Reverse engineering Java Enterprise Applications in Eclipse

Raffael Krebs, Fabrizio Perin

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract. Java Enterprise Applications (JEAs) are complex systems composed of various technologies and programming languages other than Java, such as XML or SQL. In this heterogenous context, information is spread across various components and languages and the interactions between different application elements could be hidden. On the other hand, existing reverse engineering and quality assurance techniques are unable to analyze JEAs since they are focused on specific components or languages.
In this paper we present our approach to enhance the Eclipse IDE to analyze these complex systems. We extended a meta-model for Object-Oriented systems to accommodate the heterogeneous nature of JEAs. By modeling JEAs we can support different kinds of analysis based on software visualizations and software metrics.
We implemented an Eclipse plug-in to analyze JEAs called Vera to make this solution available in one of the most largely used IDE. The integration with the development environment makes the developers keep focused on a single tool instead of relying on several once.
We demonstrate Vera by implementing a well known polymetric visualization, the System Complexity view, and an improved version of the Transaction Flow visualization. The latter shows what can be done by integrating information spread across different sources. We also present a browser that provides an analytical view of the code.

Keywords: Reverse engineering, Enterprise Applications, Eclipse plug-in, Visualizations.

1 Introduction

Since Java 2 Platform Enterprise Edition (J2EE) was introduced in 1999 it has become one of the standard technologies for enterprise application development. J2EE applications are complex systems composed of various technologies that in turn rely on languages other than Java, such as XML or SQL.

Given the complexity of these applications, the need to reverse engineer them to support further development becomes critical.

Over the years a lot of software analysis techniques and tools have been developed but all of them are focused on a single language.

In the heterogenous context of JEAs applying these techniques would result in losing the overview on the system or missing some important aspects about the interaction among its multiple components. Moreover some tools for software analysis are not part of the development environment but they are stand-alone applications. Therefore a developer have to perform a context switch every time he wants to use one of these tools. The context switch can result in a loss of focus on the problem to investigate and it is time consuming.

In this paper we present our approach to support the analysis of Java enterprise applications (JEAs) within Eclipse, one of the most widely used Java development environments. To achieve this goal we develop an Eclipse plug-in called VERA. The plug-in provides a meta-model for JEAs[1] that can be extended with more entities to represent a wider range of JEAs components. This meta-model is based on *FAMIX* [2], a language independent meta-model that describes the static structure of object-oriented software systems. VERA implements two software visualizations, one of them is specific to analyze transaction scope in JEAs. The visualizations of VERA are implemented using Draw2D[1], a lightweight toolkit for displaying graphical components on a Standard Widget Toolkit[2] (SWT) canvas. By using this toolkit it is possible to craft more software visualizations and attach them to the proper extension points provided by VERA. With the purpose to provide a different way to browse and to inspect the code, VERA implements a browser called *model browser* that highlights information such as relationships among software components or software metrics computed on the software entity selected by the user.

We demonstrate the efficiency of VERA by implementing a polymetric visualization known as System Complexity[3] and an improved version of the Transaction Flow visualization[4]. We also implement a model browser that provides an analytical view of the code to analyze. VERA is available on-line[3] and can be install like any other Eclipse plug-in.

The rest of the paper is structured as follows: In section 2 we relate our approach to previous work and we present the problem. In section 3 we present VERA and we describe implementation details of the tool. One of the most important requirement of VERA is the extensibility thus is section 4 we describe how to extend VERA and enable the analysis of Java annotations. Finally in section 5 we summarize our results, we describe the future work and we conclude.

2 Problem

This section lists some existing software analysis tools and we present the problem VERA is meant to solve.

Moose[5] is an extensive platform for software and data analysis developed in Smalltalk. It includes various utilities and relies on several tools ranging from importing and parsing data, to modeling, to scripting software visualizations. One

[1] `http://www.eclipse.org/gef/draw2d/index.php`
[2] `http://www.eclipse.org/swt/`
[3] `http://scg.unibe.ch/download/Vera/`

core element of Moose is *FAMIX*[2], a language independent meta-model that describes the static structure of object-oriented software systems. By using tools such are VerveineJ[4] or inFusion (a newer version of iPlasma[6]), it is possible to instantiate a *FAMIX* model from Java source code and then analyze it with Moose. One major drawback from the viewpoint of a Java developer is that Moose is not integrated into any Java IDE. The context switch between Moose and the Java IDE during the development process is distracting and the user can lose the focus on the problem he meant to investigate. By integrating our analysis tool into the development environment we intend to overcome the problems induced by the context switch. Eclipse is a quite widely used Java development environment and therefore well suited to getting software analysis closer to the developer. VERA, as well as Moose, uses the FAMIX model and therefore is potentially interoperable with Moose and other similar tools.

