainly interested in events of method executions. As shown in JAVAVIS [33] the JPDA could be successfully used for the purpose of dynamic reverse engineering. One big advantage of the JPDA is the built-in remote accessibility of the observed application. The event registration facility, which can be seen as a filtering mechanism, appears to be too coarse grained, since the class filter is the finest level of granularity. Nevertheless, the JPDA permits the development of reverse engineering tools for both, structural (static) models, such as class diagrams, and behavioral (dynamic) models, such as sequence diagrams.

2.2.2.2 AspectJ Load Time Weaving

Usually aspect-oriented programming is associated with recompiling the source code or byte code with aspects (a.k.a. weaving), as mentioned in section 2.2.1. Starting with version 1.1, the AspectJ technology also offers the possibility of load-time-weaving (LTW) where the defined aspects are woven into the byte code at the time they are *loaded* by the class loader of the Java virtual machine [12]. Hence AspectJ offers the possibility to trace a deployed system without modifying either source code or byte code.

An extensive discussion on how to use AspectJ for the purpose of dynamic reverse engineering of system behavior can be found in [5] and is beyond the scope of this paper. In this section we therefore restrict ourselves to the discussion of the basic concepts of AspectJ needed for this purpose. For detailed information about aspect-orientation and especially AspectJ refer to [15], [23], and [12]. Recent research results and directions can be found in [13].

Generally, aspect-oriented approaches support the modularization of cross-cutting concerns with aspects and weaving specifications. In the case of AspectJ, these concepts are realized by aspects (comparable to classes), advice (comparable to methods) and joinpoints specified by pointcut descriptors. An advice declares what to do before (*before advice*), after (*after advice*) or instead of (*around advice*) an existing behavior addressed by a specific joinpoint. The joinpoint exactly defines a point within the code execution. For retrieving the information needed to model a sequence diagram it is sufficient to take advantage of the predefined *call joinpoints* (representing a method call) and *execution joinpoints* (representing a method execution).

The definition of a joinpoint also offers the possibility of filtering. A joinpoint can address packages, classes, selected methods or work in an even more fine-grained manner. So combining those joinpoints and the arbitrary granularity of the filter mechanism allows for a flexible extraction of the information on the interactions in a running system.

2.3 Comparative Assessment

As presented in the preceding sections, there are numerous ways to implement an execution-tracing data collection mechanism. Discriminating dimensions include manual vs. automatic instrumentation of source or byte code, static vs. dynamic analysis, remote accessibility and performance issues.

If the target environment allows the combined use of version 5 of the Java platform and the latest release of the AspectJ distribution (AspectJ 5) the elegance and non-intrusiveness of the load-time-weaving mechanism in combination with the low performance impact and the expressiveness and flexibility of the join-point-based filter mechanism make the aspect-oriented approach the best solution. This approach is superior in all relevant dimensions, especially compared to the manual integration of tracing and application code due to associated maintenance problems, and compared to a custom JVM or custom JVM agents due to their inherent complexity and huge effort. Hence in our tool we use an AspectJ-based data collection mechanism but we have also implemented and evaluated a prototypical JDI-based data collection mechanism. Such a solution, however, requires a customized filtering extension to achieve an appropriate filtering granularity and suffers from performance problems, especially in the presence of graphical user interfaces.

3. Meta-Model and Data Representation

A central topic which influences other areas, e.g., visualization, editing, or export, is the question of how the recorded data are represented internally. This is best achieved by storing the data in instances of a suitable meta-model. As a sequence diagram generation tool collects information on the *execution* of a program the meta-model must be capable of representing such run-time trace data.

Of course, only a certain subset of a typical complete meta-model will be needed for representing the relevant data. As the execution of a program in an object-oriented language is realized by method calls between sender and receiver with arguments, return types, and possibly exceptions, these are the required meta-model elements. Therefore a compatible meta-model must be employed rather than the actual meta-model of the programming language. Specifically, for an object-oriented programming language like Java a generalized object-oriented meta-model can be used, such as the OMG meta-model, the Meta-Object Facility (MOF) [30], to which other languages than Java can be mapped as well. Meta-models are at the core of recent research and standardization activities in the area of the OMG's Model Driven Architecture (MDA) [28], [39] and, more generally, Model Driven Development (MDD) which encompasses approaches beyond the OMG standards, such as Domain Specific Languages (DSLs) [19] and other non UML-based efforts.

