Vera

An extensible Eclipse Plug-In for Java Enterprise Application Analysis

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Raffael Krebs

2012

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz

Acknowledgements

First of all I would like to thank my mentor Fabrizio Perin for his support. And thanks to professor Oscar Nierstrasz for giving me the possibility of writing my thesis under his supervision. Further I am very grateful for the valuable advice by other members of the SCG research group, namely Niko Schwarz, Toon Verwaest, Erwann Wernli, Mircea Lungu and Jorge Ressia.

The quality of this text has been greatly improved thanks to the painfully thorough feedback of Niko Schwarz, Toon Verwaest, Sandro De Zanet and Fabrizio Perin.

These past few months have been a personal challenge. My parents Elisabeth and Roland helped me a lot by pushing me forward in times when I lost motivation. My dear friends Sandro De Zanet, Toon Verwaest, Laura Sánchez Serrano, Miriam Hutter, Maria Iakovleva, Sara Mahnig, Florian Gysin and Martti Nirkko all had their very own ways of supporting me.

Thank you all, I owe you one!

Abstract

Java Enterprise Applications (JEAs) are not Java applications.
Instead they are complex systems which integrate various technologies and programming languages such as XML or SQL, also including Java source code. Many tools exist for reverse engineering and quality assurance on purely object-oriented program code. JEAs are heterogeneous in that they are composed using multiple languages, thus classical object-only approaches are not expressive enough.
In this thesis, we present VERA, a framework that supports JEA analysis. We develop VERA as a plug-in to Eclipse, one of the most widely used Java development environments. The tool relies on an extended version of the FAMIX meta-model that accommodates the heterogeneous nature of JEAs. Our FAMIX implementation supports multiple kinds of analysis on JEAs. VERA features extension-points that make it easy to add custom analyses, with a focus on visualizations.
We demonstrate the flexibility of VERA by implementing a well-known polymetric visualization called *System Complexity* and an improved version of the *Transaction Flow* visualization. We also present a model browser that provides an analytical view of the code at hand.

Contents

List of Figures

List of Tables

1 Introduction

Software analysis is the art of answering the question

> *"What does my program actually do?"*

and comparing the answer to the functional and structural requirements for that program. This includes finding the cause of a discovered bug, predicting reliability, identifying components of a system and their interactions, and calculating manager-proof quality measures. While such analyses can be done partly thorough manual inspection of the program code by a developer, humans often miss important details, especially in highly technical, low-level tasks. Where quality indicators and requirements can be formalized, they can be checked by computers. Accordingly, Ostermann [17] defines software analysis as

> *"Systematic, computer-aided examination of software to determine whether and why a (desired or undesired) property holds."*

1.1 Heterogeneous Sources

In this thesis, we have a special interest in Java Enterprise Applications (JEAs). Analyzing JEAs poses a particular challenge because they are built using multiple languages.

A typical J2EE application consists not only of Java source code, but is composed using multiple technologies which entail various languages and file types. A JEA typically provides a web interface written in, *e.g.*, JSP or (X)HTML. Furthermore, a database is commonly used to store persistent data —

the connection to which might be configured in an XML file, and the accesses to which are written in SQL. Additional XML or `.properties` files can contain configurations for frameworks that are used, as well as for the server on which the application will run (`ejb-jar.xml`, `log4j.xml`, etc.). The application may provide several web services which are also declared in the `ejb-jar.xml`. Finally, the application is probably built using Ant or Maven, which require yet another configuration file. Furthermore, the number of available technologies that can be used in a JEA grows over time.

An analysis tool can master that multi-language challenge by allowing its users to extend the tool, add support for new languages and implement new analyses.

1.2 Extensibility

An analysis tool should of course analyze properties that are relevant to the program under consideration. Analyses of properties that apply to several programs can be condensed into a general tool. For example, a compiler analyses properties that apply to all programs written in a certain language, such as type safety and liveness of variables. An example for a more elaborate general-purpose analysis is the detection of potential design flaws.

Whereas general tools can cover language or technology specific properties, programs can also have unique requirements. For example, when a program's design defines constraints for communication between the program's components (*e.g.*, when implementing the MVC pattern[1]), the developers certainly are interested in whether those constraints are met. An analysis for that aspect of that program most likely cannot be applied to another program without modification, because the communication constraints are unique to the first program.

So besides general-purpose analyses, a tool should offer a way to add custom analyses or adapt existing ones in order to address the very specific requirements of the program under consideration. We refer to this characteristic of a tool as being *extensible*.

1.3 Integration

Software comprehension is an important aspect of software analysis. Software comprehension describes the cognitive process of understanding what a program does and how it does it. Software developers, before creating new features or improving existing ones, actually spend a fair share of their time reading source code and understanding it [2, 3, 12, 21].

Software analysis tools can aid software comprehension; for example by giving a high-level view of the program's components, or by providing convenient means of browsing the program code. To aid software comprehension most efficiently, a tool should be tightly integrated with the development environment. Tools that are external to the develompent environment require the user to leave that environment, find and start the external tool, recall how to use the tool and wait for the program code being imported and analyzed. Even if the process is rather simple, it takes some time and it also forces the developer to perform

[1] Model-View-Controller, see for example `http://de.wikipedia.org/wiki/Model_View_Controller`

a cognitive *context switch*. The developer might lose focus on the problem to investigate, or even refrain from using the tool at all.

1.4 Vera

As an approach to extensible, cross-technology JEA analysis without context switch, we develop an Eclipse plug-in called VERA.

By integrating our tool VERA with the Eclipse IDE we avoid the *context switch* and facilitate installation, thus reducing cognitive and technical barriers that prevent its usage. For *cross-technology* analysis, we use *FAMIX* [25], a language independent meta-model that describes the static structure of object-oriented software systems. We adapt FAMIX to our specific context, *i.e.*, to accommodate the heterogeneous nature of JEAs [18]. We also use Fame [9], which allows us to programmatically reason about the FAMIX meta-model itself. For *extensibility* we make VERA's model visible and we provide extension-points where other Eclipse plug-ins can hook in and include their own software visualizations or amend the model.

Now, why did we chose the name "Vera"? As we lay a base for software comprehension and analysis tools, we help to bring forward the truth. Therefore, the name is an abbreviation of the word "veracity".

1.5 Outline

The remainder of this thesis is structured as follows. In Chapter 2 we introduce terms and concepts that are essential to the understanding of our work. Chapter 3 showcases VERA's user interface and explains its usage. Chapter 4 explains how VERA works internally. An example of how to extend VERA can be found in Chapter 5. Chapter 6 mentions related work. Then we briefly discuss a few aspects of our solution in Chapter 7, before we wrap up and list future work in Chapter 8.

2 Background

In this chapter, we explain the domain in which VERA operates as well as the technologies on which VERA is built. Based thereon, we also explain what VERA does in more detail.

2.1 Types of Software Analysis

There are three types of automated software analysis: static analysis, dynamic analysis and formal verification.

Static software analysis works on the source code of a program. Many compilers perform static analyses such as static type checking, liveness analysis of variables and bytecode optimization. These analyses need to be highly *reliable* (safe, sound) since they are automatically acted upon. Another family of analyses just produces *informative* hints, based on which humans decide what to do. For example, some tools search the program code for structural anti-patterns, which are likely to be the source of maintainability issues, or compute software metrics.

Dynamic software analysis focuses on the runtime behaviour of a program. Here, we also have both reliable and informative tools. Just-in-time compilers for example automatically optimize the parts of a program that are used frequently at runtime. Profilers help to identify the source of performance issues, *i.e.*, which parts of a program use up a lot of processor cycles and/or memory. Looking at the runtime interaction between different parts of the program can help to reveal dependencies, isolated components, features, or candidates for dead code. With runtime data, one can also check whether design constraints

are met. Software testing, in which a defined set of inputs is compared to the expected outputs, can also be considered dynamic software analysis, as well as post-mortem analysis, in which specific information about program execution is logged at runtime and analyzed after program termination.

The last and least frequently used type of software analysis is *formal verification*, where a mathematical model of the whole program is built. Such a model allows for formal proof that a program conforms to its specification. Examples of formal models of programs are finite state machines, Petri nets, process algebra, and Hoare logic.

2.2 Software Understanding

An important research question is: How can tools support the developers in building a mental model of a piece of software more efficiently?

Storey [23] compiles different strategies that developers use to explore the source code of a program. The *bottom-up* (control-flow) strategy consists of reading code and then mentally chunking together what is understood and thus forming a meaningful, higher-level abstraction. The *top-down* (functional) strategy maps knowledge about the program domain to the source code. The developer starts with a global hypothesis about the program which is then refined into a hierarchy of secondary hypotheses while browsing the program code. The *knowledge-based* (guessing) strategy leverages the developer's current understanding of the program and programming skills for asking a question and conjecturing an answer. The conjecture is then verified or falsified by looking at program code and documentation. Mayrhauser [26] found that software developers leverage all three strategies, frequently switching between them, to build up a mental model of the software concurrently at different levels of abstraction.

Most integrated development environments (IDEs) provide basic tools for browsing program code (file browser, search, jumping to the definition of methods, showing documentation in pop-ups, *etc.*), which support software comprehension to a certain degree. The top-down strategy can be specifically supported with visual abstractions of the program components and their interaction, which we refer to as *software visualizations*. Visualizations also have the special benefit that they allow to persist a mental model for documentary purposes.

Many tools have been developed that create visualizations of programs[1]. However, only few of them are integrated into an IDE. Such external tools require the developer to perform a cognitive context switch. That means getting out of the familiar world of the development environment, finding/starting the analysis tool, diving in, firing the analysis and waiting for the results. That context switch can result in a loss of focus on the problem to investigate. DeMarco and Lister [4] observed that mental interruptions have a severe impact on developer performance and can even lead to frustration. The impact is most severe when the interruption occurs during high mental workload. So in the times when the developer could best use a tool, *i.e.* to offload cognitive pressure to the tool, she should not use any that require a context switch. Salvucci and Bogunovich [20] further observed that people tend to avoid interruptions in times of high workload and defer them until the mental stress level drops. So, the mere prospect of an interruption will

[1] Webarchive: `http://ruthmalan.com/ArchitectureResourcesLinks/VisualizationInSoftware.htm`

cause a tool to not be used when most helpful. By the time the mental workload drops, the problem might already be solved. To sum up, tools that aim at aiding program comprehension should try to avoid the context switch.

We support the software developer by integrating our tool VERA with the Eclipse Java IDE. Together with the Java and Java Enterprise development tools that are already available for that IDE, VERA supports all three software comprehension strategies, as well as seamless switching between them. Integration with the IDE also keeps the cognitive interruption at a minimum.

2.3 Java Enterprise Applications

Since the Java 2 Platform Enterprise Edition (J2EE) was introduced in 1999, it has become one of the standard technologies for enterprise application development.

Maintaining a JEA, consisting of potentially many thousands of lines of Java code, a huge number of web page definitions and several kinds of configuration files, is clearly a challenge. And so is writing an analysis tool for JEAs. Computing analyses "just" for one programming language, namely Java, is not good enough. Every technology that is used in a JEA adds its own semantics, *i.e.*, adds new properties that an analysis tool should be able to compute. When multiple technologies are combined in the same application, that introduces even more properties to analyze, properties that describe the interaction of the different technologies or their collective behaviour. Furthermore, there may be different ways how a JEA can combine a given set of technologies, requiring different ways of computing cross-technology properties. A tool can support a number of frequently used technologies, but not every single one of them, let alone all the possible combinations.

Another approach is to develop multiple separate tools, each focused on analyzing a single aspect. With this approach, a new analysis tool has to be developed for every new aspect of interaction between the multiple components of a JEA. Creating a new analysis tool from scratch for every aspect is not very appealing, though. Also, specialist tools cannot provide a general overview of the system.

We envision a common base for analysis tools, and that the tools can reuse each other's analysis results.

2.3.1 Annotations

Java annotations[2] are a language extension introduced in Java 5.0. They are tags that can be added to method definitions, type definitions and variable declarations. Listing 1 shows an example of how annotations are used in Java source code.

Some annotations are predefined and shipped with the Java SDK, *e.g.*, `@Deprecated` and `@Override`. They are evaluated at compile-time and they have no arguments. Additionally, custom annotations can be defined. `@Inject` in Listing 1 exemplifies how an annotation with arguments looks like. The annotation type for a custom annotation has to be defined somewhere on the classpath. Our example annotation `@Inject` is supposed to be evaluated at runtime by some dependency injection mechanism.

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

```
@Deprecated
public class Foo implements IFoo {

   @Inject(provider = "com.example.log.Injector")
   private LoggerService log;

   @Override
   public String getName() {
      return "Foo";
   }

}
```

Listing 1: Examples of annotations.

2.3.2 EJB

Enterprise JavaBeans (EJB) is a technique peculiar to JEAs. Beans are objects a Java application makes available for use by other applications. By using Beans, multiple JEAs can work together; they are the public interface to JEA components. A common example is an implementation of a data access layer, a collection of classes that provide functionality to access a database. A web front-end can access the database through these Beans to modify the database based on user input. JEAs are deployed on specialized Java Enterprise web servers. One of the special components of such a server is the *EJB-container*. It manages the life-cycle of Beans[3] and makes them available to other applications.

EJB also encompasses the notion of *transactions* to make certain functionality atomic, which is often required when chaining together multiple steps of a business process. Transactions also include a notion of *roll-back*, *i.e.* reverting already performed changes when an error occurs during execution, which is important especially with regard to database manipulation. Transactions can be used in one of two different ways: With the first way, the programmer starts, stops and rolls back transactions programmatically through method calls. This is called *bean-managed transaction demarcation*. An alternative way is to configure transaction scopes on a per-method basis. In this case, the EJB-container will manage the transaction boundaries. This is called *container-managed transaction demarcation*.

The EJB-container can manage transaction demarcation on a Bean method in different ways. A strategy can be chosen through the so-called *transaction attribute*, which can have one of the following values:

Required The method must be invoked with a valid transaction context. Its execution is wrapped in a new transaction unless the method is invoked from within another transaction. This is the default value.

RequiresNew The method must be invoked with a *new* transaction context. If invoked inside a transaction, that transaction context is suspended during execution of this method and a new transaction is started.

[3] Instantiation, deletion and persisting, see [1] or `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts9.html`

Supports The method executes correctly regardless of whether it is invoked inside an existing transaction or not. No transaction is created or suspended.

NotSupported The method must not run in a transaction context. If invoked inside a transaction, that transaction context is suspended during execution of this method.

Mandatory Like with Required, must be invoked with a valid transaction context. No new transaction is started. The caller must provide the transaction or else an exception is thrown.

Never Like with NotSupported, must not run in a transaction context. Throws an exception if invoked with a transaction context.

More information on the semantics and configuration of transaction attributes can be found in the EJB specification[4].

Every JEA contains a central configuration file, called the *EJB deployment descriptor*. This is an XML file, `ejb-jar.xml`, which defines how the application is deployed on the server. Through this file, the programmer can register Java classes as Beans, configure which of them let the EJB-container manage transactions, and specify transaction attributes on a per-method (or per-Bean) basis. The deployment descriptor also contains other configuration elements, such as the application's name, security restrictions and resource allocation.

Since version 3.0 of EJB, many of the settings in the `ejb-jar.xml` can also be configured directly in the Java source code through annotations. Changes in annotations require a re-build and re-deploy of the whole application, whereas changes in the `ejb-jar.xml` can be applied directly on the server. Still, annotations are easier to maintain than the XML configuration, since they are located right next to the code they affect. Quickly checking whether there is an annotation on a given method simply is easier than searching for the method's name in some well hidden XML file. However, many old applications will never be migrated. And also in EJB 3.0, transaction management can still be configured in the `ejb-jar.xml`. Therefore the ability to integrate heterogeneous sources remains a key feature for analyzing EJB related aspects of JEAs.

2.4 Eclipse

Eclipse[5] is a development environment platform. It is built by a community of software developers, both individuals and companies. The community builds various projects for the Eclipse platform, including extensible frameworks, tools and runtimes for building, deploying and managing software across the life-cycle. The Eclipse community's most prominent flagship is a fully-fledged Java development environment. The Eclipse Java IDE is well-known and widely used among Java developers. Its capabilities are

[4] `http://java.sun.com/products/ejb/docs.html`
[5] `http://www.eclipse.org/`

comparable to other big Java IDEs such as NetBeans[6], JBuilder[7], IBM Websphere[8] and IntelliJ IDEA[9].

Figure 2.1 gives an idea of what the Eclipse Java IDE looks like.