Softwarenaut[7] is a static analysis tool that supports architecture recovery through visualization and interactive exploration. Its focus is to provide a meaningful top-down code exploration. The user can increase or decrease the level of detail from a coarse-grained to a very fine-grained view of the system. This kind of code exploration is not VERA's main focus. Instead, VERA is shiped with a set of immutable interactive visualizations which allow the user to visualize software entities and to jump to their definition in the source code. VERA's structure ease the creation of new software visualizations to support the exploration and the analysis of the code.

SHriMP Views[8] is a stand-alone Java application that provides flexible visualizations for any type of hierarchically organized information. The user can easily combine visualizations at different levels of detail. Visualizations are not meant to detect certain software flaws but to perform top-down exploration of the code like in Softwarenaut. SHriMP it is not integrated into a Java development environment, so it also suffers of the same context switching problems of Moose. There is an Eclipse plug-in called *Creole* that is an attempt of integrating ShriMP within the Eclipse IDE. However this plug-in is no longer maintained and constantly crashes with recent versions of Eclipse.

Architexa[5] is a commercial Eclipse plug-in that allows the user to create UML diagrams of source code in an explorative manner and it is focused on a smooth user experience. Since Architexa is a commercial product the users cannot extend it to fit their own needs, their possibilities are limited to what that company thinks is useful. VERA has been developed with the purpose of being extensible. New visualizations, metrics or queries can be created to highlight different software aspects since VERA relies on an abstract representation of the code.

inCode[9] is an Eclipse plug-in that supports the user in software quality assessment. The most relevant feature of inCode are: On-the-fly detection of design flaws, automated refactoring for correcting the flaws, architectural assessment, interactive code visualizations *etc.* inCode uses JMondrian[10] to draw the

[4] `http://www.moosetechnology.org/tools/verveinej`
[5] `http://www.architexa.com/index_c.php`

software visualizations that are interactive but cannot be customized and, unlike VERA, it focuses purely on Java code. The functionalities provided by inCode were originally developed for *inFusion* (formerly iPlasma[6]), another stand-alone analysis program for Java, C# and C++.

X-Ray[6] is another Eclise plug-in which provides some source code visualizations. X-Ray is open-source, so it is possible for other plug-ins to use its model, *e. g.,* plug-ins like *Citilyzer*[6] or *Proximity Alert*[6] rely on X-Ray's model to create their own visualizations. X-Ray differs from VERA since it is only focus on Java code. VERA on the other hand relies on a meta-model that can represent all the parts composing a JEA and, by doing so, it enables crosscutting analysis that include not only the Java code *e. g.,*VERA's first main visualization, the Transaction Flow view, requires information from both Java source files and an XML file.

MoDisco[7] is an extensible framework to develop model-driven tools to support use-cases of existing software modernization. One main difference between VERA and MoDisco is the technology used to support the analysis of heterogenous applications. VERA is based on FAME while MoDisco relies on Ecore. FAME is a superset of a subset of EMOF that has feature comparable to Ecore since also Ecore is based on EMOF. Anyhow, FAME is simpler than Ecore and so more clear to understand and easier to extend. Since FAME is self describing it is possible to exchange model and meta-model definitions using the same mechanism. Therefore we choose FAME as meta-meta-model because it is possible to share information with the tools developed in the very active Moose ecosystem. VERA's final purpose is to be a simple tool to support code understanding and related activities through software visualizations and software metrics. MoDisco seems to have a larger scope since it tries to support the full process of software analyses and modernization.

We list the main features of the tools presented in Table 1. For each tool we also gives a qualitative estimation about how good a feature is implemented. Regarding MoDisco we based our evaluation based just on the documentation we found on-line.

3 Vera

In this section we describe our solution and we expose the implementation of VERA. We describe the overall architecture of our tool and then we detail the single components composing it.