3.1 Meta-Models from MDD Tools

MDD technologies usually generate executable software from instances of meta-models [46]. That implies that tools for such technologies need a representation of the respective meta-model. An example is the free openArchitectureWare Framework (OAW) [48] which includes a meta-model implementation in Java. One of the advantages is that exporting the meta-model to various formats is supported including a number of XMI dialects. The decision for the use of such a meta-model is a trade-off between the advantages of reusing a stable quality implementation and the overhead involved with a much larger meta-model than needed and a certain amount of inflexibility due to the external dependency (e.g., reliance on third-party bug fixing or version schedules).

3.2 UML2 Meta-Model

Since the UML2 specification [32] defines 13 types of diagrams and a large number of classes it would be quite expensive to implement the full UML2 meta-model from scratch. The EclipseUML2 project [9] provides a complete implementation of the UML2 meta-model. It is based on the Eclipse Modeling Framework (EMF) and its meta-model Ecore [10]. While the EMF was designed as a modeling framework and code generation utility the objectives of EclipseUML2 are to provide an implementation of the UML meta-model to support the development of UML based modeling tools. EclipseUML2 and the EMF also provide support for the XML Metadata Interchange language (XMI) [31] export of the model. This built-in XMI serialization is a big advantage for the model exchange as described in section 5.1.1. Despite those advantages the usage of the UML2 meta-model for representing only sequence diagrams could cause some cognitive overhead as most parts of the UML structure won't be needed.

3.3 Custom Meta-Model

The overhead produced by using the complete UML2 meta-model leads to the idea of developing a light-weight custom meta-model. As mentioned in the introduction one can reduce the model to a few basic components which will result in a very light-weight design. However, one has to face the drawbacks of developing an export mechanism in addition to persistence and visualization mechanisms.

3.4 Comparative Assessment

As the central abstraction in a sequence diagram reverse engineering tool is the data about the recorded sequences from actual program runs, its representation in instances of a meta-model is a crucial question. There are two basic options to choose from: reusing an external meta-model or implementing a custom meta-model. The reuse of an external meta-model offers the well-known substantial advantages of software reuse [21], such as implementation cost savings and proven implementation quality. From a Java perspective both options are equally viable: Third-party meta-model implementations are very often based on Java technology and Java is also well suited for a custom implementation. In the given situation where only a very small subset of the meta-model is needed and the cost of a custom implementation is low we opted for the simplicity and flexibility of a custom implementation.

4. Visualization and Model Post-Processing

One central requirement for a UML2 sequence diagram reverse engineering tool is the visualization of the recorded data, that is, some form of transformation or mapping from the meta-model instances to a visual representation. Indeed, as a human observer interacts with such models primarily in their visual form the *graphical display* as a sequence diagram can be considered the main purpose of such a tool. In the following sections we will discuss the possibilities of generating diagrams after recording the tracing data and analyze a number of possible methods. We also describe interactive rendering during the data recording.

4.1 Third-Party Batch Visualization

Methods based on third-party tools visualize the collected model information by exporting to viewable formats or intermediate stages of such formats. Generally, we can differentiate between using common graphics formats (such as PNG, JPG etc.) and displaying the result in third party UML modeling tools.

The main drawback of using static graphics formats is the lack of adequate diagram interaction possibilities. As bitmap formats offer the most simple export of a visualized diagram we will briefly explain a lightweight technology for generating various kinds of graphics output. The free tool UMLGraph is an example of such a technology. It allows the specification of a sequence diagram in a declarative way [40], [41] using pic macros. With the GNU plotutils program pic2plot [18] these specifications can be transformed into various graphics formats (such as PNG, SVG, PostScript, and many more). An approach for integrating this technology into a tool is the usage of a template engine (e.g., Apache Jakarta Velocity [2]) for transforming the instances of the meta-model to the pic syntax and applying pic2plot to the result. Main advantages include the implicit use of a high-quality layout engine and the broad graphics format support. Generally, all methods described in section 5.2 that lead to graphics export can be used for such batch visualizations.