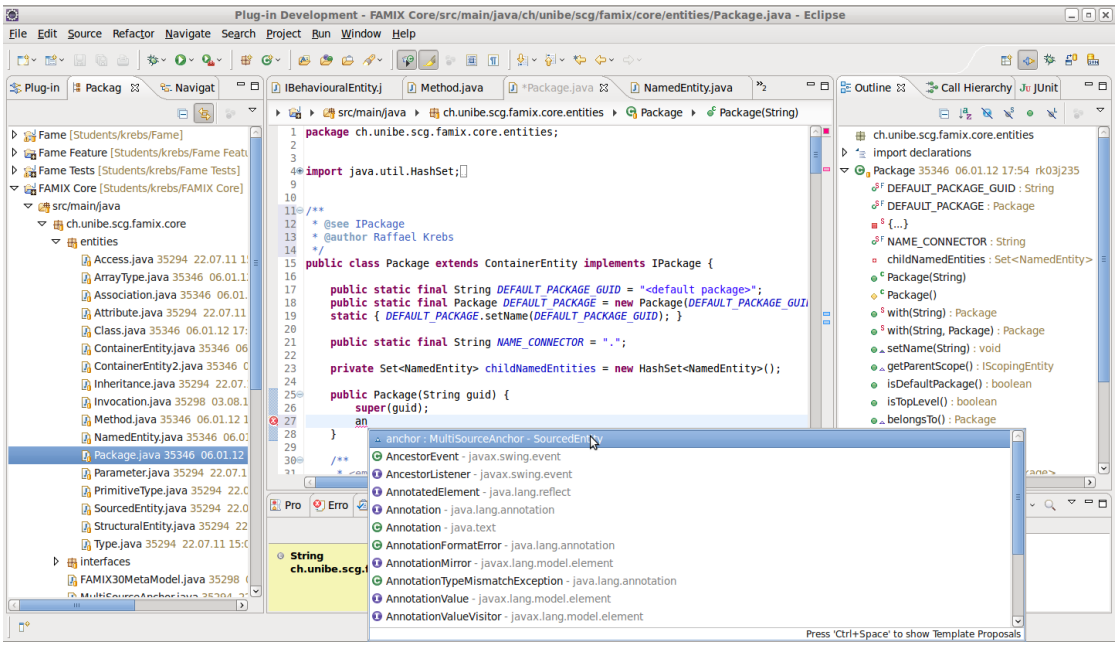

Figure 2.1: A screenshot of a running Eclipse application. The IDE has an editor and several tools, such as a file browser (on the left), a code outline (on the right) and content assist (pop-up).

2.4.1 User Interface

In the following chapters, we use the terms *project* and *view*, which are two basic Eclipse concepts. In this section, we introduce both of them shortly, along with related terms.

Every Eclipse instance runs in a so-called workspace, *i.e.*, a folder which contains both Eclipse meta-data and the source files (program code, configuration files, *etc.*) of the software under development. The source files within a workspace are organized in so-called projects. Usually, every project has its own folder in the workspace, containing project specific meta-data and source files. The `/src/` directory in the project folder contains the Java source code. Source code that lies somewhere in a folder outside the Eclipse workspace can be included in a project, which is called *linking* sources to the project. Linked sources are treated as if they were defined directly inside the project. Other directories inside the project may contain other sources such as HTML files, Ant scripts, *etc.*. A project can be seen as a unit that contains all source files that are required to compile the software. Build tools are usually configured on a

[6] `http://netbeans.org/`
[7] `http://www.embarcadero.com/products/jbuilder`
[8] `http://www.ibm.com/software/awdtools/developer/application/`
[9] `http://www.jetbrains.com/idea/`

per-project basis. Projects can also include other Eclipse projects in their build path, which allows one to split the software into multiple parts.

The user interface of the Eclipse IDE consists of an editor for source files and a set of installed tools, so-called views. Eclipse plug-ins can contribute new views to the user interface. The user can choose which views she wants to display. Obviously, the usefulness of a given view depends on what the user is currently doing. Eclipse embraces this by grouping the views into so-called perspectives. A plug-in can define its own perspective with a set of predefined views in it. The user can customize the UI by adding or removing views from a perspective. For example, the Java perspective is used for coding; the Debug perspective for stepping through the program at runtime; and the SVN perspective for version control. A view can appear on multiple perspectives.

2.4.2 Extension Mechanism

Extensibility in Eclipse is achieved with *plug-ins* that contribute new functionality to each other. Every plug-in can define *extension-points* where other plug-ins can hook in by declaring *extensions*. In that context, we call the plug-in that defines the extension-point the *host plug-in* and the plug-in that declares an extension the *guest plug-in*.

Plug-in contributions commonly are code but can also be just data, *e.g.*, help texts.

The contributions are evaluated by the host plug-in. The guest plug-in cannot change the code of the host plug-in, just add new code. Due to this one-sided coupling, guest plug-ins can be developed independently from the host plug-in without having to worry about breaking the host plug-in.

There is also a notion of *plug-in fragments* to refine plug-ins. A fragment is not a fully-fledged plug-in but instead provides a slight variation of its host plug-in. Fragments are for example used wherever native code is required. The host plug-in can then be offered for download in OS-specific variants. Furthermore, tests are sometimes factored out to a fragment, which is not included in the installable version of the host plug-in at all. This has the advantage of a smaller memory footprint and a faster plug-in load time. Since tests for Eclipse plug-ins can require a complete Eclipse workspace filled with test data, this can make quite a difference.

Before a plug-in can contribute anything to an Eclipse application, it has to be installed there. Every Eclipse application has its individual set of functionalities, depending on which plug-ins are installed. The Eclipse community offers various downloads of Eclipse which differ in the set of plug-ins that are pre-installed. The user can easily install additional plug-ins at will. One of those downloads is the Plug-in Development Environment, the “Eclipse PDE”. This is how Eclipse plug-ins are usually developed: with Eclipse itself.

Additional plug-ins are usually installed from within the running Eclipse application. The Install wizard lists all available plug-ins which can then be installed with only a few clicks. To find available plug-ins, the installer scans a number of so-called *update-sites*, Eclipse’s plug-in repositories. In more recent versions of Eclipse, update-sites are also called *software sites*. A newly downloaded Eclipse application already includes URLs of many update-sites of community members. Official community membership is not required to provide an update-site, though. Everyone who creates a plug-in can publish it on their own

update-site. Internally, an update-site is simply a collection of files that must be served by a web server at a public URL. The files include numerous compressed archives containing the plug-ins as well as meta-data that aids installation. Creation of custom update-sites is facilitated by the Eclipse PDE.

When published on an update-site, plug-ins are commonly wrapped in so-called *features*. An Eclipse feature includes one or more plug-ins that provide a coherent set of new functionalities to the user. It also includes information about the publisher, copyright and licensing (see Appendix C). The same plug-in can be used in multiple features. Finally, features can be grouped into categories for a better overview and searchability.

For technical instructions on the realization of what we discussed here, please consult one of the Eclipse books, *e.g.*, *Contributing to Eclipse* by Gamma and Beck [5], or Eclipse's on-line documentation[10,11].

2.5 Capturing data

In order to carry out automated software analyses, we need a model that represents the application's low-level components. A model is a collection of Java objects that represent application entities and store information about them. The process of creating a model of a program is called *importing*. During the import, the program's source code is read and model objects are instantiated. The extracted information can be properties such as the name of a Java class, or relationships such as class inheritance.

But which properties are available on each object? What is the API of the model? This kind of information resides in a model of the model, the so-called meta-model. The meta-model consists of so-called *entities*, each of which describes the common behaviour of a certain type of model objects.

Finally, one should be able to programmatically reason about the meta-model. For example to list all available properties of a meta-model entity. This can be solved by introducing an even higher level of abstraction: a meta-meta-model.

2.5.1 The Fame Tower

Fame [9] provides these three levels of abstraction by means of a trinity of models, called the "Fame Tower": the FAMIX Model (FM), the FAMIX Meta-Model (FM2) and the Fame Meta-Meta-Model (FM3).

2.5.1.1 The FAMIX Model

The model objects that are created during the import phase are stored in in a so-called *model repository*. Every model object is assigned a unique identifier, which can be used to retrieve that exact object from the repository at a later time (*i.e.*, when performing analyses on the model or collecting data for a visualization).

[10] `http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html`
[11] `http://help.eclipse.org/indigo/index.jsp?nav=/4_1`

2.5.1.2 The FAMIX Meta-Model

The FAMIX meta-model describes the static structure of object-oriented software systems. Its core is language independent; with entity types such as `Class` and `Method` it can describe for example Smalltalk or Java program code. Existing implementations are written in Smalltalk. For our Eclipse plug-in, we needed to re-implement it in Java.

Figure 2.2 shows the core of the FAMIX meta-model as it is implemented in Smalltalk. The FAMIX entities are represented by a hierarchy of classes. Various extensions to the FAMIX Core already exist and further extensions can be easily added. For example, the Java extension to the `Class` entity adds a boolean property `isEnum`.

2.5.1.3 FM3, Fame's Meta-Meta-Model

Every entity in the FAMIX meta-model has a name. And each entity introduces some properties. Most types inherit properties from another type. That kind of information is sometimes required when programming in a more generic fashion. For example when we want to list the names of all properties of a given entity, we would have to collect all properties of its type and supertypes. To that end, information about entities as well as their properties and inheritance relationships needs to be formalized in a model of the meta-model.

FM3 operates with three types of elements: `FM3.Class`, `FM3.Property` and `FM3.Package`. The `FM3.Classes` correspond to the FAMIX entities. Every `FM3.Class` is associated with multiple `FM3.Properties`, respective to the FAMIX entities' properties. Finally, the `FM3.Package` introduces a notion of scoping. A `FM3.Package` is intended to contain all entities of a meta-model, so *all* our entities declare to be part of the "FAMIX" package. FM3 is a rather minimalistic meta-meta-model, yet expressive enough for our needs. Figure 2.3 illustrates the different types of elements in FM3. And when we look at Figure 2.2 from a different angle, it really just shows all `FM3.Classes` of FAMIX `CORE`, alongside their `FM3.Properties`.

```
@FamePackage("FAMIX")
@FameDescription("NamedEntity")
public interface INamedEntity extends ISourcedEntity {

    @FameProperty(name="name")
    public String getName();

    @FameProperty(name="belongsTo", derived=true)
    public IContainerEntity belongsTo();

    @FameProperty(name="parentPackage", opposite="childNamedEntities")
    public IPackage getParentPackage();

}
```

Listing 2: FM3 annotations on a meta-model entity interface (Javadoc omitted for clarity).

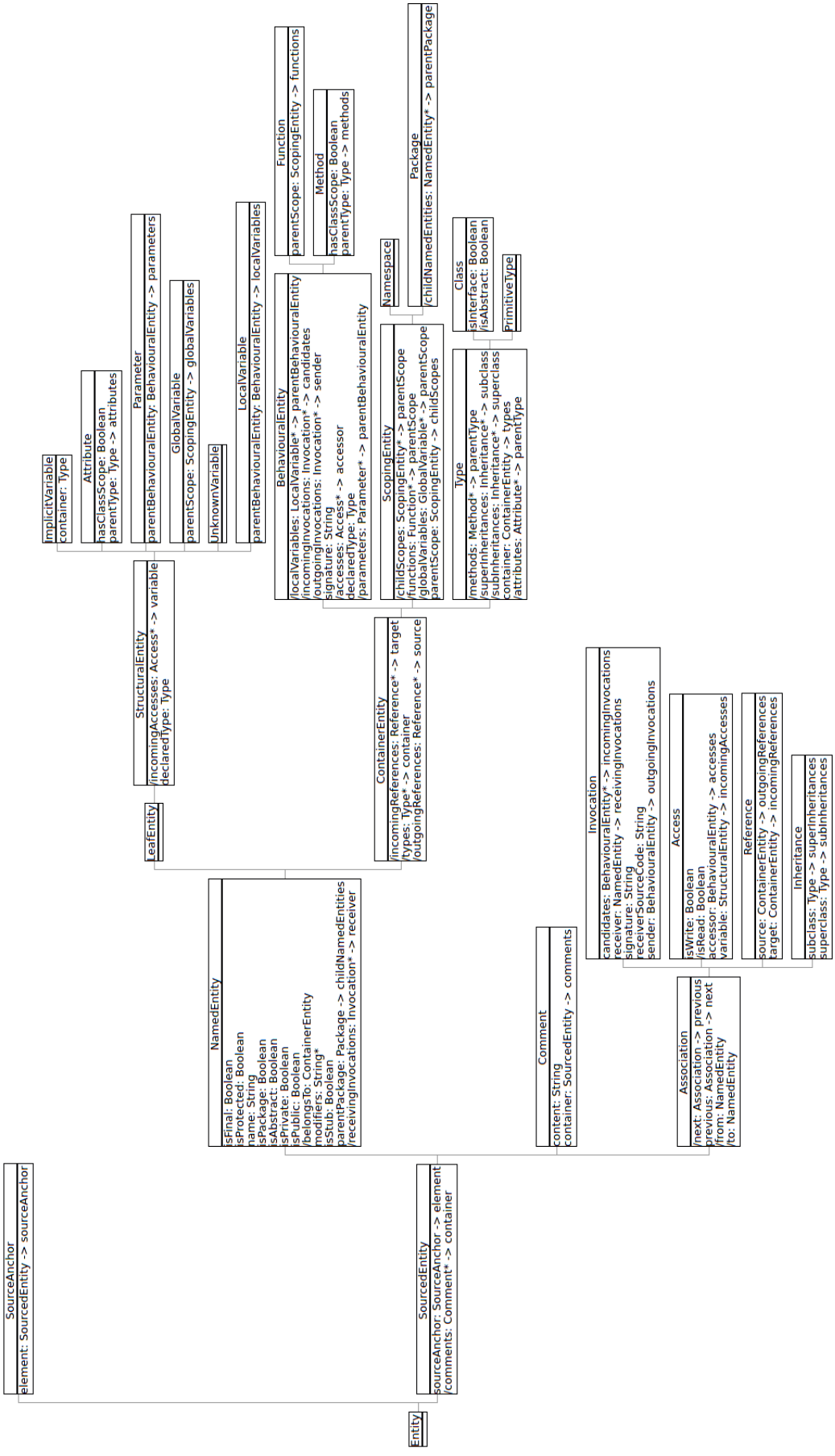

Figure 2.2: FAMIX Core 3.0, the base for Vera's meta-model.

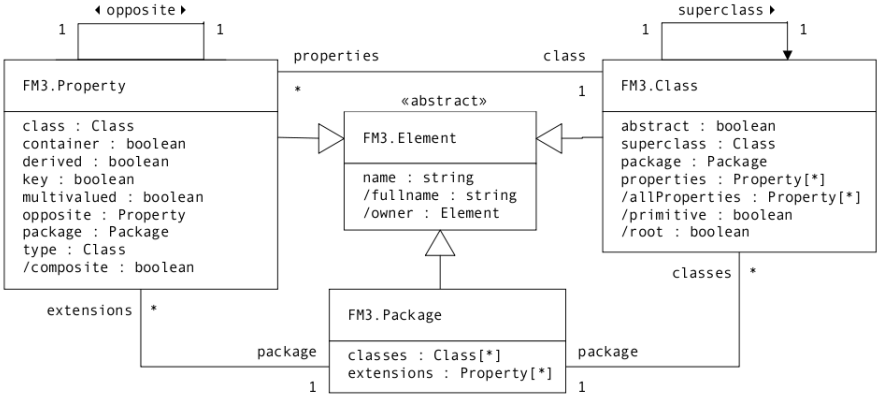

Figure 2.3: FM3, Fame's meta-meta-model.

The Fame tower, next to the model repository, also contains a meta-repository to store information such as which entity provides which properties. It contains so-called *meta-descriptions*, one per meta-model entity. Those meta-descriptions are instances of `FM3.Class`. They are built at runtime, based on Java annotations on the FAMIX interfaces. Listing 2 illustrates the use of the Java annotations that are used to attach meta-information to Java interfaces. The Java annotation types, as well as the meta-description class definition, are provided by Kuhn's Fame implementation [9].

2.6 Summary

In this chapter, we learned about a number of concepts that are important to understand why and how we built VERA.

We have seen that there are three fundamentally different types of software analysis: static analysis, dynamic analysis and formal verification. We also learned that a good software comprehension tool should support top-down, bottom-up and knowledge-based comprehension strategies, as well as fast switching between them; and that it should be integrated into the development environment, since switching to an external tool requires a cognitive context switch, which a has negative impact on both developer productivity and tool acceptance.