VERA provides a base to analyze JEAs within Eclipse, it support the use of software visualizations and it is extensible. The Transaction Flow visualization implemented in VERA is an example of the capabilities of VERA to deal with JEAs. Prior to EJB 3.0, the JEA transaction attributes had to be specified in an XML configuration file. So the information was spread between a Java class and a XML file. By relying on a unified meta-model we can integrate information

[6] `http://xray.inf.usi.ch/`
[7] `http://eclipse.org/MoDisco/`

	SW Visualizations	IDE integration	Extensible	JEAs
Moose	++		++	+
Softwarenaut	++			
Shrimp / Creole	++			
inFusion / inCode	+	++		
Architexa	++	++		
X-Ray	++	++	+	
MoDisco	?	++	+	+
VERA	+	++	+	+

Table 1. Features of the most relevant applications that support software understanding and analysis. (+: decent, ++: good)

from different sources and enable new kind of analysis. The integration with the development environment overcame the problem related to the context switching to another tool and keep the developers focus on the problem they intend to solve. VERA can be extended by adding to its meta-model new entities and by extending information importers. By using Draw2D to implement software visualizations, VERA gives the possibility to the users to craft new custom visualizations on the software model.

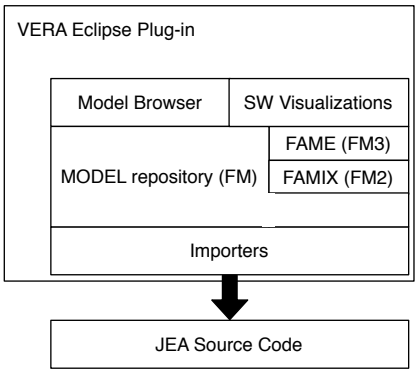

Fig. 1. Architecture of VERA

Figure 1 illustrates VERA's architecture which is composed of three main parts: the *application model*, the *importers* and the *analysis tools*. The *application model* contains the software model generated by the importers as well as the FAMIX meta-model and the Fame meta-meta-model. The *importers* extract information from the source code or other sources and instantiate the model of the application accordingly. The *analysis tools* consist for now in two software

visualizations, the System Complexity view and the Transaction Flow view, and in a model browser that shows analytical information about the software entities.

3.1 The Application Model

The application model consists of three components: the FAMIX Model (FM), the FAMIX Meta-Model (FM2) and the Fame Meta-Meta-Model (FM3). The FM is an instance of the FM2 and it contains all the objects representing the source code. It only exists at runtime, *i. e.,* after the importers are done parsing the source code. The FM is stored in a *model repository* which is fully accessible from the other layers. The FM objects are also called entities.

FM2 is a language independent meta-model that describes the static structure of object-oriented software systems. VERA enriches this meta-model to represent not only object-oriented code but also other aspects peculiar to JEAs such as relational databases or EJBs. The classes implementing FM2 define getter and setter methods for all the properties of the model objects. In addition they define helper methods, some of which are specific for the Java source code. On top of the FM2 classes we introduced a layer of interfaces with the purpose of separating the part of FAMIX that models OO languages from the one that models Java language specific characteristics. These FM2 interfaces define also the methods related to FAMIX properties. In the model repository together with the model entities is stored a meta-description for all the FAMIX types. These meta-descriptions are built at runtime based on annotations on the FM2 interfaces. The meta-descriptions are instances of the FM3 classes which are, together with the annotation types, provided by a Fame implementation in Java[11].

3.2 Importers

There are two importers provided by VERA: The first is a Java importer that uses the Eclipse Java parser to generate an Abstract Syntax Tree (AST) of the Java code. The AST is then visited by the importer to create the entities composing the FAMIX Model. These entities are finally stored into the model repository. The second importer is a parser for EJB deployment descriptors which are XML files. The importer extracts information about the method's transaction attributes and it enriches the Java model with that information.

VERA defines an `importers` extension-point where other importers can hook and by doing so they can be used during the import phase of the project to analyze. Extending plug-ins can alternatively hook their importers during the AST visiting phase. Importer extensions can also specify the order in which the importers must be executed and which kind of dependencies there are among different importers *e. g.,* it is possible to specify that the importer for Java code must run before the importer for the deployment-descriptors.