4.2 Real-Time Visualization

It is an interesting option to perform model visualization during the data collection process. Especially for slower running or GUI input driven programs this can be a useful way of observing the behavior of the program in real time during the recording process. In [33] a similar approach is taken and combined with explicit means to trigger the steps in the program execution resulting in a special kind of visual debugger.

A number of implementation choices exist, especially the development of a custom viewer and an SVG-based solution. Although SVG is better known as a vector format for static graphics it also supports the visualization of time-dependent data in the form of animations [50]. For this purpose it is possible to programmatically add nodes to the SVG tree to reflect the temporal evolution of the diagram. In principle, the same two possible SVG-based approaches as those detailed in section 4.3.1 can be used.

4.3 Interactive Visualization and Editing

Viewing a non-modifiable diagram can already be useful. For diagrams constructed manually with a design tool this may be sufficient because these diagrams are usually not overly large as the content is under direct and explicit control of the modeler. If, however, the diagram is generated automatically by collecting data from an executing program it can quickly become very large. This may be caused by too many participants (lifelines) in the diagram or by recording too many interactions over time or by showing too much or unwanted detail. As pointed out by Sharp and Rountev [42] such a large diagram quickly becomes useless to a human user and thus a possibility to interactively explore the diagram is needed. Such an interactive viewing can in principle be extended to support the editing and modification of a diagram for further export. We describe three possibilities for realizing an interactive interface in the following sections.

4.3.1 SVG Based Solution

A viewer for the interactive exploration and possibly manipulation of sequence diagrams can be realized with Scalable Vector Graphics (SVG) [50]. We describe this W3C standard-based vector graphics format in more detail in section 5.2.2. The two principle possibilities are:

1. SVG combined with EcmaScript
2. Extension of an open source SVG viewer

In the first case the model is exported to an SVG image and combined with an embedded EcmaScript program for interaction. The scripting language EcmaScript [8] is the standardized version of JavaScript. While the latter was originally introduced by Netscape for client-side manipulation of web pages and the browser, EcmaScript is a general-purpose scripting language. It is the official scripting language for SVG viewers with standardized language bindings [52]. EcmaScript provides mouse event handling and enables the manipulation of SVG elements, attributes, and text values through the SVG Document Object Model (DOM). Nodes can be added and deleted and values modified. As SVG elements can be grouped during the export process and attributes can be inherited it becomes feasible to manipulate a whole UML sequence diagram element with a single attribute change in the parent group. It is beneficial that such an EcmaScript-based interactive explorer can be embedded into (or referenced from) the SVG file. Thus the image can be explored interactively in every SVG-compatible viewer including web browsers equipped with an SVG plug-in. Disadvantages of such a scripting language compared to a high-level object-oriented programming language like Java include limitations of the core language libraries, as well as fewer third-party libraries and generally comparatively poor (though slowly improving) tool support for EcmaScript development.

As an alternative an interactive viewer can be based on a Java implementation of an SVG library, such as the Apache Batik Toolkit [3] which includes parser, viewer, and an implementation of the Java language bindings according to the standard [51]. This toolkit is an open-source product which can be extended to support custom behavior either by modifying the existing viewer or by adding event-handlers with DOM-manipulating custom behavior to the view leaving the core viewer unmodified. While the first possibility as described in the preceding section is more powerful it requires changes to the original code which is a potential source of maintenance problems. The second approach is comparable to the one described for EcmaScript but with the greater power of Java compared to EcmaScript. The required manipulation of the DOM is possible but sometimes troublesome. The main advantage of using the Java API of Batik is the possibility to reuse a stable production-quality and free implementation. This approach to extend the existing Batik SVG viewer with custom interaction possibilities is described in [29] for the display of interactive maps within the Geographic Information Science (GIS) domain.