In the scope of this thesis, we operate on Java Enterprise Applications, which are a conglomerate of different programming languages and frameworks. Therefore, a software analysis tool that operates on a single programming language is not suited to JEA analysis. Furthermore, the numerous technologies used in JEAs as well as the numerous ways of using/combining those technologies, all require dedicated analysis capabilities — a near impossible task for a single analysis tool. But writing tools from scratch all the time is also cumbersome.

With VERA, we provide a tool that improves Eclipse's software comprehension support. We provide visualizations, which mainly support the top-down strategy, and we allow for fast switching between software comprehension strategies and avoid a context switch by integrating tightly with Eclipse's user

interface. Eclipse also makes installing and updating VERA very easy.

While VERA facilitates software comprehension, its main focus is on analyzing JEAs. VERA supports mostly static software analysis, while post-mortem analysis of a program's runtime behaviour is also possible. We face the diversity of languages and technologies with a language-independent meta-model, FAMIX, and by explicitly making VERA extensible, facilitated by Eclipse's extension mechanism. VERA's extensibility also allows us to customize it to the specific needs of a program set-up or specific quality assurance interests. Thus, VERA serves as a cross-language framework to build software analysis tools upon.

And, to close the circle, extensions to the meta-model also allow for richer ways of navigating the program's source code, thus further improving software comprehension.

3

VERA on the Surface

In this chapter, we explain what VERA does and how it integrates with Eclipse's user interface (UI).

VERA introduces two new Eclipse views which can be added to any perspective: one for the visualizations and one for the model browser. In addition, VERA integrates with context menus and the help system.

3.1 The Model Browser View

After installing VERA (see appendix A), the user can open the view "Vera model browser". As illustrated in Figure 3.1, the model browser shows all properties of objects within the model, one at a time. A label at the top shows the name of the object currently being shown. The browser's body consists of two panels: In the left panel, the names of the object's properties are listed. When one of the property names is clicked, the value of that property is shown in the right hand side panel. If that value is a collection, its elements are listed.

The browser uses VERA's meta-model both to find out what properties a given model entity has and to fetch the value(s) of that property. Property values that are model entities themselves can be navigated to by clicking on them. This reveals how much data is available in the model and allows one to quickly browse that data. The model browser also integrates with Eclipse's Java editor. When the current selection in the source code corresponds to a model entity, the browser switches to that entity. Likewise, when the user has browsed to a certain model entity, she can show the source code of that entity in the Java editor by holding down the `Ctrl` key and clicking on that entity. This matches the key bindings in our

visualizations.

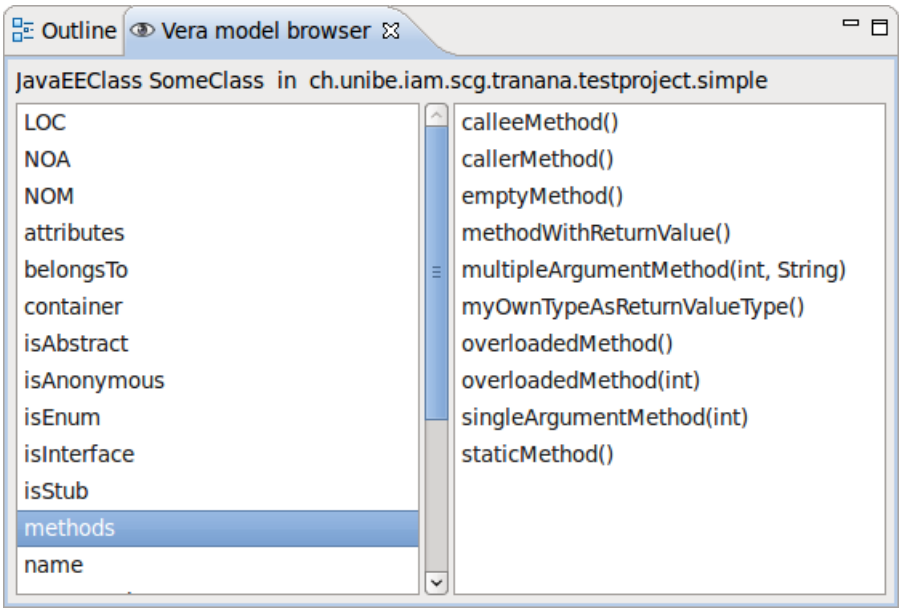

Figure 3.1: The model browser showing an entity named “SomeClass”.

3.2 The Visualizations View

Whereas the model browser presents raw data from the model repository, the view “Vera visualizations” (see Figure 3.2) is used to render problem-centric pictures. This view allows the user to visualize a project in the workspace. VERA is currently designed to work with Java projects only. However, support for other types of projects could easily be added. In the view’s toolbar all available visualizations are presented by icons (No. 1 in Figure 3.2). VERA provides three default visualizations which are explained in the next sections. When the user clicks on a visualization icon, VERA will import the selected project’s source code and render the respective visualization. To stop VERA from re-importing every time the user requests a visualization, the import can be frozen (No. 2 in Figure 3.2), which causes Vera to reuse cached imports. The generated visualization will be displayed in the view’s canvas (No. 3 in Figure 3.2).

3.3 The Packages Visualization 

The first visualization provides a simple overview of the classes defined in a project, grouped by package. Note that this includes classes that have been added to the classpath by linking them to the project (as explained in Section 2.4.1 on page 10), if there are any.

Figure 3.3 shows an example of such a visualization. It consists of nested rectangles. The outermost rectangles (black border, white area) represent packages. They contain black rectangles representing

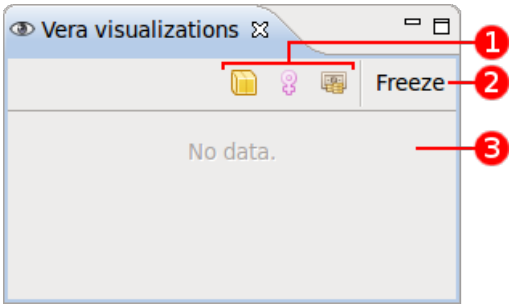

Figure 3.2: VERA's visualizations view

classes inside the package. Finally, a class contains a small gray rectangle for each method it defines. The visualization only displays classes and methods that are defined inside the project (including linked sources).

The visualization supports a rudimentary top-down software comprehension strategy. The user can define her own guidelines concerning the number of classes per package or the number of methods per class. The packages visualization can then help one to find disproportionate packages and classes. The visualization gives an immediate impression of how big the packages of the application under analysis tend to be.

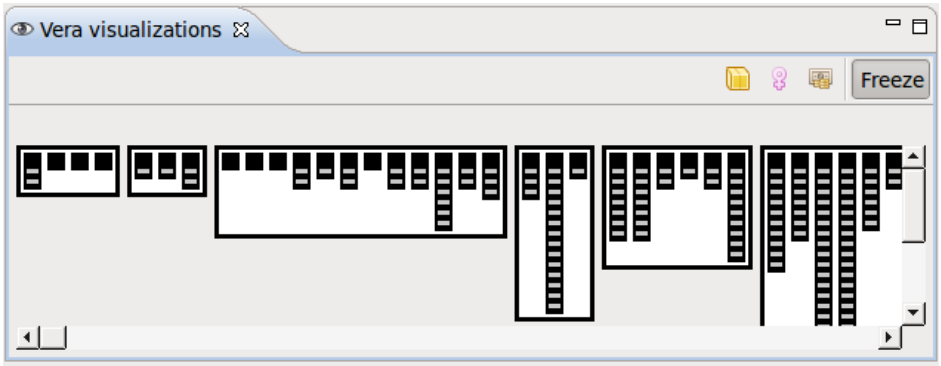

Figure 3.3: The Packages visualization

3.4 The System Complexity Visualization

Figure 3.4 shows VERA's System Complexity visualization, which is a classic polymetric software visualization [10, 11]. It features both class inheritance and metrics on the classes. Every class is represented by a rectangle. The classes are organized in trees that reflect the inheritance hierarchy of classes; specialization increases downwards. The shape of a rectangle is determined by three metrics on the underlying class as follows:

width: number of attributes (NOA)

height: number of methods (NOM)

color: number of lines of code (LOC), where the highest LOC is assigned the color black

Our implementation shows all classes that are defined within the project's Java source files. Classes from archives or external sources (such as the JDK) are omitted, even if they are a superclass of a shown class. For example, if the project defines multiple direct subclasses of `javax.swing.AbstractAction`, these will not be shown in a common inheritance tree. These subclasses will appear as inheritance roots instead. This is mainly because some commonly used libraries contain huge classes which would then appear very dark and all project classes would appear very bright, defeating the purpose of the LOC metric.

The System Complexity visualization allows the user to spot various system properties, patterns and anti-patterns. For example, the developer might be interested in the general use of inheritance; dark, big classes are an indication for bad separation of responsibilities; wide classes contain much data, but provide only few ways of accessing it.

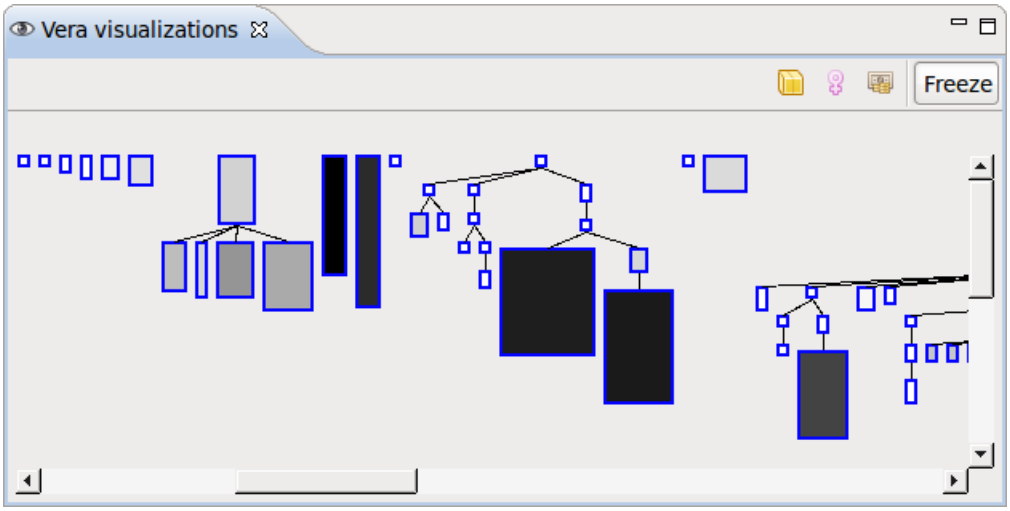

Figure 3.4: Excerpt from a System Complexity visualization by VERA

3.5 The Transaction Flow Visualization

The third visualization that Vera provides out of the box demonstrates its capabilities to combine information from multiple sources. The Transaction Flow visualization [19] addresses two specific questions of JEA developers: (1) Which methods are invoked directly or indirectly inside a transaction and how are they dispersed? (2) Which methods might start unnecessary transactions? That second question is especially interesting when related web pages or web services are experiencing performance issues.

The visualization gives an overview of the methods involved in a transaction (see Figure 3.5). It lays out the methods in an invocation tree. The methods that start transactions are at the top, and below are the methods that are invoked. For a better overview, the methods are grouped by class. This results in the EJB classes being displayed at the top of the visualization. The classes are connected with lines which indicate that a method in the top class invokes a method in the class below. In addition, the bean methods are colored according to the transaction attribute which is configured for them. Blue methods start a transaction (`Requires` or `RequiresNew`) and yellow methods may run inside one (`Supports`).

The question concerning unnecessary transactions is addressed by drawing candidate methods in a magenta color. To be a candidate, a method must have a transaction attribute of `RequiresNew` and must only be called by methods which are already part of a transaction. When a method is colored magenta, it really is just a candidate. There are cases where the `RequiresNew` is appropriate. Other applications might call the method, or the method might really do something that should not be done in an existing transaction but rather always in a new, dedicated transaction.

Of course the developer also wants to know exactly which method invokes which other method, both inside the same class and across classes. To this end, the developer can highlight the invocation tree of a single method simply by hovering over it with the mouse, as illustrated in Figure 3.5. The figure only shows a relatively small section of the visualization. In addition to scrolling, the view "Vera visualizations" can be maximized for a better overview.

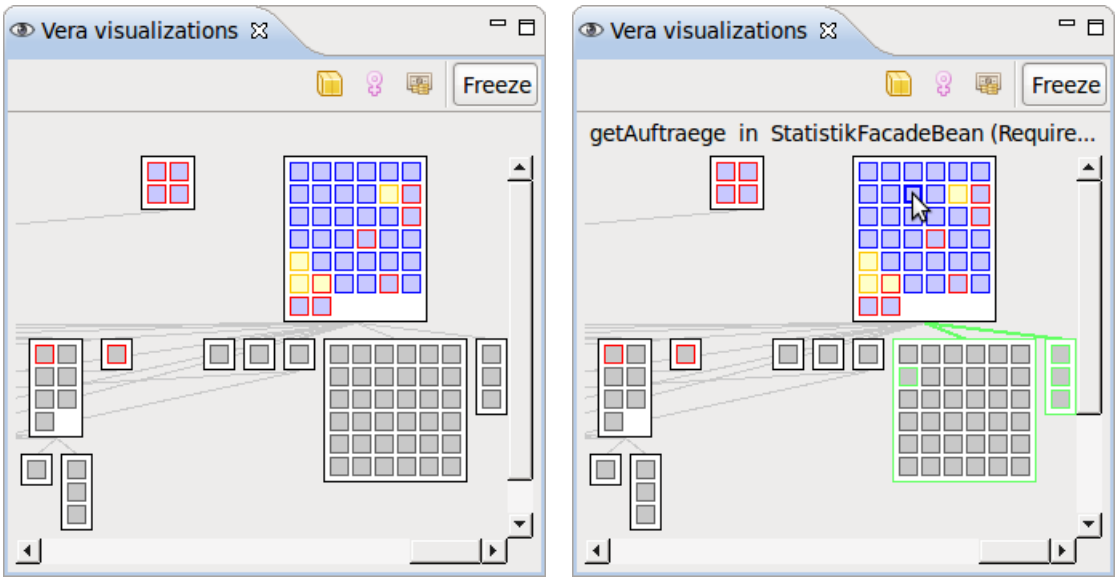

Figure 3.5: The Transaction Flow visualization: Hovering over a method with the mouse highlights its call hierarchy. The highlighted methods, classes and connections are colored differently.

3.6 Further visualization features

The visualizations explained above all share the following user interaction features:

- When an entity is hovered over with the mouse, its name is displayed in a status label (see Figure 3.5).
- When an entity is clicked, it is shown in the Model Browser (see Section 3.1).
- The user can navigate to the source code by clicking on an entity while holding the `Ctrl` key.
- If the visualizer provides a legend and/or a caption, they can be displayed by clicking on the visualization while holding the `Shift` key.
- Context sensitive help is available through the configured hot key.
  The default hot key is `F1` on Windows, `Shift+F1` on Linux and `Help` on Mac.

3.7 Summary

In this chapter, we have seen VERA from the perspective of a user. We explained that VERA introduces two new views to Eclipse's user interface; one to display visualizations and another for a generic model browser. In the next chapter, we will see how VERA works internally and how it can be extended.

4 Architecture

In this chapter we describe our solution, VERA, in more detail. We provide two views on the internals of our tool by explaining the logical components as well as the decomposition into multiple Eclipse plug-ins.

VERA can be logically decomposed into three main parts: the *model tower*, the *importers* and the *analysis tools*, as illustrated in Figure 4.1. The *model tower* provides an abstract representation of the software at hand. The *importers* extract information from the source code or other sources and populate the application model accordingly. The *analysis tools* currently consist of two software visualizations, called System Complexity and Transaction Flow, as well as a model browser that aids software comprehension and shows analytical information about the software entities.

VERA can be extended by writing a new Eclipse plug-in. As we explain VERA's logical components in the following sections, alongside we also introduce the extension-points that VERA offers to extending parties. More details about VERA's nature of being an Eclipse plug-in follow later in this chapter. For now, just note that in order to use VERA's extension-points, an Eclipse plug-in must specify a dependency to VERA's main plug-in, which has the identifier `ch.unibe.scg.vera`.