3.3 Visualizers

One way of bringing the model closer to the user is through software visualizations. VERA adds its own view to the Eclipse workbench for showing visualizations. Therefore, VERA extends the *ViewPart* class of Eclipse and relies on a number of so called visualizers for the rendering. Further visualizers can be added through the VERA's `visualizers` extension-point. So far, VERA provides two visualizations, System Complexity[3] and Transaction Flow[4]. To create the software visualizations we used Draw2D toolkit. In an earlier development stage we also evaluated JMondrian[10], the Java implementation of the Mondrian framework[12] to script visualization in Smalltalk. This last framework is less expressive and powerful than Draw2D and it is not constantly maintained. Anyhow, the model of JMondrian is simpler than the one of Draw2D so JMondrian is a good alternative to create very simple visualizations. Draw2D is also part of other frameworks like Zest[8] or GEF[8] that can help the users to create more complex visualizations by adding more layouts and by handling interactions generated by mouse and keyboard. We are not aware of other frameworks that can have the same potentiality of Draw2D, however, VERA is made in such a way that its is actually possible to use different visualization frameworks.

3.4 The Model Browser

Another way to expose the information contained into the FM is through the VERA's model browser. Every time a user selects one entity either in a visualization provided by VERA or in the source code editor, the model browser shows the property of that entity *e. g.,* by selecting a class into the Transaction Flow visualization the browser will display software metrics like the number of methods (NOM) or the number of attributes (NOA) of the selected class as well as more standard data like the class name. The model browser entirely relies on the Fame annotations within the meta-model interfaces described earlier.

3.5 User interface

VERA adds two views to the Eclipse workbench: the visualization view and the model browser view. These views can be enabled and disabled and it is possible to add them to the standard Java perspective.

The standard or the custom visualizations of VERA are displayed within the *visualizations view*. To facilitate the user in adding custom visualizations, VERA defines a `visualizers` extension point. For each visualization hooked to the extension point, VERA automatically creates a button in the view's toolbar to make the visualization available. Figure 2(b) shows the visualization panel of VERA displaying the System Complexity visualization[3] of the current project's model.

The VERA *model browser* view, shown in Figure 2(a), exposes information of a single model object contained into the FM. On the left side of the model