4.3.2 Custom Viewer

The most flexible approach is to build a custom viewer from scratch in Java, or even based on diagramming libraries such as the Graphical Editing Framework (GEF) [11] or JGraph [22]. The advantage of this approach is that the structures in a sequence diagram can be manipulated at the appropriate level of abstraction. In the SVG implementation manipulation of sequence diagram elements requires manipulation of the geometric representation of these elements. In that case the programmatic interaction is at the wrong level of abstraction, namely at the level of the graphical presentation and not at the level of the model itself. With a custom viewer, however, the display can be modified as response to user input by manipulating instances of the meta-model and their graphical representation. This can be achieved by adhering to the well-known Model-View-Controller (MVC) paradigm [26], a combination of the Observer, Composite, and Strategy design patterns [14] which promotes better design and maintainability. In this design changes can be applied to the model and automatically reflected in the graphical representation.

The drawback to this approach is primarily the fact that the rendering has to be implemented from scratch using only the basic abstractions such as points, lines, and text provided by the programming language, in this case Java or a suitable library. For a more complex interactive viewer, which may support hiding, folding, deleting of structures, or zooming and other manipulations, the greater expressiveness and power of Java (compared to SVG-viewer embedded EcmaScript) clearly outweighs this disadvantage. Additionally, if the model storage and diagram visualization concerns are handled within the same technology the overhead for and complexity of interfacing between technology layers (e.g., between Java and SVG/EcmaScript) can be saved. This is especially important for advanced interaction features which require semantic information stored in the model.

4.4 Comparative Assessment

In this section we described various sequence diagram visualization options and technologies. These include batch, real-time, and interactive visualization. While the batch mode provides basic visualization support, the usefulness of a sequence diagram reengineering and visualization tool is greatly increased if real-time and interactive visualization are supported. Thus, our tool also supports these two advanced options. The described SVG-based approaches have mainly the following advantages:

- Reuse of the base rendering mechanism of commercial (SVG browser plugins) or open source viewers (e.g., Batik)
- Ubiquitous deployability in the case of an EcmaScript-based viewer embedded within the SVG document due to readily available web browser plugins

However, these advantages are reduced by the cost and effort of bridging the technology gap between the recording and model storage technology (Java) and the viewing/rendering technology (EcmaScript/SVG). Especially for the advanced interaction features of our tool the flexibility of a custom viewer is crucial. We therefore decided to implement a custom viewer based solely on Java without an SVG-based implementation.

5. Export

In a model reengineering tool the model information is represented at different levels including an abstract non-visual level for the core model information and a more concrete level for the visual representation in a graphical user interface. The information at both levels has a distinct value for its respective purpose and therefore a tool should be able to export this information at both levels. Additionally, a third possibility is to export an animated version of the model. Such an animation combines the graphical representation with a temporal dimension thus capturing some of the actual dynamics encountered during the recording phase.

5.1 Model Information Export

Models are exported for a number of reasons including:

1. Import into other UML tools

2. As source for transformations to other representations, such as content-only (e.g., graphics) or textual model descriptions (e.g., DSLs)

3. As a persistence mechanism for the modeling application if it allows some form of editing or manipulation the state of which might need to be persisted

Options for such an export are XMI export, JavaBeans XML export, or custom (possibly binary) file formats. We describe each option briefly in the following.

5.1.1 XML Metadata Interchange (XMI)

UML models can be represented and shared between modeling tools and repositories with the help of the XMI standard defined by the OMG [31]. This standard is quite comprehensive and has evolved considerably to the current version. However, the existence of various dialects of the standards (as evidenced by the different model import filters of some modeling tools), constitutes a major problem for interoperability.

5.1.2 XML Data-Binding Based Serialization

An alternative export of model information can be accomplished by using the default XML serialization mechanism of the Java language. The initial Java Beans serialization mechanism was a binary one (see next section), which was and still is useful as a short-time serialization mechanism, e.g., for RMI remoting. It is very sensitive to versioning problems and unsuited to processing by non-Java technologies. Due to these problems and to the general growing importance of XML technologies, and in order to support long-term persistence a new XML-based serialization mechanism for Java Beans was added to the language in version 1.4 [44].

In light of the XMI interoperability problems the robustness, availability, and simplicity of this serialization mechanism can outweigh its limitations, namely the missing import capabilities into third-party modeling tools, especially in connection with a custom meta-model. The advantages of this serialization mechanism are limited to certain situations where (light-weight) models are created for documentation or ad-hoc communication purposes. This mechanism should not be used for creating persistent software lifecycle artifacts where model interchange is crucial.