4.1 The Model Tower

VERA uses Kuhn's Java implementation of the Fame tower [9]. It provides a repository for model objects, meta-model entities and meta-meta-model entities (FM3); and it also connects these three levels of abstraction. Every model object is assigned a unique identifier, which can be used to retrieve that exact

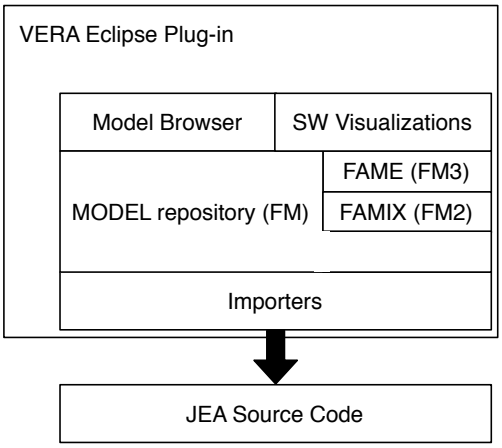

Figure 4.1: Components of VERA: Based on the various JEA sources, the importers create a model which is then presented to the user through software visualizations and a generic browser.

object from the repository at a later time (*i.e.*, when performing analyses on the model or collecting data for a visualization).

4.1.1 An improved Model Repository

Fame's model repository provides most of the functionality we require. It is a container for model objects and also links them to their meta-descriptions. In order to provide more convenience methods, we wrapped a Fame model repository in our own implementation of a model repository.

One type of convenience method retrieves all objects that are instances of a certain entity. That comes in handy when writing an analysis or collecting data for a visualization. We also introduced the notion of *collection filters*. A filter has a method `CollectionFilter<E>#matches(E)`, where `E` can be an arbitrary type. Additional methods like `select(Collection<? extends E>)` allow the programmer to conveniently apply a filter to a collection of model objects, or to combine multiple filters. The model repository's methods to retrieve model objects have variants that accept a filter as an additional argument.

4.1.2 The FAMIX Meta-Model in Java

Our aim was to make our Java implementation of the FAMIX meta-model as easy to extend as the original Smalltalk implementation. As it turns out, this is all but trivial. The Java language simply does not provide any way of patching existing class definitions. We considered various approaches to implementing our meta-model in an extensible way. We detail these approaches in the discussion in Chapter 7.

The solution we finally chose for our Java meta-model works as follows: The entities of FM2 as shown in Figure 2.2 are represented by a hierarchy of Java classes. These classes define getter and setter methods for all the properties of the model objects, as well as some helper methods which allow to query the model more conveniently. The model objects are instances of these classes. That class-instance relationship

provides a natural mapping of model objects to meta-model entities.

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. Another purpose is to facilitate understanding FAMIX by separating the gist of FAMIX from implementation details. Also, completely independent implementations of FAMIX can be developed based on these interfaces. On the downside, VERA's program code is now implemented against the classes, not the interfaces. This could be circumvented by introducing yet another set of interfaces that include all setter and helper methods. But that would mainly add complexity instead of improving clarity, thus maintainability would not be improved. Figure 4.2 illustrates the solution described here.

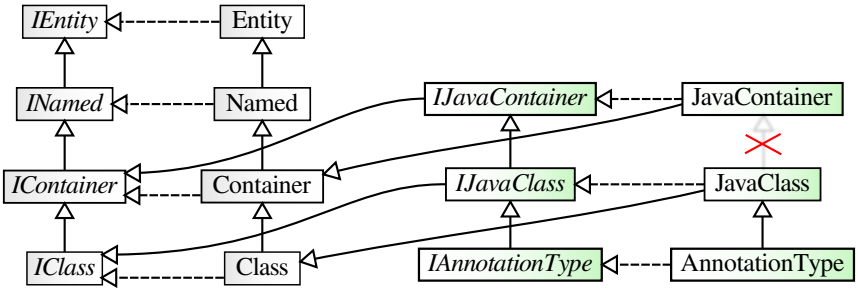

Figure 4.2: VERA's meta-model design. Boxes represent classes; interfaces are depicted by *italic* names. The boxes on the right demonstrate how an extension can specialize or add entities.

Following this solution, we implemented a Java port of FAMIX CORE which includes nearly all of the entities and their properties. We also wrote a Java extension, again as classes and interfaces. This extension includes entities to represent object-oriented code as well as other aspects peculiar to JEAs [19].

Whoever writes an Eclipse plug-in that extends VERA can add more entities to the meta-model. Extending VERA's meta-model involves three steps: First, the entity has to be defined, *i.e.*, a new entity interface and a corresponding class must be implemented. Second, the interface needs to be annotated with Fame annotations as explained in Section 2.5.1.3. Lastly, the new entity must be registered. VERA offers an Eclipse extension-point for that purpose. The full identifier of that extension-point is `ch.unibe.scg.vera.metamodel`. In order to extend VERA at that point, the extending plug-in needs to add an entry to its `plugin.xml` file, as shown in Listing 3.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.metamodel">
        <with class="com.example.ISomeNewMetamodelEntity" />
        <with class="com.example.IAnotherEntity" />
        ...
    </extension>
    ...
</plugin>
```

Listing 3: Declaration of a meta-model extension.

Such an extension instructs VERA to process the Fame annotations in the listed entity interfaces and enrich the meta-model accordingly, *i.e.*, add a meta-description (an instance of an FM3.Class) for that entity to the meta-repository. This is facilitated by Fame itself.

4.2 Importers

When the user chooses to analyze an Eclipse project, the importers are invoked, create model objects (the FM, an instance of the FM2), and store them in the model repository. Importers gather the information they need to populate a FAMIX model by looking at a program's source files. Importers can either operate on the Java source code or on any other file. That, together with a model to hold all extracted information, is the key to analyze software with heterogeneous sources, such as JEAs.

We designed VERA in such a way that the scope for an analysis is always one Eclipse project. The importers will focus on the software that is defined *directly* in this project. Every project knows its classpath, which can consist of directories inside the project, system libraries, other Eclipse projects and any other linked source. For example in a Java project all the classes defined *inside* that project's source folders are interesting; system libraries are only imported partially according to their use from within the project's source code.

Our strategy with importers is to invoke them on demand. That is, whenever the user requests a visualization. Extending parties can alternatively invoke imports programmatically with a call to the method `Vera#getModel(IJavaProject, IProgressMonitor)`. The class `Vera` is a singleton; the instance can be obtained with a call to the static method `Vera.getDefault()`. The `IJavaProject` parameter accepts an object representing an Eclipse project, which can be obtained through Eclipse's JDT API, *i.e.*, use `JavaCore.create()`. An `IProgressMonitor` to pass in as the second argument can be obtained by running the code inside an Eclipse *job*.

Jobs are a notion of the Eclipse runtime to execute possibly long-running tasks in a background thread, which prevents Eclipse's user interface from being blocked. Instructions on how to use the Eclipse's concurrency infrastructure can be found in the on-line documentation[1]. Good examples can also be found in VERA's source code, as VERA consistently uses Eclipse jobs for importers (and for rendering visualizations, as we will see shortly).

Since the importers can take some time to run, the user is given the option to *freeze* re-importing. While VERA is frozen, existing models of Eclipse projects are reused. Freezing can also be thought of as *memoizing* or *caching* imported models. This comes in handy when the user wants to create different visualizations of the same project. Just rendering a visualization is much faster than importing the whole project. When the sources of a project change, VERA should be unfrozen and the project re-imported to reflect the changes.

An incremental re-import only of the changed sources is not supported at the time being. There are two main reasons why we decided against that. First, even an incremental import takes some time. For

[1] `http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/runtime_jobs.htm`

developers who are used to frequently hit the save button, even waiting one or two seconds more can get tedious. Second, while Eclipse does provide a decent mechanism to determine which Java source files changed since the last build, such a mechanism is missing for non-Java files. Thus the good solution would be to implement a change tracking mechanism and not invoke an incremental import on every Java build but only at times when a consistent model is required, *e.g.*, when the user requests a visualization.

4.2.1 Extensibility

Since VERA is built to be extended, custom importers that extract additional information can be added. To this end, VERA provides the two extension points `ch.unibe.scg.vera.importers` and `ch.unibe.scg.vera.AST_importers` (subsequently referred to as `importers` and `AST_importers`, respectively). Whereas the `importers` point is suited to any kind of importer, the `AST_importers` point is meant for importers that operate exclusively on Java source files. The two extension points are explained in more detail in the following subsections.

This does not make the individual importers extensible, though; an existing importer's behaviour cannot be changed by extending parties. But importers can work together by specifying dependencies on each other, which influences the order in which the importers are executed. The dependency mechanism allows extending parties (which add their own importers) to rely on a certain state of completeness of the model.

Note that we do not allow replacing existing model objects in the model repository because doing so could introduce compatibility issues with other VERA extensions that depend on the original entity. Also, values that have been assigned to a property by an existing importer should not be touched by an extension. Doing so can break existing analyses (implemented by VERA's core or unknown third parties).

For the two visualizations we implemented so far, we required a representation of the Java source code as well as some information from an XML file. We implemented multiple importers that work together to provide that information.

4.2.2 The Java AST Importers

Since VERA operates within Eclipse, we do not have to parse the Java source files ourselves. VERA relies on Eclipse's Java parser to generate an Abstract Syntax Tree (AST) of the Java source code. An AST is a very low-level model of the source code contained in one single Java source file. It essentially contains everything that is written in that file, not as a sequence of characters but rather as a nested structure of Java objects which represent the logical components of the source code. For example, every AST will contain a type definition, therein some attribute and method definitions, and so on.

That AST is then consumed by VERA's Java importer following a visitor pattern. The importer creates FAMIX entities and stores them in the project's model repository. Existing entities can be modified, whereas replacing entities in the model repository is forbidden. It might look tempting at first to replace an existing entity with a more specialized one that has more properties. But changing existing behaviour (*i.e.*, by overriding methods) would most certainly break existing clients. Furthermore, replacing an entity

in the model repository does not replace existing references to it from other model objects, thus causing inconsistency in the model.

VERA's main importer creates a model that contains a nearly complete subset of all pure Java entities. This includes packages, types (interfaces, classes, enums, member classes, anonymous classes; including generics), methods (including generics) and inheritance relationships.

These elements are imported in one pass over each Java file's AST. While traversing one AST, the importer encounters objects that are not defined inside the same file and have not yet been imported. Just using the current AST, resolving these objects requires extra work. VERA's importers can conveniently delegate that work to Eclipse's parser. Given an AST object, Eclipse can usually resolve it and provide a so-called *binding*. That binding can further be traversed. For example, the binding for a class definition is an ITypeBinding which can be queried for its name, visibility modifiers, implemented interfaces and superclass. When we retrieve the superclass of an ITypeBinding, we get another binding. Likewise, all non-primitive members of bindings are bindings themselves. We create new model objects based on these bindings. This allows us to combine the visitor pattern with a factory that creates model objects recursively. For example when a class definition is encountered in the AST, the importer passes its binding to a factory method. The factory creates a new `JavaEEClass` object. This includes, *e.g.*, setting the containing package. If the the model repository does not yet contain an object for that package, then the factory just creates one on the fly. That requires that every object has a unique identifier, which is conveniently provided by the bindings. The factory does not initialize all properties of an object, though. For example the creation of methods is initiated from the importer's `visit(MethodDeclaration)` method.

While this main Java importer does most of the work, some entities are still missing or incomplete (*i.e.*, not yet implemented). That includes attribute access, local variables, method overriding, class and instance initializer blocks, Javadoc comments and visibility of types and methods. Importing these is left to other importers.

In some cases, bindings cannot be resolved. When the project has no compiler errors, these cases are very rare. The importer can handle such unresolvable bindings by declaring the model object as stub of a certain kind. As an example, let method B be called from within method A. But the class that defines B is not yet on the classpath, so B cannot be resolved. The importer should then mark A as a stub. An object can be marked as a stub for various reasons, thus every mark is accompanied with a short message text. In our example this means that A is not just flagged as a stub but as "stub because not all invoked methods could be resolved". This way, whoever performs an analysis on the model can check whether all required properties could be imported. For example, methods like A are rendered with a red border in our Transaction Flow visualization.

Extending parties can add their own importer to leverage the ASTs through VERA's `AST_importers` extension-point. Listing 4 shows what needs to be added to the extending plug-in's `plugin.xml` file in order to register a new AST importer.

Next to the importer's implementing class (which has to be a subclass of `JavaASTImportVisitor`), a unique identifier (`id`) has to be given. The Listing also shows how to add a dependency to another AST importer. The value of the `after` attribute must be the `id` of an existing AST importer. Note that only

dependencies to other AST importers are possible (*i.e.*, not on omnivore importers, see next section). At the moment, `after` is the only kind of dependency VERA supports.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.AST_importers">
        <AST-importer
            class="ch.unibe.scg.vera.importer.JavaASTMethodInvocationImporter"
            id="Vera.Java-method-invocation-importer">
            <dependency after="Vera.Java-basic-importer" />
        </AST-importer>
    </extension>
    ...
</plugin>
```

Listing 4: Registering the new importer through the `AST_importers` extension point.

The AST importer in Listing 4 has been implemented as a proof of concept. It analyses which Java method invokes which other Java methods, creates instances of the meta-model entity `MethodInvocation`, and adds the new model objects to the model repository. This importer requires that all methods are already present in the model repository, so we specify that it has to run `after` VERA's main Java importer.

4.2.3 Omnivore (non-Java) Importers

So far, we talked about importers that consume Java ASTs, but this is just a specialized form of importer. There is another extension point that allows one to hook in a more general type of importers, which we call *omnivore* importers. Instead of being fed an AST, they are supposed to find and treat the files they need by themselves. Although we call these importers "omnivore", they are not required to consume *all* files inside a project. They *can* consider all kinds of file, but most of them will probably specialize on one or two kinds. Since there is only one other type of importer, the omnivore importers can also be thought of as the "non-Java" importers.

```
<extension point="ch.unibe.scg.vera.importers">
    <importer
        class="com.example.HtmlImporter"
        id="html_importer">
        <dependency after="id_of_some_other_omnivore_importer" />
    </importer>
</extension>
```

Listing 5: Registering an omnivore (non-Java) source importer through the `importers` extension point.

The `importers` extension-point, through which omnivore importers for any kind of sources can be registered, looks very similar to the `AST_importers` extension-point, as can be seen in Listing 5. The main difference is that an omnivore importer does not need to extend a given class but instead must implement the interface `ch.unibe.scg.vera.extensions.VeraImporter`. Also, omnivore importers can only depend on other omnivore importers, not on AST importers.

VERA ships with one omnivore importer that operates on an XML file. That file is the EJB deployment descriptor, the `ejb-jar.xml` we already encountered in Section 2.3.2. It is located in the application's META-INF folder, for example at `.../workspace/MyProject/WebContent/META-INF/ejb-jar.xml`. The importer uses an XML parsing library which allows one to traverse the XML structure of the document as a tree. The XML document itself does not become part of VERA's model, nor does its content. Instead, the importer cherry picks parts of the content data, based on which it both creates new model objects (JavaBeans) and modifies existing ones (adds transaction attributes to the Bean class' methods).

As a special case, we implemented an omnivore importer that finds all Java source files and generates ASTs for all of them. The AST of each file is passed on to all registered AST importers. The id of that consolidating importer is `"Vera.ASTImporter"`.

4.3 Visualizers

One way to present the model to the user is through software visualizations. VERA embraces this with its visualizations view (see Section 3.2). That view relies on a number of so-called *visualizers* to render visualizations. When the user requests a visualization, the imported model is passed to one of the visualizers, which then uses that data to render a visualization. The visualizations view makes available all visualizers that are registered through VERA's `ch.unibe.scg.vera.visualizers` extension-point (subsequently called `visualizers` for short). Listing 10 shows what an extending plug-in has to add to its `plugin.xml` file to register a new visualizer.

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

Listing 6: Declaration of a visualizer extension

Every visualizer is registered with a title and an optional icon to represent that visualizer in Eclipse's UI. The implementing class has to implement the interface `ch.unibe.scg.vera.extensions.VeraVisualizer` which consists of two methods. The first method `getSWTControl(Composite) : Control` is invoked during VERA's initialization, *i.e.*, when Eclipse starts. The visualizer is supposed to create its own SWT Control (as a child of the passed SWT Composite) to display the visualization on. That control will probably be (or contain) an SWT Canvas, which is suited for rendering. The visualizations view manages those controls and, when the user requests a visualization, displays the control of the respective visualizer. Right before the control is displayed, the second method of the `VeraVisualizer` interface, `visualize`

`(IProjectModelRepository, IProgressMonitor)`, is invoked on the corresponding visualizer. In this method, the visualizer is supposed to render its visualization on its SWT Control (and report progress through the passed monitor).

By allowing the visualizer to create its own SWT Control to render on, VERA allows the use of any drawing framework that can be displayed on an SWT Control (this includes Swing graphics). An AWT Frame can be embedded in SWT using an `org.eclipse.swt.awt.SWT_AWT` bridge.