[8] `http://www.eclipse.org/gef/zest/index.php`

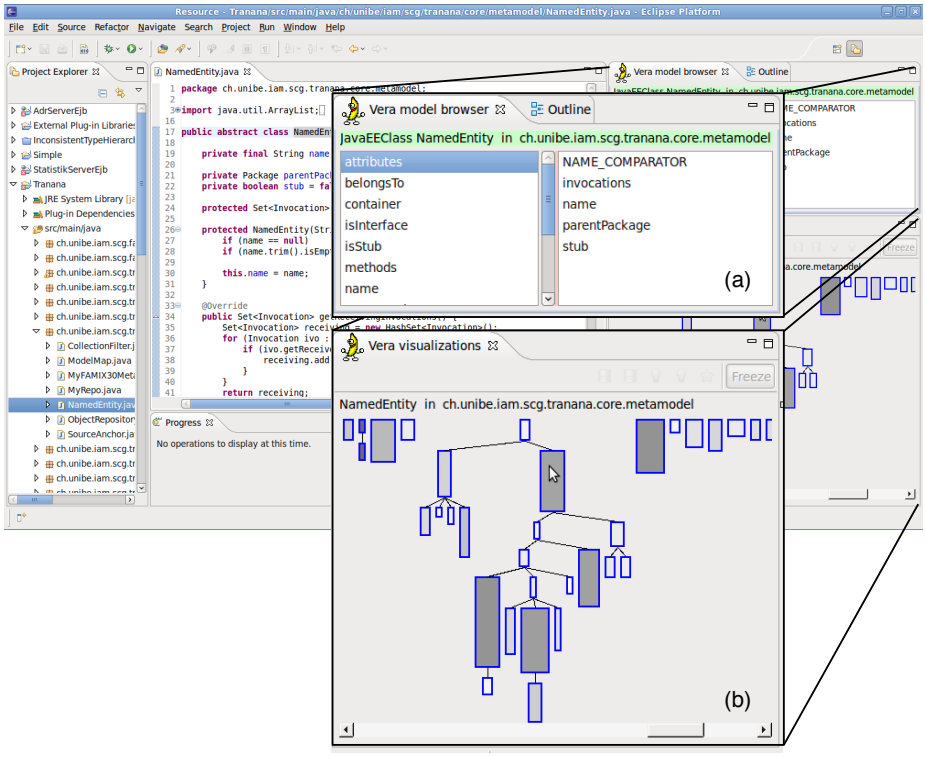

Fig. 2. Screenshot of VERA's GUI

browser all properties of the object are listed. By selecting one of these properties, its value is displayed on the right side of the browser. Multivalued properties are displayed as a list. If the selected property value is a Java entity itself, both the model browser view and the Java editor will jump to that entity. Likewise, whenever a Java entity is selected in the workspace VERA will show the entity's properties into the model browser.

4 Extension Example

This section describes an example of how VERA can be extended. In particular, we present a scenario in which we want to enable VERA to analyze Java annotations.

Java annotations are not part of the program itself even though they can be added to the Java source code[9]. Java annotations carry extra information inside the Java code and they are contained into Java files. By extending VERA we will allow the user to collect information about the Java annotations defined and

[9] `http://download.oracle.com/javase/tutorial/java/javaOO/annotations.html`

used into an application and to analyze them. To extend VERA it is necessary to write a new Eclipse plug-in as described in the following sub-sections.

4.1 Extend the meta-model

The first step is to extend the meta-model of VERA. For this scenario we decided to add two entities: `AnnotationType` and `AnnotationInstance`. As the names suggest the first entity represent the Java annotation types, the second model the Java annotation instances spread around the code. We choose the FAMIX entity type `NamedEntity` as the superclass of both classes. Together with the two new entities we need to implement two new interfaces `IAnnotationType` and `IAnntotationInstance` which should both extend `INamedEntity`. In the interfaces we need to define the getter methods for the information contained into the new entities *e. g.,* the name of the annotation type, or the list of entities that have been annotated. These methods should be annotated with `@FameProperty`, and the interfaces themselves should be annotated with `@FameDescription` as shown in Listing 1. The annotation types having `@Fame` as prefix are part of the Fame implementation and their instances are used to attach FM3-related meta-information to Java classes. In other words, these annotations are used to populate the meta-repository.

```
1  @FameDescription("AnnotationInstance")
2  interface IAnnotationInstance extends INamedEntity {
3
4      @FameProperty{name="annotatedEntity"}
5      INamedEntity getAnnotatedEntity();
6
7      @FameProperty{name="annotationType", opposite="instances"}
8      IAnnotationType getAnnotationType();
9  }
10
11 @FameDescription("AnnotationType")
12 interface IAnnotationType extends INamedEntity {
13
14     @FameProperty{name="instances", multivalued=true, derived=true,
       opposite="annotationType"}
15     Collection<IAnnotationInstance> getInstances();
16 }
17
18 class AnnotationInstance extends NamedEntity implements
       IAnnotationInstance {
19     // implement the interface methods, add some helper methods
20 }
21
22 class AnnotationType extends NamedEntity implements IAnnotationType {
23     // implement the interface methods, add some helper methods
24 }
```

Listing 1. Definition of the classes and the interfaces for the Java annotation scenario

Given that we have implemented the entities in Listing 1, we can declare a meta-model extension in which we feed the central meta-repository with the two new meta-model classes as shown in Listing 2.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.meta-model">
        <with class="com.example.AnnotationType" />
        <with class="com.example.AnnotationInstance" />
    </extension>
    ...
</plugin>
```

Listing 2. Declaration of the meta-model extension

Now that the meta-model has been modified we can instantiate a model containing also information regarding Java annotations.

4.2 Custom importers

In this scenario we decided to create two importers; one to import the annotation types, another to import the annotations instances. Since we have to importers we need to declare two AST-importer extensions. Both importers will need to consider just Java files. By running the type importer first, the instance importer can verify that every annotation instance, *e. g.,* of a method, corresponds to a known annotation type. It is possible to ensure the correct execution order of the importers by specifying that the annotation instance importer should run *after* the annotation type importer as shown in Listing 3.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.importers">
       <AST-importer
             class="com.example.AnnotationTypeImporter"
             id="Java-annotation-type-importer">
       </AST-importer>
       <AST-importer
             class="com.example.AnnotationInstanceImporter"
             id="Java-annotation-importer">
             <dependency after="Java-annotation-type-importer" />
       </AST-importer>
    </extension>
    ...
</plugin>
```

Listing 3. Declaration of the Importers extensions

4.3 Expose the data

By using the importers we can instantiate a model containing the new FAMIX Entities defined earlier. Since all the information within the model is accessible

it is possible to expose this information by creating a new Eclipse view and by choose any kind of model representation we like. Instead, we can use VERA's visualizer infrastructure to create a visualization of the model using Draw2D. By considering our scenario we can implement a software visualization like, for instance, the Annotation Constellation[10].