5.1.3 Custom File Format

A binary custom (with respect to the contents not to the general structure) file format can be realized easily. To this end, the mentioned *binary* Java serialization mechanism is applied to modeling information represented in memory as an instance of the meta-model. The usefulness of such an export is quite limited, however, and can mainly be used as a proprietary persistence format for the application itself. It is not well suited for further processing or exchange with other tools, mainly due to its non-self-describing syntactic nature (i.e., binary Java persistence format) and missing meta-model (i.e., the static design of the stored objects).

5.2 Graphics Export

For many users and usage scenarios the export of modeling information is not needed; the export of images is sufficient. As mentioned earlier sequence diagrams play an important role in software specification and documentation artifacts as well as a basis for test coverage criteria [4], [24]. For the use within these documents and activities a visual form is needed and therefore a possibility to export diagram representations of the model both as static graphics and as animated diagrams.

5.2.1 Bitmaps

Bitmaps are the most simple of graphics formats and a large number of formats exist. The most popular formats include uncompressed bitmaps like Windows bitmaps (BMP) or TIFF bitmaps and compressed (lossy or lossless) formats like GIF, PNG, and JPEG. The main advantages of these formats include their wide-spread use, graphics tool and native browser support. The most popular formats like GIF and JPEG are also directly supported in programming languages like Java without third-party libraries or filters. Due to the discrete nature of the information encoding the contents of such an image cannot in general be scaled (or rotated by arbitrary angles) without lowering the image quality. This is especially true for drawings and text which are the constituents of sequence diagrams. Thus, bitmaps are primarily useful for direct processing between applications (e.g., via screenshots and clipboards), general screen-based use or medium-quality printed documentation but not necessarily for high-quality printing, such as books etc.

5.2.2 Vector Graphics

Vector graphics do not suffer from the inherent limitations of bitmaps with respect to image manipulations such as zooming. The structures in vector graphics images are not rastered but represented by a more abstract description on the level of lines for general drawings and letters for text. This enables reproduction at arbitrary resolutions and in many cases also leads to a smaller file size. Vector graphics formats exist in proprietary versions such as Adobe's PostScript (PS), Encapsulated PostScript (EPS), and PDF formats or open-standards based versions, notably the W3C's Scalable Vector Graphics (SVG) [50]. They also can be differentiated by their binary (PDF) or textual representation (SVG, PS). Of these formats SVG is the only XML-based format.

Although the Adobe family of formats is proprietary it is very widely used for electronic documents (PDF) [1] with the free Adobe Acrobat viewer, printers (PS), and within the printing industry. So sequence diagrams exported to PDF are immediately useful for sharing, viewing, and printing. Although free [27] as well as commercial programming libraries [35] for the generation of these formats exist the known disadvantages (e.g., legal as well as technical issues) of a proprietary format are most relevant for the creation process. Also the level of abstraction in these libraries varies and the API itself is not standardized. In principle, PDF can be generated directly at a high level of abstraction with the help of XSL formatting objects (XSL-FO) [49]. These formatting objects can be applied to a serialized form of an instance of the meta-model. Interestingly, this part of the XSL specification has enjoyed far less success than the XSL transformation part and is not widely supported. However, there is a fairly advanced implementation called FOP (Formatting Objects Processor) within

Apache's XML Graphics Project. A custom implementation of, e.g., a PostScript export is not advisable as the complexity and investment can be considerable.

The SVG standard [50] is a fairly recent development by the W3C consortium. The current production version 1.1 includes rich support for static vector graphics including text and bitmap integration as well as support for animation. As an XML-based format it is widely supported in the areas of generation (data-binding), parsing and transformation technologies (XSLT, XSL-FO) and provides very good editor, modeling tool, and persistence support. While these are generic advantages of XML-based formats special support for SVG is also growing in the area of viewers (e.g., Apache Batik) and browser plug-ins (e.g., Adobe, Corel) and as persistence format in many graphics tools. The Apache Batik Project [3] also provides a Java API for SVG parsing, generation, conversion, and a DOM implementation compliant to the standard.