We render the software visualizations presented in Chapter 3 using the Draw2d toolkit. Draw2d is also part of other frameworks like Zest[2] or GEF[3] that can help the users to create more complex visualizations by adding more layouts and by handling mouse and keyboard interaction. In an earlier development stage, we also evaluated jMondrian [8], the Java implementation of Mondrian [15], a visualization scripting framework originally written in Smalltalk. We found that jMondrian is less expressive and less powerful than Draw2d, and it is not maintained. However, the model of jMondrian is simpler than the one of Draw2d, so jMondrian is a good alternative when creating very simple visualizations.

For these two drawing frameworks there are abstract classes available that implement the interface `VeraVisualizer` and simplify creation of visualizers for the respective drawing framework. So if you intend to render a visualization using Draw2d or jMondrian, consider subclassing `Draw2dVisualizer` or `JMondrianVisualizer`, respectively. The latter uses `SWT_AWT` bridging as mentioned above.

4.4 Plug-in Components

VERA is not just a single Eclipse plug-in, but really a set of different plug-ins that work together. We decided to factor out components that have been developed by third parties, that are optional and/or that can stand alone. In this section, we introduce each of the components and their dependencies. Table 4.1 on page 32 lists all dependencies between plug-ins, including the ones that are provided by the Eclipse community (*e.g.*, Eclipse runtime, JDT, JUnit, UI integration, Draw2d).

`ch.akuhn.fame` The Java implementation of Fame by Kuhn. The FM3 annotations which we use to add the meta-level information to our entities are defined inside this plug-in, alongside with a parser that evaluates the annotations and instantiates both the meta-model and meta-meta-model. Fame also provides a rudimentary meta-model generator and has the capability of exporting a model to an MSE file. The source code of this plug-in can be used outside Eclipse without modification.

`ch.unibe.scg.famix` This plug-in contains our own Java implementation of the FAMIX Core 3.0 meta-model. This includes the entity definitions both as interfaces and as classes. Note that our Java extension to the FAMIX meta-model is not contained in this plug-in. Instead, the Java specific entities are defined directly in VERA's core plug-in. The entities are annotated with FM3 annotations, which introduces a plug-in dependency to the Fame plug-in. This plug-in's source code is completely independent from Eclipse, too.

[2] `http://www.eclipse.org/gef/zest`
[3] `http://www.eclipse.org/gef`

List of Plug-in Dependencies and Extensions		
Plug-in	Dependens on plug-in	Includes extensions to point
ch.akuhn.fame	-	-
ch.akuhn.fame.tests	org.junit4	-
ch.unibe.scg.famix	ch.akuhn.fame	-
ch.unibe.scg.vera	ch.akuhn.fame ch.unibe.scg.famix org.eclipse.core.runtime org.eclipse.core.resources org.eclipse.core.expressions org.eclipse.draw2d org.eclipse.equinox.common org.eclipse.jdt.core org.eclipse.jdt.ui org.eclipse.jface.text org.eclipse.ui org.eclipse.ui.commands org.eclipse.ui.menus org.eclipse.ui.navigator org.eclipse.ui.navigator.resources org.eclipse.ui.views org.eclipse.help	ch.unibe.scg.vera.AST_importers ch.unibe.scg.vera.importers ch.unibe.scg.vera.metamodel ch.unibe.scg.vera.visualizers org.eclipse.ui.views org.eclipse.ui.commands org.eclipse.ui.menus
ch.unibe.scg.vera.core.tests	org.junit4	-
ch.unibe.scg.vera.help	ch.unibe.scg.vera	org.eclipse.help.toc org.eclipse.help.contexts
lrg.jMondrian	-	-
ch.unibe.scg.vera_jMondrian	ch.unibe.scg.vera lrg.jMondrian org.eclipse.core.runtime org.eclipse.jdt.core org.eclipse.jdt.ui org.eclipse.ui	ch.unibe.scg.vera.visualizers

Table 4.1: Vera's plug-ins, their dependencies to other plug-ins and which extension-points they use. The gray plug-ins and extension-points are external to VERA.

`ch.unibe.scg.vera` This is VERA's main plug-in. Extending plug-ins should specify a dependency to this plug-in; it provides the infrastructure for adding meta-model entities, importers and visualizers, including the visualizations view. It also includes the the model browser view, the Java extensions to the

FAMIX Core, the Java importers, and the three Draw2d visualizers. It depends on the FAMIX Core and Fame plug-ins, as well as on numerous plug-ins outside VERA. Eclipse's UI plug-ins allow VERA to register its two new views, use the typical command buttons in the view's toolbar and add entries to pop-up menus. Eclipse's core plug-ins allow VERA to leverage Eclipse's extension mechanism, *i.e.*, VERA can define its own extension-points and find all installed extensions to those points at runtime. The Draw2d plug-in is obviously used by the three included visualizers. Lastly, Eclipse's help plug-in allows VERA to hook the correct help texts to the currently displayed visualization (and other parts of VERA).

`ch.unibe.scg.vera.help` We also factored out the help contents into a separate plug-in. The help contents are quite independent from the rest, and they are just data, no code. (Note that we are talking about help texts for a VERA *user*, not about code comments. The Java source code itself is quite thoroughly documented with Javadoc.) Being in a separate plug-in, the help texts can be improved and then published without re-publishing the code. Always increasing the version number on the code does not make any sense when the code did not change at all. The help contents we provide are not very extensive, yet, so there might be quite a few updates in the future. This plug-in depends on VERA's main plug-in.

Extending parties can include help texts for their visualizations directly in their plug-in; there is no need to specify a dependency to the `ch.unibe.scg.vera.help` plug-in. VERA's help plug-in gives an example of how to add help content through Eclipse's help system. For further instructions, please refer to Eclipse's on-line documentation[4].

`ch.unibe.scg.vera.core.tests` and **`ch.akuhn.fame.tests`** The unit tests for Vera are not included in Vera's main plug-in. Instead, we factored them out to a *plug-in fragment*. For the tests, we use a testing workspace, but we do not include it in the test plug-in. Shipping the tests without the necessary test data would not make any sense, so we keep them separate in this plug-in fragment, which we do not ship with VERA. The tests as well as the workspace contents can still be downloaded directly from the SVN repository (see Appendix A).

As we did for VERA, we also factored out the unit tests for Fame to a plug-in fragment.

`lrg.jMondrian` The Java implementation of Mondrian by the LOOSE Research Group[5].

`ch.unibe.scg.vera_jMondrian` When evaluating drawing frameworks, jMondrian was a candidate. But since it is not stable, yet, and because Draw2d is more suited to our needs, we decided against jMondrian. This plug-in provides the abandoned prototypes of the System Complexity and Packages visualizations, written with jMondrian. It depends on VERA's main plug-in and the jMondrian plug-in. Extending parties can refer to this plug-in to see how visualizations generated by a visualzation framework other than Draw2d can be embedded with VERA.

[4] `http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/ua_help.htm`

[5] `http://loose.upt.ro/reengineering/research/jMondrian`

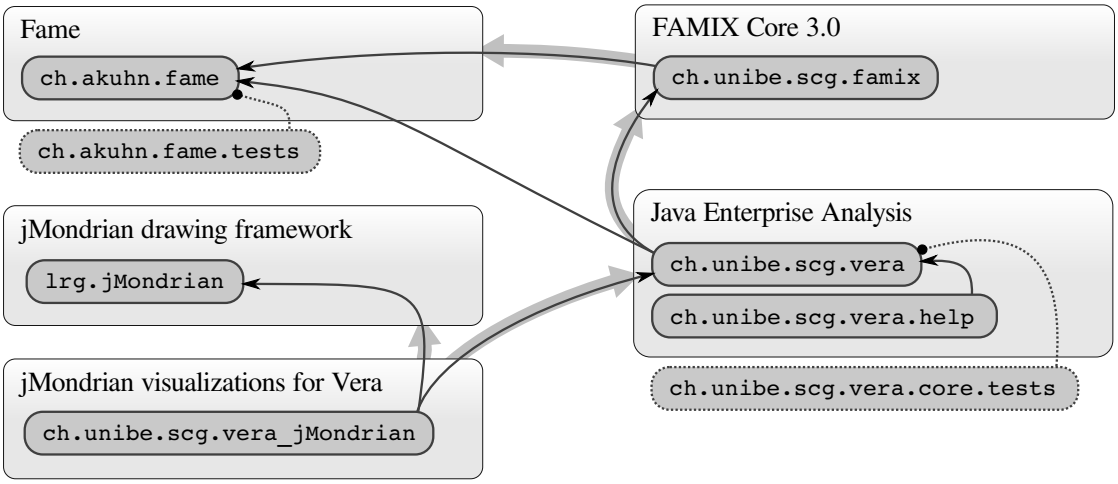

Figure 4.3: Overview of Vera's Eclipse plug-ins and installable Eclipse features. The inner boxes and thin arrows stand for the plug-ins and their dependencies. The outer boxes and grey arrows stand for the features and their dependencies. The dotted boxes stand for plug-in fragments containing the tests.

4.5 Installable Features

We created an update-site for VERA (see Appendix A) on which we publish different combinations of our plug-ins as separate Eclipse features. (The terms plug-in, feature and update-site are explained in Section 2.4.2.) Figure 4.3 gives a graphical overview of VERA Eclipse features and their dependencies, while the details are explained in the text below.

Java Enterprise Visualization. This is the main Eclipse feature, as it wraps VERA's core plug-in. When this feature is installed, the two features *FAMIX Core* and *Fame* will automatically be installed, too, as they are prerequisites. Thus, this feature installs all one needs to get started with VERA. This feature is published under a BSD 3-clause license.

For people who want to extend VERA or reuse parts of it, there are four more features that include resources for plug-in development. Most of the features on that update-site wrap exactly one of the plug-ins mentioned in the previous section. But some features depend on others (according to the dependencies among the plug-ins they wrap), which ensures that all required plug-ins are installed. For example when the jMondrian visualizers are installed, the jMondrian framework is automatically installed along with it.

Fame and **jMondrian.** We mainly publish Fame and jMondrian as separate downloads because we do not know any other source that offers them for download on an Eclipse update-site. What we publish is what we use for VERA, our forks of the original versions. We include bugfixes and adaptations to our needs. Note that more recent versions of jMondrian might exist already, so you might want to contact its original developers. Kuhn's Fame is published under a GPL + LGPL dual license, jMondrian under a BSD 2-clause license.

FAMIX Core. The FAMIX Core feature contains our Java implementation of the meta-model. This is interesting for people who plan on using our Java implementation of the FAMIX core to analyze other languages, *e.g.*, for Eclipse's C/C++ development environment. Our FAMIX implementation is published under a BSD 3-clause license.

jMondrian visualizers. This feature ships the abandoned jMondrian visualizers. They are not intended to be used any more. We publish that work as an example of how a different drawing framework can be used for visualizations. It is also published under a BSD 3-clause license.

4.6 Summary

In this chapter we have seen that VERA provides a base to analyze JEAs within Eclipse. It exposes an expressive meta-model of the source code and some JEA specifics, invokes information importers that populate a model, and provides a small set of visualizations. All of the three, model, importers and visualizers, are built to be extended. We also learned that VERA is composed of multiple Eclipse plug-ins which can be installed following the standard Eclipse way.

In the following chapter we see how extending parties can leverage all of this.

5
Extension Example

This chapter exemplifies how VERA can be extended. In particular, we present a scenario in which we enable VERA to analyze Java annotations.

Our scenario revolves around Cool Coders, Incorporated. This fictional software development company maintains an old Java web application which still uses EJB 2.1. With VERA's Transaction Flow visualization they were able to quickly identify several ways of improving the codebase. They would very much like to have the same kind of analysis for more recent applications that use EJB 3.0. But in that modern version of EJB, the information about transactions lies in Java annotations instead of the `ejb-jar.xml` file.

The Cool Coders decide to extend VERA and enable it to process annotations in Java source code. If it were just for the Transaction Flow visualization, all they would need is a new importer which interprets the EJB specific annotations in the Java source code, creates JavaBean model objects, and sets the transaction attributes on the Bean's methods accordingly. The Cool Coders realize that annotations also appear in other contexts in some of their projects. So they decide to add annotations as first-class objects to the application model. VERA's meta-model does not yet contain any entity that could represent annotation instances. So the Cool Coders first need to extend the meta-model. They also need an importer which reads the Java annotations, creates instances of the new meta-model entity and add these new model objects to the repository. This same importer can also do the work of interpreting the EJB specific annotations.

With the application model so enriched, the Cool Coders can then reason about annotations in the Java source code. They can run any analysis they like and display the results to the Eclipse user in an appropriate way.

In the remainder of this chapter, we show how the Cool Coders extend the meta-model, add a custom importer and add a custom visualizer.

5.1 Extending the meta-model

In a first step the Cool Coders extend the meta-model of VERA. This scenario requires one additional class: `AnnotationInstance`. As the name suggests, that new entity represents the Java annotation instances. A new entity for the Java annotation types is not necessary. Although representing annotation types with their own entity would be a natural thing to do, it turns out that annotation types must be represented by the already existing meta-model class `JavaEEClass`. This is due to the fact that Eclipse's Java parser considers annotation types special cases of Java classes. Thus, VERA's existing Java importer is oblivious to the difference between a class definition and an annotation type definition; It just creates a `JavaEEClass` model object whenever it encounters an annotation type definition. That object is then added to the model repository and cannot be replaced by another object.

The Cool Coders embed our new entity `AnnotationInstance` in the existing FAMIX meta-model as a subclass of `SourcedEntity`, since it can be related to a location in the Java source code but is not similar to any existing subclass of `SourcedEntity`. The new entity implements a new entity interface `IAnnotationInstance` which extends `ISourcedEntity`, following the same pattern as the rest of the meta-model. The interface defines the getter methods for the information contained in the new entities, *i.e.* references to the annotation's type and the annotated entity. These methods are annotated with `@FameProperty`, and the interface itself with `@FameDescription` and `@FamePackage`, as shown in Listing 7.

```
@FamePackage("FAMIX")
@FameDescription("AnnotationInstance")
public interface IAnnotationInstance extends ISourcedEntity {

    @FameProperty(name="annotationType")
    IClass getAnnotationType();

    @FameProperty(name="annotatedEntity")
    INamedEntity getAnnotatedEntity();
}

public class AnnotationInstance extends SourcedEntity implements IAnnotationInstance {
    // implement the interface methods, add some helper methods
}
```

Listing 7: Definition of the meta-model entity for Java annotation instances.

The Cool Coders place the two definitions in the `/src/` folder of their plug-in, in two separate files in the `ch.unibe.scg.vera.metamodel` package. They register their new entity through VERA's `metamodel` extension-point as shown in Listing 8.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.metamodel">
        <with class="com.example.AnnotationInstance" />
    </extension>
    ...
</plugin>
```

Listing 8: Declaration of the meta-model extension.

With that addition to the meta-model set up, the Cool Coders can start enriching the model of an Eclipse project with information about Java annotations.

5.2 Adding a custom importer

The Cool Coders need a new importer for the annotation instances. It will only look at Java source code, so we declare an extension to VERA's extension point `AST_importers`. The new annotation importer should run *after* the Java importer, so it can verify that every annotation instance corresponds to a known annotation type. The Cool Coders register the importer by adding the contents of Listing 9 to the `plugin.xml` file of their plug-in.