To achieve this goal we have to implement a class that constructs the visualization using Draw2D starting from the information within the FAMIX model. This should extend the abstract class `Draw2dVisualizer` which handles the task of embedding the Draw2D figure into the VERA's visualization view. We then register our new visualizer by declaring a visualizer extension as shown in Listing 4.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.visualizers">
        <visualizer
            title="Annotations Constellation"
            id="com.example.visualizers.annotations"
            class="com.example.AnnotationConstellation"
            icon="icons/foo.png" />
    </extension>
    ...
</plugin>
```

Listing 4. Declaration of the visualizations extensions

5 Conclusion

Java Enterprise Applications (JEAs) are complex systems that integrate various technologies which in turn rely on different programming languages other than Java, such as XML or SQL. Modern software analysis tools and techniques are not able to capture important aspects of JEAs because of their heterogenous nature. Few attempts have been made to provide instruments to support the analysis of Enterprise Applications and only one of them is available within a development environment.

In this paper we presented VERA, an Eclipse plug-in that aims to support the analysis of JEAs. VERA provides a meta-model for enterprise applications that can be extended with more entities to represent a wider range of JEAs components. By using Draw2D, a lightweight toolkit for displaying graphical components on an SWT Canvas, VERA provides two default software visualizations and gives the possibility to craft more of them. The model browser of VERA highlights information such as relationships with other components or software metrics computed on the software entity selected by the user.

In the future we plan to make VERA even easier to extend: We intend to simplify as much as possible the process of adding new components to the meta-model. We plan to integrate a plug-in for scripting visualizations in JRuby. We

[10] `http://www.themoosebook.org/book/externals/visualizations/annotation-constellation`

want to simplify the way in which additional information can be displayed in the model browser.

Acknowledgments - We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 – Sept. 2012). We would also want to thank Oscar Nierstrasz and Jorge Ressia for their comments on this paper and their support on this project.

References

1. Perin, F.: Enabling the evolution of J2EE applications through reverse engineering and quality assurance. In: Proceedings of the PhD Symposium at the Working Conference on Reverse Engineering (WCRE 2009), IEEE Computer Society Press (October 2009) 291–294
2. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00), IEEE Computer Society Press (2000) 157–167
3. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer-Verlag (2006)
4. Perin, F., Gîrba, T., Nierstrasz, O.: Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In: Proceedings of International Conference on Software Maintenance 2010. (September 2010)
5. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), New York NY, ACM Press (2005) 1–10 Invited paper.
6. Marinescu, C., Marinescu, R., Mihancea, P., Ratiu, D., Wettel, R.: iPlasma: An integrated platform for quality assessment of object-oriented design. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005). (2005) 77–80 Tool demo.
7. Lungu, M., Lanza, M.: Softwarenaut: Exploring hierarchical system decompositions. In: Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering), Los Alamitos CA, IEEE Computer Society Press (2006) 351–354
8. Storey, M.A., Best, C., Michaud, J.: SHriMP Views: An interactive and customizable environment for software exploration. In: Proceedings of International Workshop on Program Comprehension (IWPC '2001). (2001)
9. inCode: inCode — eclipse plugin for code analysis (2009) http://www.intooitus.com/inCode.html.
10. JMondrian: JMondrian — java implementation of the mondrian information visualization framework (2009) http://loose.upt.ro/iplasma/jmondrian.html.
11. Kuhn, A., Verwaest, T.: FAME, a polyglot library for metamodeling at runtime. In: Workshop on Models at Runtime. (2008) 57–66
12. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: An agile visualization framework. In: ACM Symposium on Software Visualization (SoftVis'06), New York, NY, USA, ACM Press (2006) 135–144