These properties make SVG a suitable format for the export of models as diagrams. Additionally, SVG supports a generic extension mechanism for handling metadata [50]. In principle, this could be used to embed model information – possibly directly in XMI format – or processing status and history into the diagram representation. The SVG file could then be used as both an image format and a persistence format for the modeling application itself. An example of embedding domain knowledge at the level of model information into SVG metadata is described in [29] for the geographic domain. Additionally, SVG supports vector graphics based animation, which we describe in the next section.

5.3 Animation

The usefulness of animation as a tool for improving the understanding of sequence diagrams has been studied by Burd et al. [7], who find that control flow comprehensibility can be improved by using animated sequence diagrams compared to static sequence diagrams. There also seems to be initial support for such animated diagrams in commercial products [47].

The consideration between animated bitmaps (GIF) and vector graphics (SVG) is similar to that for the case of static diagrams. While the support of animated GIFs in browsers and, generally, in tools is better, SVG animation offers the known advantages of vector graphics, i.e., smaller file size, better quality, and scalability for text and drawings. Additionally, SVG animations are just XML files and could thus be easily post-processed by, e.g., XSL transformations to generate different representations, such as textual descriptions of the sequence or series of single images.

5.4 Comparative Assessment

For a sequence diagram reengineering tool export possibilities are very important. This includes export of both the semantic (model) information as well as a visual description (image data). Despite the well-known practical XMI interoperability problems support for this model exchange format is mandatory for a modeling tool. The built-in XML and binary serialization formats of the Java language provide useful mechanisms for intermediate storage (e.g., for model transformations) and for proprietary persistence formats, respectively.

Graphics export support includes more widely-used bitmaps, such as GIF with associated scaling and printing problems, and the less common but more scalable vector graphics formats, such as SVG. With this mix of advantages and drawbacks there is no single solution but support of both kinds of formats is useful. Animation export possibilities are a useful enhancement which can contribute to the improvement of model and, hence, program logic comprehension.

To support model information persistence for the application itself, we opted to use the JavaBeans built-in XML format. This also offers the possibility to easily extend the export to XML-based standards, like XMI and SVG, by applying XSL Transformations [53] to the serialized model. We will use this approach to support at least one important XMI dialect as part of our future work.

As with the other concerns, the mixture of features built into the Java language and the availability of third-party libraries and interfaces provide a strong foundation for the tool implementation.

6. Conclusion and Future Work

In this paper we presented a detailed study of technological choices for various implementation aspects of a dynamic UML sequence diagram reengineering tool from a Java-centric perspective. The implementation of such a tool presents a considerable challenge and many important strategic and technological decisions have to be made. Our study complements the existing body of literature on the subject of sequence diagram reengineering, or more generally, trace visualization, by adding thorough technical advice for those interested in attempting the implementation of such a system, especially in the Java language. In many cases there is no one single correct technological solution to a given implementation problem. By comparing the advantages and drawbacks of each alternative and reporting experiences from our own implementation this study provides assistance for informed technological decisions within the domain of Java-based sequence diagram reengineering, especially in the areas of data collection, data representation with meta-model instances, interactive model visualization and various export options.

We showed that Java is a very suitable language for the development of such a tool in two respects: While the virtual machine-based execution mechanism provides excellent support for tracing mechanisms for data collection, the many advanced features of Java discussed above as well as the rich set of existing libraries for many aspects facilitate the development of the tool as a whole. Thus Java is both: a technology that lends itself to a number of elegant reengineering techiques as well as a powerful means to implement a reengineering tool. The former provide access to the necessary tracing information while the latter processes this information.

The tool described in this paper is currently being integrated with the MORABIT component runtime infrastructure, a middleware for resource-aware runtime testing [43]. In the future we plan to enhance our own prototype implementation to include advanced features such as animation export, multithreading support and plug-in-based IDE and API-based custom integrations.

7. ACKNOWLEDGMENTS

This work has been funded by the Klaus Tschira Foundation (KTS) and the Landesstiftung Baden-Württemberg within the MORABIT research project. We thank our colleagues Elad Messing and Daniel Brenner for reviewing the manuscript and the anonymous reviewers for providing helpful suggestions for improvement.