```
<plugin>
    ...
    <extension point="ch.unibe.scg.vera.AST_importers">
        <AST-importer
            class="ch.unibe.scg.vera.importer.AnnotationInstanceImporter"
            id="Java-annotation-instance-importer">
            <dependency after="Vera.Java-basic-importer" />
        </AST-importer>
    </extension>
    ...
</plugin>
```

Listing 9: Registering the new importer through the `AST_importers` extension point.

While visiting annotations of Java classes and methods, the importer creates instances of the new entity `AnnotationInstance` and adds them to the model repository. The implementation of the new importer is outlined in Appendix B.

5.3 Exposing the data

With the extensions for both the meta-model and the importers in place, the Cool Coders are ready to expose the information to the developer.

They decide to implement an Annotation Constellation[1] visualization. That visualization shows the names of all used annotations on Java classes. Labels for frequently used annotations are larger, and classes that use the same annotations are placed closer together. The visualization helps in finding annotations which are often used together, and finding out which technologies are used. Furthermore, it makes explicit the fact that annotations introduce extra dependencies, which leads to extra complexity. Figure 5.1 shows what that visualization looks like.

The Cool Coders implement such a visualization in Java and register it as shown in Listing 10.

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

Listing 10: Declaration of a visualizer extension

5.4 Summary

In this chapter, we followed the Cool Coders in their process of extending VERA with their own Eclipse plug-in. We have seen a meta-model extension, accompanied by an AST importer extension, and a visualizer extension.

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

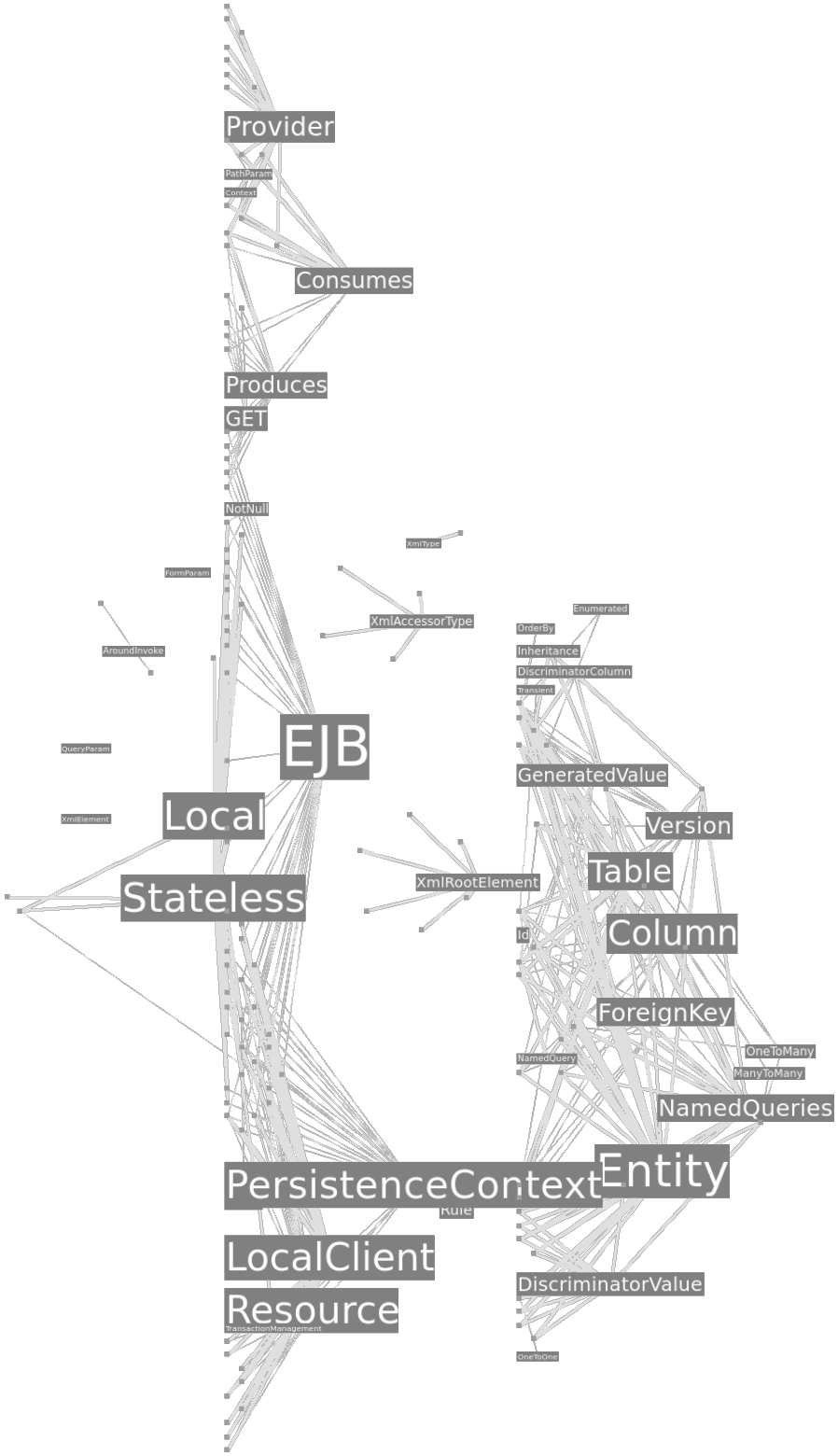

Figure 5.1: Illustrative image for the Annotation Constellation visualization (in this case applied to a JEA). The labels stand for annotation types, the small boxes for Java classes; the lines depict on which classes there are annotations of which type. By courtesy of the Moose Book.

6
Related Work

In this chapter we present closely related existing software analysis tools for Java applications. We chose tools that provide visualizations and meet some of the three goals we have for VERA. This is just a small selection of all the tools that exist nowadays. Canfora and Di Penta [6] provide a more general overview.

6.1 Moose

Moose [16] 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 modelling, to scripting software visualizations with the drawing framework Mondrian[1].

One core element of Moose is *FAMIX* [25], a language independent meta-model that describes the static structure of object-oriented software systems. The FAMIX meta-model is extensible. Different extensions exist that can describe a variety of different software systems, including JEAs. By using tools such as VerveineJ[2] or inFusion (a newer version of iPlasma [14]), it is possible to create a *FAMIX* model from Java source code and export it to a file. The model can then be imported into Moose and analyzed there. Thanks to meta-model extensibility, composable tools and scripting capabilities for visualizations, the user can adapt Moose to fulfil a very wide range of analysis tasks.

However, in order to use the tools provided by Moose, the user should be familiar with the Smalltalk language, the Smalltalk working environment, and the internals of Moose. For Java developers this means

[1] `http://www.moosetechnology.org/tools/vw/mondrian`
[2] `http://www.moosetechnology.org/tools/verveinej`

that they have to get to know a completely different world of software, which might keep them from trying Moose in the first place. A related issue for Java developers is the cognitive context switch between Smalltalk/Moose and Java/IDE during the development process.

VERA does not provide nearly as many different analyses and visualizations out of the box as Moose does. Both VERA and Moose are extensible and, thanks to FAMIX, support cross-technology analysis. Moose's capability of scripting visualizations is something that we consider a missing feature in VERA.

6.2 SHriMP Views

The Simple Hierarchical Multi-Perspective tool (SHriMP) [22, 24], displays architectural diagrams of software using nested graphs. Its user interface embeds source code inside the graph nodes and allows the user to navigate the nodes following low-level dependencies. It supports animated panning, zooming, and fisheye-view actions for viewing high-level structures. Compared to Moose, SHriMP also requires a context switch but is very easy to learn and intuitive to use. SHriMP does not offer its users the possibility to add custom visualizations or other types of analysis; it is not extensible.

There is an Eclipse plug-in called *Creole* that integrates SHriMP with older versions of the Eclipse IDE, as shown in figure Figure 6.2. However this plug-in is no longer maintained and is incompatible with recent versions of Eclipse (> 3.3). (This adds another criterion to the helpfulness of an analysis or software comprehension tool: Whether it is actively maintained.)

6.3 Softwarenaut

Softwarenaut [13] is a static analysis tool that supports architecture recovery through visualization and interactive exploration. Its focus is to support top-down code comprehension. The user can selectively increase or decrease the level of detail from a coarse-grained to a very fine-grained view of the system, while simultaneously being presented with a visualization and some metrics. This kind of code exploration could also be implemented on top of VERA. The three visualizations VERA provides so far support the top-down comprehension strategy mostly by giving a visual overview of the system.

As can be seen in Figure 6.3, Softwarenaut simultaneously presents the user with the browser and complementary information such as a visualization and some metrics. Eclipse allows Vera to add its visualization in a similar manner right next to the main focus of work. In Softwarenaut, the main panel features the visualized software components. In Eclipse, the main panel is the Java editor.

Softwarenaut again is a tool external to the IDE, thus requires a context switch. It interoperates with Moose and can therefore process different source languages.

In the remainder of this chapter, we focus on tools that are closer to the Java developer. They all integrate with the Eclipse IDE, a widely used Java development environment.

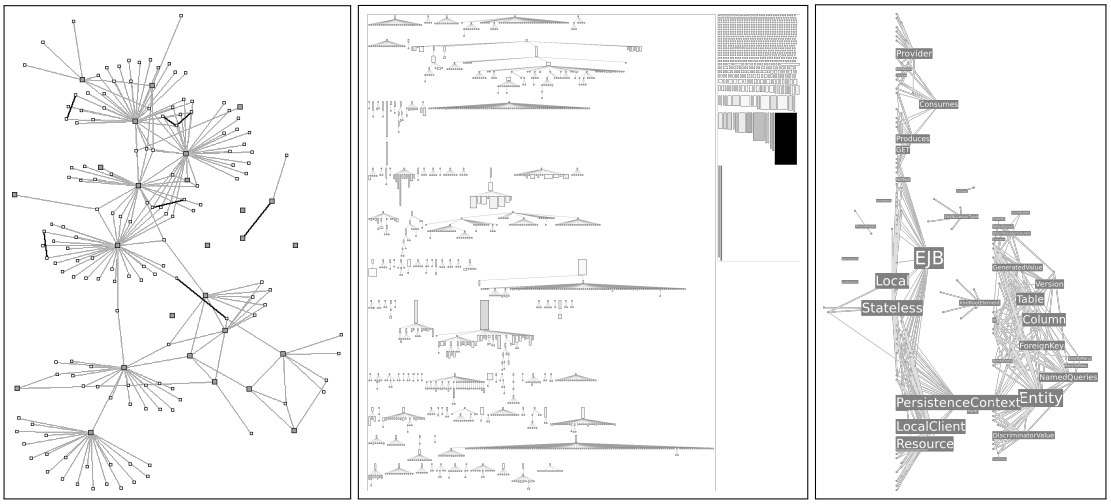

Figure 6.1: Three visualizations by Moose (drawn with Mondrian)

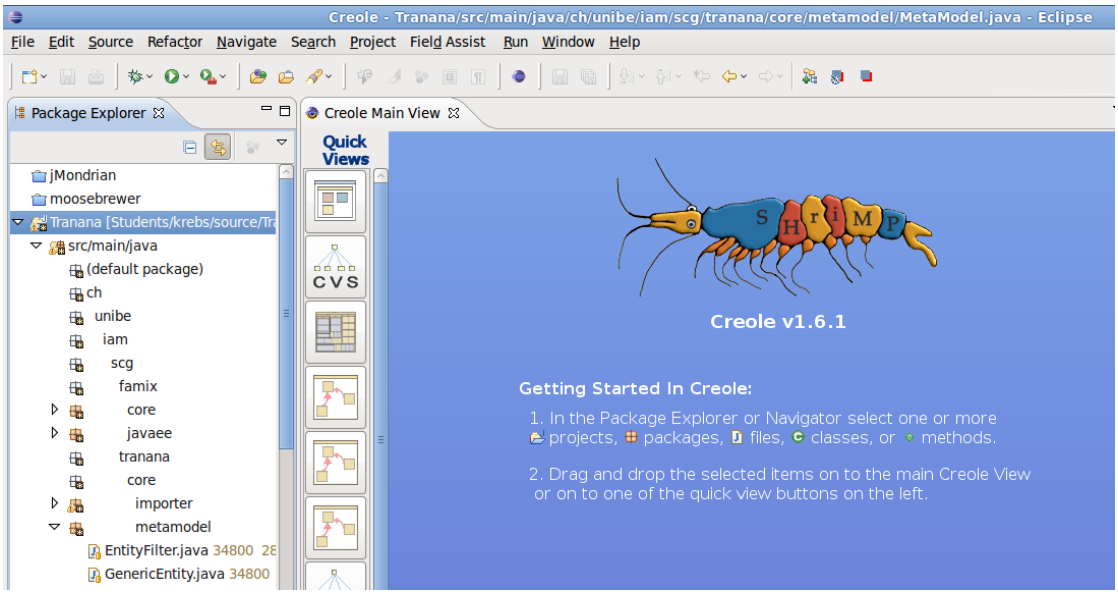

Figure 6.2: Screenshot of Creole, the SHriMP Eclipse plug-in.

6.4 Architexa

Figure 6.4 shows three visualizations created with Architexa[3], a commercial Eclipse plug-in that allows the user to create UML-like diagrams of Java source code in an explorative manner.

By explorative we mean that the user is first presented with a very high-level view of a system and then allowed to increase the level of detail on any component. This is similar to Softwarenaut in that both mostly facilitate the top-down and knowledge-based comprehension strategies. This explorative way of navigation allows the user to create many different views on a system that highlight different aspects of the software design. It is useful to get to know the code and also for creating images for documentary purposes.

VERA also differs from Architexa in the ambition to enable the analysis of JEAs. While Architexa focuses on visualizing pure Java, we consider the ability to combine information from different sources vital. Also, a single company can hardly think of and realize useful visualizations for all possible combinations of technologies and frameworks. We should enable the developers to build their own tools, which are tailored to their exact needs, on top of existing ones. Therefore the next tools we present are all built to be extended.

6.5 X-Ray

X-Ray[4] is an open-source Eclipse plug-in which provides a small set of interactive source code visualizations, such as the one shown in Figure 6.5. It has been developed as a Bachelor's project at the University of Lugano.

In contrast to VERA, X-Ray uses a very minimal model of the source code (just Java packages and classes), which limits its analytical capabilities. That makes X-Ray ill-suited for analyzing JEAs. Also, it does not allow the user to easily add custom visualizations like VERA does. Nonetheless it is a good example that providing a reusable model is a good starting point for different kinds of analysis: There are at least two other Eclipse plug-ins that create their own visualization based on that model: *Citilyzer*[5] and *Proximity Alert*[6].

6.6 inCode

inCode [7] is another Eclipse plug-in that supports the user in software quality assessment. Its main features are on-the-fly detection of design flaws, automated refactoring for correcting the flaws, architectural assessment, and interactive code visualizations (see Figure 6.6). The functionality provided by inCode was originally developed for *inFusion* (formerly *iPlasma* [14]), another stand-alone analysis program for Java, C# and C++. inCode itself focuses purely on Java code.

[3] http://www.architexa.com/index_c.php
[4] http://xray.inf.usi.ch/
[5] http://atelier.inf.unisi.ch/~biaggia/citylyzer/
[6] http://atelier.inf.unisi.ch/~casarela/ProximityAlert/

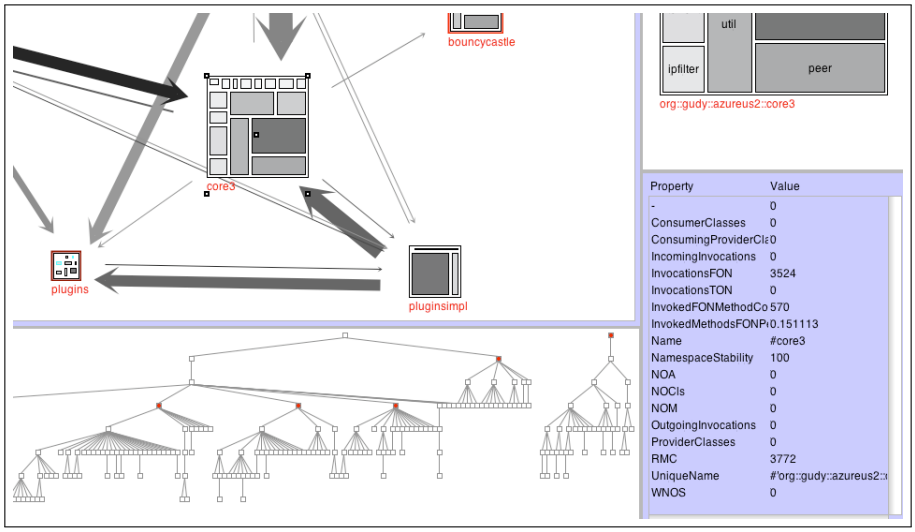

Figure 6.3: Screenshot of Softwarenaut, featuring the different parts of the user interface.

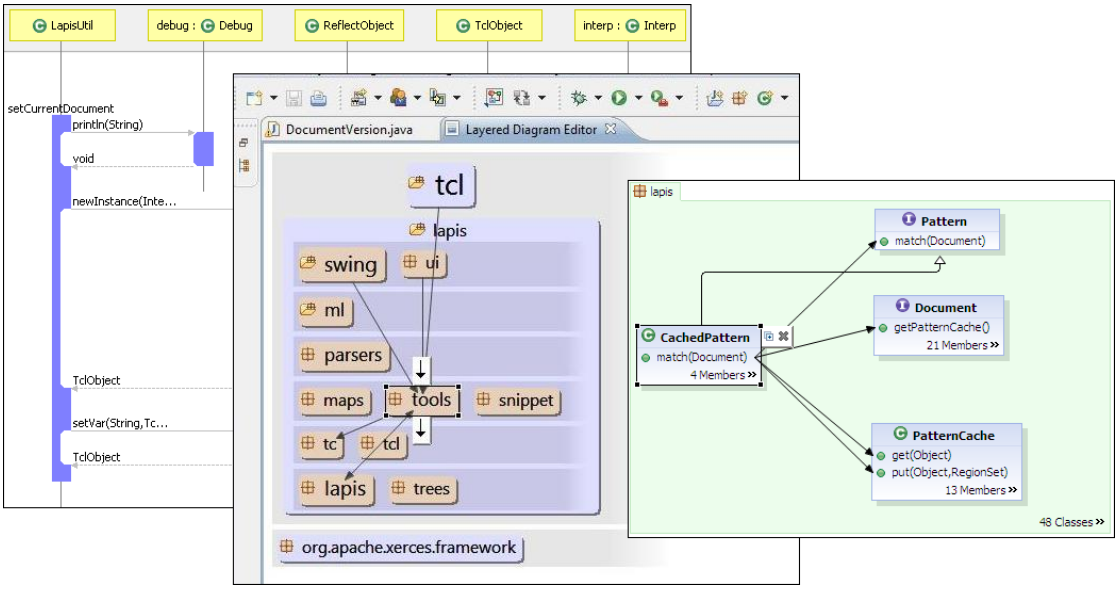

Figure 6.4: Visualizations by Architexa: Sequence (left), "layered" (center) and class (right) diagrams. The components can be expanded at will, relationships between components are shown when a component is hovered over with the mouse cursor.

inCode does not expose its model for reuse. But the drawing framework that is used for the visualizations, jMondrian, is readily available.

While X-Ray provides a reusable model, inCode provides a reusable drawing framework. A tool that supports JEA analysis should provide both an extensible model *and* an extensible set of visualizations. With VERA we aim to produce a framework that supports the implementation of analyses comparable to inCode's.

6.7 MoDisco

MoDisco[7] is an Eclipse plug-in that provides an extensible framework to develop model-driven tools that support the *modernization* of legacy software. This includes quality assurance, documentation, improvement and migration. Each of these tasks requires the user to write programs that use MoDisco's framework, *i.e.*, in order to use MoDisco, one must get to know it in its whole complexity; There is no default model or tools for, *e.g.*, Java software. The general architecture of MoDisco is similar to that of Vera and can be seen in Figure 6.7. MoDisco is a more general framework, within which Vera could have been implemented, but was not.

In contrast to MoDisco, Vera includes default visualizations, whereas MoDisco provides none. Vera can be used out of the box, without writing a single line of code, and is able to perform simple analyses on source code. Furthermore, unlike MoDisco, VERA ships with a base meta-model that should be reasonably close to the users' needs. That is to say, when someone wants to add a custom analysis to VERA, she will already find a lot of information in VERA's existing meta-model, while MoDisco does not provide anything by default.

[7] `http://eclipse.org/MoDisco/`

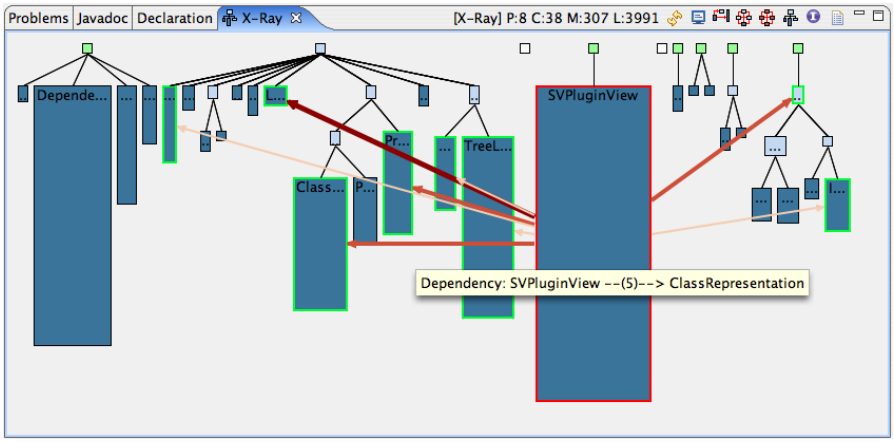

Figure 6.5: System Complexity visualization by X-Ray. Additional information about the classes is shown when the mouse is moved over them; the class figures can be dragged around; filters can be applied.

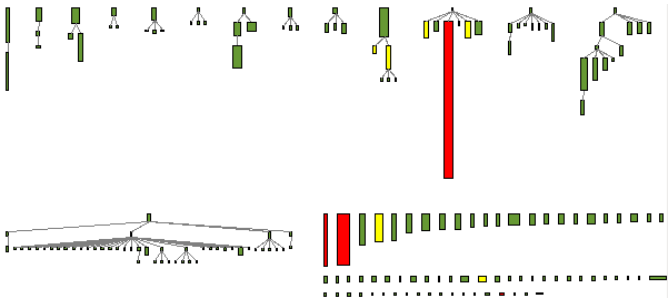

Figure 6.6: A System Complexity-like visualization by inCode. The boxes represent Java classes in the software, colored according to detected possible design flaws.

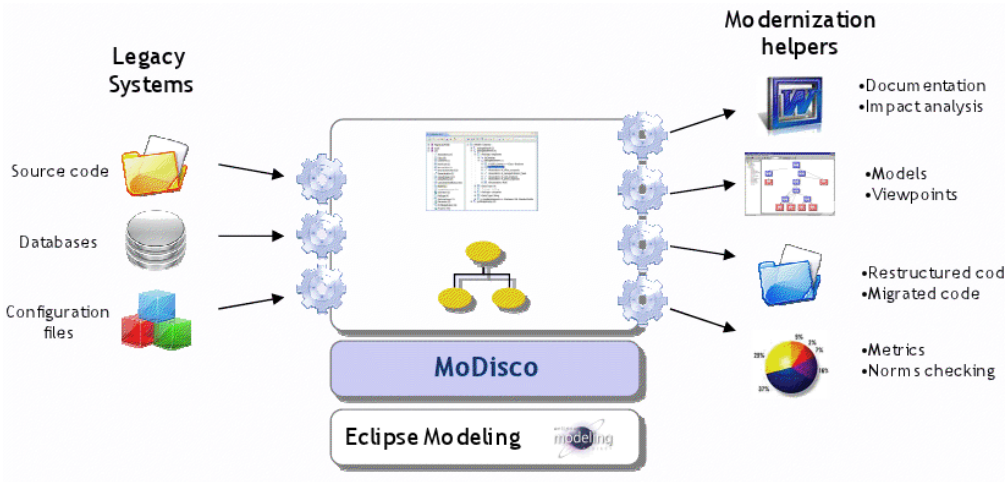

Figure 6.7: Conceptual overview of MoDisco.

6.8 Summary

In this chapter, we presented tools related to VERA with a focus on three properties: *integration* with the development environment to avoid context switch, whether they offer to add custom analyses through *extensibility* and their *multi-language* capabilities, which are essential to JEA analysis. In Table 6.1 we list the tools and give qualitative estimates of those three properties, plus their visualization capabilities.

Overview of Analysis Tool Features				
	SW Visualizations	IDE integration	Extensible	JEAs
Moose	++		++	+
Softwarenaut	++			
Shrimp / Creole	++			
Architexa	++	++		
X-Ray	+	++	+	
inFusion / inCode	+	++		