8. REFERENCES

[1] Adobe Systems. *PDF Reference Fifth Edition, Adobe Portable Document Format Version 1.6*. Adobe System Inc., partners.adobe.com, 2004.

[2] Apache. *Velocity*. The Apache Jakarta Project, jakarta.apache.org/velocity/.

[3] Apache. *Batik SVG Toolkit*. The Apache XML Project, xml.apache.org/batik/.

[4] Binder, R. *Testing Object Oriented Systems. Models, Patterns and Tools*. Addison Wesley, 1999.

[5] Briand, L.C., Labiche, Y., and Leduc, J. *Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Real-Time Java Software*. Technical Report SCE-04-04, Carleton University, 2004.

[6] Briand, L. C., Labiche, Y., and Miao, Y. Towards the Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering* (November 13 - 17, 2003). WCRE. IEEE Computer Society, Washington, DC, 2003, 57.

[7] Burd, E., Overy, D., and Wheetman, A. Evaluating Using Animation to Improve Understanding of Sequence Diagrams. In *Proceedings of the 10th international Workshop on Program Comprehension* (June 27 - 29, 2002). IWPC. IEEE Computer Society, Washington, DC, 2002, 107.

[8] ECMA International: *Standard ECMA-262 ECMAScript Language Specification*. ECMA International, www.ecma-international.org, 1999.

[9] Eclipse Project: *The EclipseUML2 project*. Eclipse Project Universal Tool Platform, www.eclipse.org/uml2/.

[10] Eclipse project. *Eclipse Modeling Framework (EMF)*. Eclipse Project Universal Tool Platform, www.eclipse.org/emf/.

[11] Eclipse Project: *Graphical Editing Framework (GEF)*. Eclipse Project Universal Tool Platform, www.eclipse.org/gef/.

[12] Eclipse Project: *AspectJ project*. eclipse.org, www.eclipse.org/aspectj/.

[13] Filman, R. E., Elrad, T., Clarke, S. and Aksit, M. *Aspect-Oriented Software Development*. Pearson Education, 2005.

[14] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] Gradecki, J. and Lesiecki, N. *Mastering AspectJ - Aspect Oriented Programming in Java*. Wiley Publishing Inc, 2003.

[16] Gargiulo, J. and Mancoridis, S. Gadget: A Tool for Extracting the Dynamic Structure of Java Programs. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering SEKE* (June 2001). 2001.

[17] Gannod, G. and Murthy, S. Using Log Files to Reconstruct State-Based Software Architectures. In *Proceedings of the Working Conference on Software Architecture Reconstruction Workshop*. IEEE, 2002, 5-7.

[18] GNU: *The plotutils Package*. Free Software Foundation Inc., www.gnu.org/software/plotutils/.

[19] Greenfield, J., Short, K., Cook, S. and Kent, S. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley Publishing Inc, 2004.

[20] Hamou-Lhadj, A. and Lethbridge, T. C. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre For Advanced Studies on Collaborative Research* (Markham, Ontario, Canada, October 04 - 07, 2004). H. Lutfiyya, J. Singer, and D. A. Stewart, Eds. IBM Centre for Advanced Studies Conference. IBM Press, 2004, 42-55.

[21] Jacobsen, I., Griss, M. and Jonnson, P. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1997

[22] JGraph ltd. *JGraph*. www.jgraph.com/.

[23] Kiselev, I. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, 2003.

[24] Fraikin, F. and Leonhardt, T. SeDiTeC " Testing Based on Sequence Diagrams. In *Proceedings of the 17th IEEE international Conference on Automated Software Engineering* (September 23 - 27, 2002). Automated Software Engineering. IEEE Computer Society, Washington, DC, 2002, 261.

[25] Kollman, R., Selonen, P., Stroulia, E., Systä, T., and Zundorf, A. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (Wcre'02)* (October 29 - November 01, 2002). WCRE. IEEE Computer Society, Washington, DC, 2002, 22.

[26] Krasner, G. E. and Pope, S. T. *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. SIGS Publications, 1988, pages 26-49.

[27] Lowagie, B. *iText*. www.lowagie.com/iText/.

[28] Mukerji, J. and Miller, J. *MDA Guide Version 1.0.1*. Object Management Group, www.omg.org, 2003.