MoDisco		++	+	+
VERA	+	++	++	+

Table 6.1: Features of the most relevant applications that support software understanding and analysis. (+: decent, ++: good)

7

Discussion: Meta-Model Extensibility

Implementing the FAMIX meta-model was one of the major challenges of this project. Our goal was a meta-model that is easily usable, easily extensible and at the same performs well at runtime. In this chapter, we describe different approaches to implementing a meta-model and discuss them with regard to the following criteria:

Truly extensibile. We want VERA to be modular. A fork is not what we call an extension.

Can be realized in Java. The Smalltalk approach requires language features Java does not have.

Polymorphism in extensions. Extensions can put their new meta-model entities into a type hierarchy.

DRY (Don't Repeat Yourself). A new meta-model entity can reuse the code of its super-entity.

Post-import specialization. Extensions can specialize objects created by existing importers.

Explicit API. Uses Java's static typing to expose a model object's API.

Runtime performance. Heavy use of reflection will slow down the model.

As an example, we again consider the use case where someone wants to add support for Java annotations (see Chapter 5). In the example chapter we mentioned that introducing a new meta-model entity `AnnotationType` would be a natural thing to do. In this chapter, we assume that there is just the FAMIX CORE available and an extension wants to add Java specifics, including an `AnnotationType` entity as well

as an `isEnum` property for Java classes and also properties related to Java Generics. Both Java classes and Java methods are parameterizable, so the Generics related properties should go to a common super-entity. The first common super-entity of `Class` and `Method` in FAMIX CORE is `ContainerEntity` (see Figure 2.2 on page 14).

In the following sections, we discuss different approaches to meta-model extensibility.

7.1 Collaboration and Forking

As we are about to see in this chapter, extensibility is not easy to realize at all in Java. The pragmatic programmer might just abandon meta-model extensions completely and instead choose to modify the meta-model collaboratively in the community of all VERA developers. When people cannot agree on a common model (e.g. a meta-model with totally different entities and properties might be required for a C++ tool), someone can start an independent fork (copy the FAMIX Core implementation and thereafter change the copy independently from the original, also referred to as *clone and own*). There is a high chance that such forks will be incompatible with each other, *i.e.*, only one fork could be installed in an Eclipse instance at a time. Changing the original entities will most likely also require changes in the original analyses. Furthermore, forks do not automatically share bug fixes and other improvements.

We do not want to force every extending party to maintain the code of the whole meta-model, but just the parts they are interested in. Hence we consider this approach a zero option for meta-model extensibility and do not adopt it in VERA.

7.2 Patching existing entity definitions

In the Smalltalk world, extensions to the meta-model can easily be realized by patching the original FAMIX classes. New properties can be added to an existing meta-model entity by adding accessor methods through a patch. Changes in the original entity are immediately reflected in the patched version. In Java however, there is no way to change existing class definitions (at development time). Hence this is not an option for VERA.

7.3 Parallel Class Hierarchies

A naive approach to approximating Smalltalk-like extensibility is to use parallel class hierarchies. The meta-model entities of FAMIX build one huge inheritance tree. We picture that hierarchy vertically, as illustrated in Figure 7.1. For our example extension (adding support for Java to the FAMIX CORE), we might just subclass existing entities in order to specialize them. We add the property `isEnum` to the `Class` entity by creating a new `JavaClass` entity. Similarly, we add new properties related to Java Generics through a new entity `JavaContainerEntity`, a specialization of `ContainerEntity`. We picture the extension's entities to the right hand side, thus they introduce horizontal inheritance. This approach has one major issue: Vertical inheritance is not possible among entities introduced by the extension. I. e.,

the `JavaClass` entity cannot inherit at the same time from `Class` and `JavaContainerEntity`. Java does not support multiple-inheritance.

This approach also suffers from a difficulty which affects all approaches that involve horizontal class inheritance: The extension cannot change how existing importers work. In the example from Figure 7.1, the original importer will probably already create instances of the `Class` entity. The extension would prefer instances of `JavaClass`, but it cannot change the original importer. Thus a custom importer has to be added that migrates existing `Class` objects in the model repository to instances of the more specific `JavaClass`. Alternatively, the core importers could be changed to use an abstract factory for model objects. That would allow the extension to provide its own factory which produces instances of `JavaClass` from the start. Still, in case there is another extension which has its own idea of how `JavaClass` should look like, these two extensions cannot both replace the model objects without breaking each other. Multiple independent extensions cannot coexist when they specialize the same core entities. We call this the *post-import specialization difficulty*.

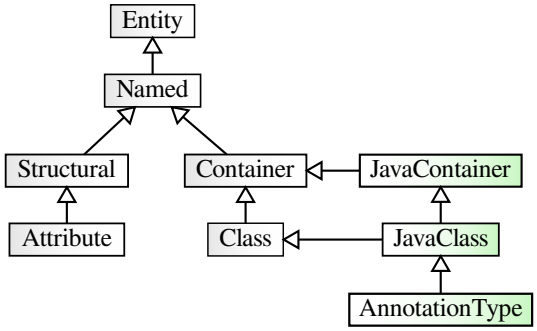

Figure 7.1: Approximating Smalltalk's patching with parallel class hierarchies. The boxes to the right (with green shadows) demonstrate how classes are added by an extending party. This is not possible in Java since it requires multiple-inheritance.

7.4 Horizontal Class Inheritance

As a single-inheritance variation of the first approach, we can leave out vertical inheritance in extensions, as shown in Figure 7.2. Only the original meta-model forms a hierarchy of entities. Extensions then specialize some of the original entities by subclassing them. There is no relation at all between the extension's new entities. With this approach, extensions can only specialize the leaves of the original inheritance tree (and introduce new leaves); polymorphism in the extension's new entities is restricted to the inheritance relationships in the original meta-model. For example, suppose the two original entities `Class` and `Method` are subclassed in the extension by `JavaClass` and `JavaMethod`, respectively. Now we want to write an analysis that operates on all generic objects, *i.e.* on both classes and methods. But with this approach, we cannot refer to "all generic objects". We must write our analysis twice, once for Java classes and once for Java methods. Since we have no multiple-inheritance, we cannot just add the information about Java Generics to a common superclass of `JavaClass` and `JavaMethod` — the

original meta-model (*i.e.*, the common superclass `ContainerEntity`) is untouchable for extensions. If we introduced a new entity `JavaContainerEntity` and added the Generics related information there, neither `JavaClass` nor `JavaMethod` could make use of it, since they are only subclasses of `ContainerEntity` but not of `JavaContainerEntity`. When introducing a new entity (*e.g.*, `AnnotationType`), it can either horizontally specialize an existing entity (as seen in the figure) or use vertical inheritance to allow for polymorphism. Furthermore, this approach suffers from the post-import specialization difficulty.

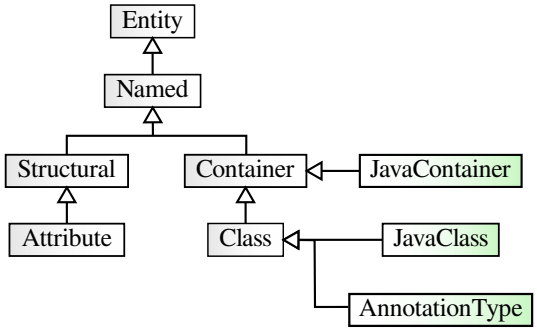

Figure 7.2: Leaving out vertical inheritance in the extension as a single-inheritance variation to the first approach. VERA's core Java importers still create instances of the core classes on the left. Polymorphism in the extension is lost.

7.5 Parallel Interface Hierarchies and horizontal Class Inheritance

This approach leverages the fact that the restriction of single-inheritance does not apply to Java interfaces. The parallel entity hierarchies are built using interfaces. Compared to the first approach, this solution allows for restricted subtype polymorphism. For example, let `IClass` be a sub-interface of `IContainer` in the original meta-model. An extension now specializes these two entities with two new interfaces `IJavaClass` and `IJavaContainer`. The `IJavaClass` can at the same time extend `IJavaContainer`, parallel to the original entities. The implementing classes `JavaClass` and `JavaContainer` inherit (horizontally) from the respective original meta-model entities. Figure 7.3 illustrates this example. A restriction of this approach is that there is still no vertical class inheritance, *i.e.*, while `IJavaClass` specializes `IJavaContainer`, `JavaClass` does not specialize `JavaContainer`. As a consequence, new methods which are defined in `IJavaContainer` have to be implemented in both `JavaContainer` and `JavaClass` (as well as in every other sub-entity of `JavaContainer`, if any). Such code duplication can be reduced through composition: Create a helper class which implements the new functionality and delegate to such a helper object from both `JavaContainer` and `JavaClass`. Like this, only the delegation code is duplicated, while the actual new functionality is implemented but once. This is WET (Write Everything Twice), the opposite of DRY.

This approach still suffers from the post-import specialization difficulty.

There is still no polymorphism among the concrete meta-model classes, but it allows us to implement analyses against common super-interfaces of newly introduced entities. Something like this is what we chose for our implementation of the FAMIX meta-model. Our implementation slightly differs from this

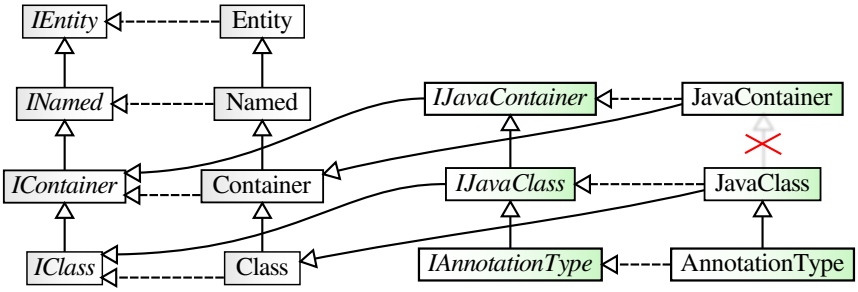

Figure 7.3: Multiple-inheritance is possible with interfaces. Though still no polymorphism (*i.e.*, code duplication) in the concrete extension classes. Interfaces are depicted by *italic* names.

approach in that we did not add setter or helper methods to the interfaces. Especially the helper methods are helpful when writing analyses, though, which is why we ended up writing our visualizations against the concrete meta-model classes, not the interfaces. Extensions might want to add more than just the property getters to their interfaces. In the following approaches, we assume that meta-model entity interfaces contain *all* methods.

7.6 Purely declarative Meta-Model Definition

We considered the possibility of specifying the meta-model completely declaratively. With this approach, the meta-model entities are defined in XML in an extension-point configuration (or in an MSE file). The concrete entity classes are generated at runtime and are a combination of all functionality defined in all present extensions. There are two issues with this approach.

First, our meta-meta-model FM3 contains not enough information about how properties relate to each other. For example the `belongsTo` property is marked as "derived" from other properties. But the information about how it is derived lies solely in the meta-model source code; FM3 does not model different kinds of derivation. (The `belongsTo` property is in fact aliased by other properties in several places near the leaves of the meta-model inheritance tree.) We would need a more expressive meta-meta-model in order to generate meta-model code. Also, the meta-model should provide helper methods (which are not just plain getters and setters), which can hardly be generated. Maybe declaratively including static methods on external classes as helper methods would be an option.

The second issue is that the meta-model API is implicit, *i.e.*, not described by a Java class or interface but only in XML. But we (and the compiler) need to know a model object's API at development time. This is required to program both importers and analyses, in fact in every place where model objects appear in the source code. That issue can presumably be solved by generating a set of interfaces for the meta-model entities for use at development time.

With such an approach, one can circumvent class inheritance related issues, *i.e.*, restricted polymorphism and code duplication, and probably also post-import specialization. A major challenge with this approach is implementing it in an easily understandable and usable fashion.

7.7 Adapters for all Properties