[29] Merdes, M., Häußler, J. and Zipf, A. GML2GML: Generic and Interoperable Round-Trip Geodata Editing - Concepts and Example. *8th AGILE Conference on GIScience*, 2005.

[30] OMG: *Meta Object Facility (MOF) Specification Version 1.4*. Object Management Group, www.omg.org, 2002.

[31] OMG: *XML Metadata Interchange (XMI) Specification version 2.0*. Object Management Group, www.omg.org, 2003.

[32] OMG: *UML 2.0 Superstructure Specification*. Object Management Group, www.omg.org, 2004.

[33] Oechsle, R. and Schmitt, T. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the

Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, international Seminar* (May 20 - 25, 2001). S. Diehl, Ed. Lecture Notes In Computer Science, vol. 2269. Springer-Verlag, London, 2002, 176-190.

[34] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. Visualizing the Execution of Java Programs. In *Revised Lectures on Software Visualization, international Seminar* (May 20 - 25, 2001). S. Diehl, Ed. Lecture Notes In Computer Science, vol. 2269. Springer-Verlag, London, 2002, 151-162.

[35] Qoppa. *jPDFWriter*. Qoppa Software, www.qoppa.com/jpindex.html.

[36] PRESTO. *RED Project*. Presto Research Group Ohio State University, nomad.cse.ohio-state.edu/red/.

[37] Rountev, A., Volgin, O., and Reddoch, M. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Lisbon, Portugal, September 05 - 06, 2005). M. Ernst and T. Jensen, Eds. PASTE '05. ACM Press, New York, NY, 2005, 96-102.

[38] Systä, T., Koskimies, K., and Müller, H. 2001. Shimba—an environment for reverse engineering Java software systems. *Softw. Pract. Exper.* 31, 4 (Apr. 2001), 371-394.

[39] Soley, R. and Group, O. S. *Model Driven Architecture*. Object Management Group, www.omg.org, 2000.

[40] Spinellis, D. *UMLGraph*. www.spinellis.gr/sw/umlgraph/.

[41] Spinnelis, D.: On the Declarative Specification of Models. *IEEE Software Volume 20 Issue 2*. 2003, pages 94-96.

[42] Sharp, R. and Rountev, A. Interactive Exploration of UML Sequence Diagrams. In *Proceedings of the IEEE Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*. 2005, 8-13.

[43] Suliman, D., Paech, B., Borner, L., Atkinson, C., Brenner, D., Merdes, M., Malaka, R. The MORABIT Approach to Runtime Component Testing. In *Proceedings of the Second International Workshop on Testing and Quality Assurance for Component-Based Systems. (TQACBS06)*. 2006

[44] Sun Microsystems. *API Enhancements to the JavaBeans Component API in v1.4*. Sun Microsystems Inc, java.sun.com/j2se/1.4.2/docs/guide/beans/changes14.html, 2002.

[45] Sun Microsystems. *Java Platform Debugger Architecture (JPDA)*. Sun Microsystems Inc., http://java.sun.com/products/jpda/index.jsp.

[46] Stahl, T. and Völter, M. *Model-Driven Software Development*. Wiley, 2006.

[47] Sysoft. *Animation of UML Sequence Diagrams" - Amarcos*. Sysoft, http://www.sysoft-fr.com/en/Amarcos/ams-uml.asp.

[48] Thoms, C. and Holzer, B. Codegenerierung mit dem openArchitectureWare Generator 3.0 - The Next Generation. *javamagazin 07/2005*, 2005.

[49] W3C. *Extensible Stylesheet Language (XSL) Version 1.0*. W3C Recommendation, www.w3.org, 2001.

[50] W3C. *Scalable Vector Graphics (SVG) Version 1.1 Specification*, W3C Recommendation, www.w3.org, 2003

[51] W3C. *Java Language Binding for the SVG Document Object Model*. W3C Recommendation, www.w3.org. 2003.

[52] W3C. *ECMAScript Language Binding for SVG, W3C Recommendation*, www.w3.org, 2003.

[53] W3C. *XSL Transformations (XSLT) Version 1.0 Specification*. W3C Recommendation, www.w3.org, 1999