The next approach avoids class inheritance issues by means of a generic model object. This object itself does not have any properties. Yet it can represent an instance of an arbitrary meta-model entity, both of VERA's base plug-in and all extensions. This is accomplished through stateful adapters. An adapter provides methods specific to a certain meta-model entity, *i.e.*, getter and setter methods for one or multiple properties. It also stores those properties' values, that is why we call the adapters *stateful*. Every model object has its own set of adapter instances. Every extension can add its own adapter to a model object (at runtime, through a method call). Everyone can retrieve an arbitrary adapter (think: an arbitrary property) of a model object, as long as they know the adapter's class. This provides great flexibility for extensibility. Figure 7.4 gives a simplified class diagram of this approach.

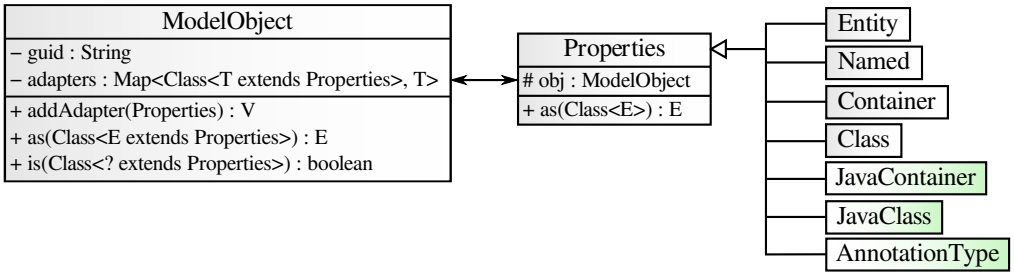

Figure 7.4: No need for object migration or code duplication with this adapter solution. But the API of a model object is not explicit and there is no polymorphism among entities at all.

Since this approach completely avoids inheritance, there are no issues concerning code duplication and post-import specialization. But there is also no polymorphism among meta-model entities at all! In fact, the meta-model is completely implicit with this approach. There might be a specification somewhere on a sheet of paper, which defines the set of adapters that make up each meta-model entity. But there is no guarantee that a given model object provides a well-defined set of adapters. Any adapters can be added at any time. In other words: the benefits of static typing are lost with this approach. When developing an analysis, people would "just have to know" which model objects refer to what kind of source code artefacts (based on the model object's unique identifier), and hope that the desired adapters have been added. If a developer believes that all adapters are well set, she can just query the model repository for any objects that have adapters of certain types. That might actually be a convenient way to write analyses. Still, the fact that the documentation of the meta-model is external to the source code makes it cumbersome to work with and maintain. The fact that there is no explicit meta-model, let alone a meta-meta-model, also makes it much harder to programmatically reason about the model.

An interesting variation of this approach involves a declarative meta-model. The meta-model entities are defined though an extension-point. The specification is written in XML in the `plugin.xml` file. Every entity is assigned a String identifier and a collection of adapters its instances must provide. Model objects are created by a factory which at the same time adds all adapters specified in VERA's core plug-in as well as in any installed extensions. After that, no further adapters can be added. Of course this cancels the runtime

flexibility of the adapters approach. But based on the declarative meta-model definition, programmatic reasoning about the model is possible again. And the definition is at the same time a documentation which is guaranteed not to become out of date. A meta-model browser can be used which combines such documentation of all extensions that are available at development time. This would of course require implementing such a meta-model browser, as well as significant changes to the Fame implementation (which instantiates the FM3 meta-meta-model).

7.8 Using `java.lang.reflect.Proxy`

The Java reflection package offers a way of dynamically creating "proxy" classes whose instances can be cast to a given set of interfaces. That set of interfaces does not have to be known at compile time, but only when the proxy object is created. Adding more interfaces after creation is not possible, though. All method invocations on a proxy instance are passed on to an *invocation handler* which then decides how to proceed further.

With this approach, an extension can add an interface to the meta-model and specify that it should contribute properties to an existing entity, *i.e.*, that the new interface stands for the "same entity" as an interface from the FAMIX Core. We can collect all "same entity" interfaces from all VERA extensions that are installed in an Eclipse application and then use a generic invocation handler which delegates the method calls. The delegation target for a new interface should be provided by the extension that introduced it. Whenever model objects are created (*e.g.*, by VERA's original Java importer), they are proxies which already know all properties specified by extensions. With this approach, we do not implement any class that represents a "model object", instead every meta-model entity has its own, dynamically generated proxy class. The only concrete classes are the delegation targets. We call them "property handlers" since their instances store the values of one or more properties as well as the methods to access them. See Figure 7.5 for a class diagram.

The class of a proxy object is only available at runtime, yet we (and the compiler) need to know its API in order to program against it. This is solved by casting proxy objects to one of the interfaces they implement before calling any methods on them. In VERA's core plug-in, the model object will be cast to the core version of its meta-model entity. In a VERA extension, the same object will be cast to the extension's more specialized version.

Property handlers must be separated from the meta-model type hierarchy. If a property handler just implemented an entity interface, it would have to implement all methods defined in all entity-super-interfaces, too. We can avoid that by factoring out *all* method definitions of an entity interface to a new, additional super-interface thereof. That *property interface* is then independent from the meta-model type hierarchy. Like that, a property handler can implement just the property interface, which makes it independent from entities. The property interface can also be split up into multiple property interfaces, each representing exactly one property (including setter, getter and convenience methods for that property). Property handlers can implement an arbitrary number of property interfaces. While this approach increases complexity compared to other approaches, it also has its benefits. For example, it allows one to reuse

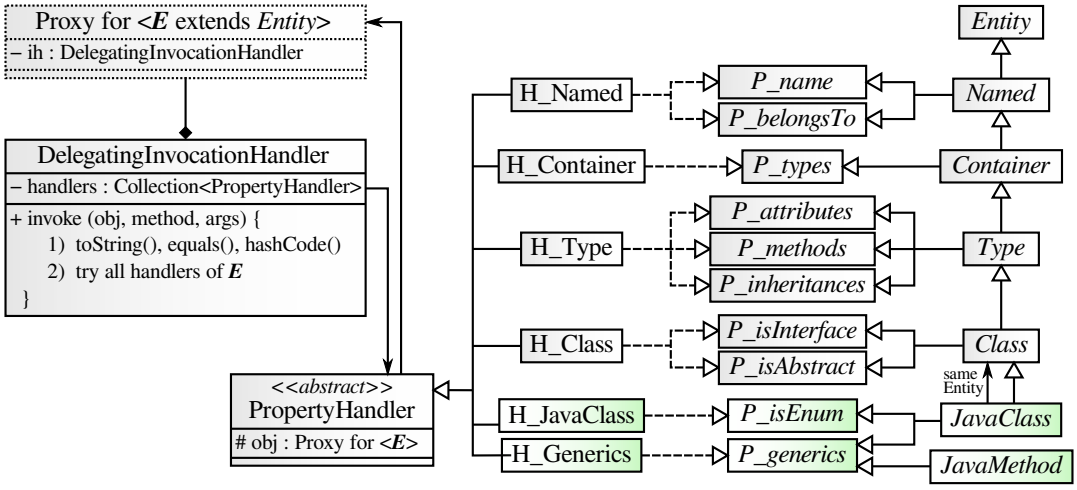

Figure 7.5: By allowing an extension entity to explicitly specialize an existing entity ("same Entity" arrow in the picture) and by the use of `java.lang.reflect.Proxy`, both polymorphism and an explicit API are possible. This increases complexity (additional property interfaces). Only a simplified meta-model structure is shown; superinterfaces of `JavaMethod` are omitted. It is a "same entity" as a `Method` core entity, which indirectly extends `Container`. VERA's core plug-in must consolidate all the "same entities" from all installed extensions.

a single property (including its handler class) on multiple entity interfaces in separate branches of the meta-model. No need to artificially put the property's methods in a common super-entity and then throw an `OperationNotSupportedException` from every subclass to which the property does not apply.

For all of this to work, the proxy approach also relies on a declaration of the meta-model extensions in the `plugin.xml`. That declaration lists the entity interfaces (*e.g.*, `Class`), interfaces which specialize existing entities (*e.g.*, `JavaClass`) and the property handler for each property interface.

Programming analyses with this approach is somewhat similar to using adapters. But instead of retrieving an adapter for a property from the model object, it is cast to the meta-model entity interface that includes the desired property (can be done by a convenience getter on the model repository: `repo.get(Method.class, "some_method_guid")`). A major difference between the adapter approach and the proxy approach is that only the latter allows for polymorphism among meta-model entities.

One issue with this approach is that performance problems may arise due to heavy use of reflection.

7.9 Nesting Adapters: The Property Onion

This approach organizes property handlers into a hierarchy of nested adapters, which are realized as proxies. This is very similar to the proxy approach explained above, as we can see in Figure 7.6. This approach also provides an explicit model API, and at the same time adds the runtime flexibility of the adapters approach; there is no need for post-import specialization, and no need for a notion of "same entity". The property onion approach uses even more reflection and is more complicated in its internals.

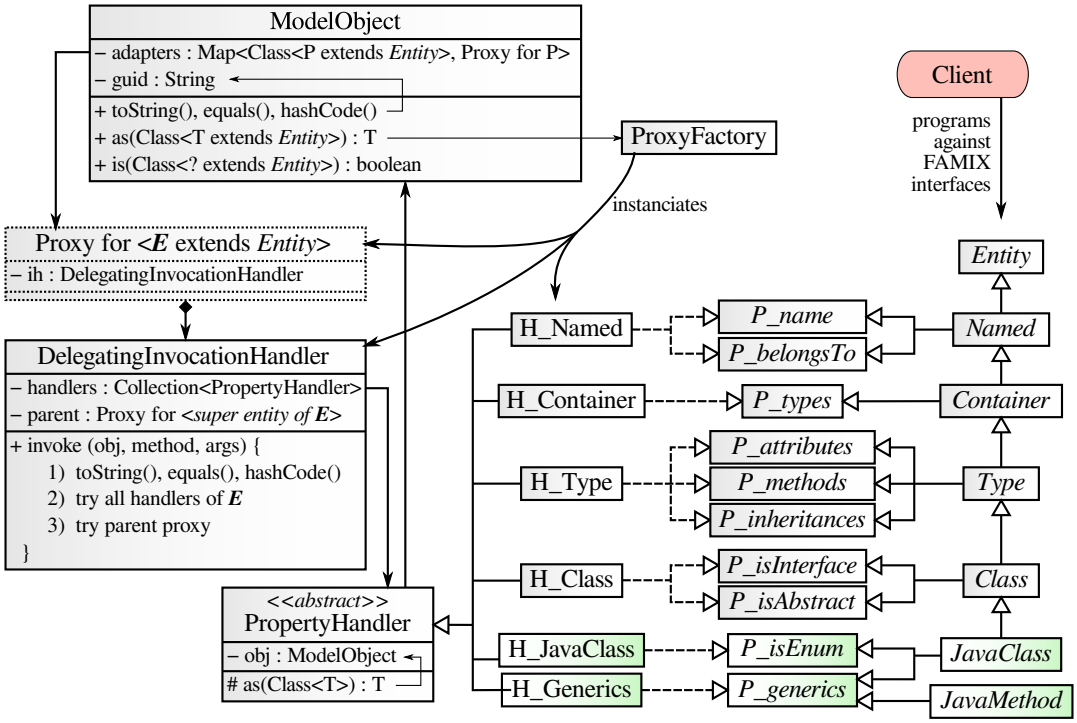

Figure 7.6: The property onion approach allows to add properties to existing entities without class inheritance problems and at the same time provides an explicit API through the entity interfaces.

But the analysis developer does not have to know about its internals. She can just use `modelObject.as(MyNewEntityInterface.class)` at any time to get a more specialized view on a model object. Just add a new interface through a meta-model extension point, specify which handler classes should be used for its property interfaces, and a new adapter is ready for use. There is also no need to migrate existing objects, just adding an adapter (through a call to `as()`) is enough and does not affect the original behaviour of the object at all. In comparison to the last two approaches, the Property Onion requires less declarative programming inside the `plugin.xml` file. Only the connections from property interfaces to property handlers have to be set up.

With this approach, a model object is not an instance of a meta-model entity class, similar to the adapter approach. In order to find out whether a model object represents an instance of a certain meta-model entity, one can call the test method `is(Class<?>)`. This can also be embedded in a convenience method of the repository `repo.get(Method.class, "some_method_guid")`, which takes as an additional argument the class of an expected entity interface and returns the corresponding adapter (or raises an exception).

7.10 Summary

In this chapter, we discussed several approaches to implementing a meta-model. We have seen that Java's static typing in combination with the lack of facilities to change existing class definitions imposes very

particular constraints. Accordingly, VERA's current meta-model implementation provides only restricted extensibility.

With the last four approaches, we discussed ways of circumventing those constraints using either code generation or reflection. Particularly the impact of using reflection on the meta-model's runtime performance would be interesting to investigate.

8 Summary

In this master's thesis we lay a base for software comprehension and analysis tools based on Eclipse. In the course of the work for this thesis we found many related tools. Among these, our tool VERA is the only one that is at the same time integrated into the development environment, capable of handling heterogeneous sources and extensible.

Using our tool, we implemented some basic analyses as well as one specific to Java Enterprise Applications, and embedded our visualizations in Eclipse. From a user perspective, VERA is easy to install and use. The import takes some time, but apart from that, usage is seamless. From the viewpoint of an extending party, adding a custom visualization is very easy, too. The meta-model is not quite as easy to extend as we would have liked it to be, therefore we discussed alternative approaches that might improve this point.

Adding a custom visualization to Eclipse is indeed facilitated by VERA, but creation of that visualization is still completely up to the extending party. We evaluated Draw2d and jMondrian as drawing frameworks and found that Draw2d is quite usable.

8.1 Lessons learned

The most valuable lesson we take from the work for this thesis is that open source is not equal to modularity. For example, we found many related analysis tools, but none of them provided a decent meta-model that we could reuse. With jMondrian and Fame we did use some other open source projects, but that required

forking them. They were not specifically built to be reused/extended in the exact fashion we intended to use them. While we still believe that modularity is worth striving for, we came to be a bit more pragmatic about reusing existing software. Not everyone designs their software to be reused, so sometimes reusing means forking.

Other things we learned are mainly technical details about implementing an Eclipse plug-in, a meta-model and visualizations. From these things, we would like to stress one in particular, although it is not a very novel insight at all: Writing informative Javadoc matters! Programming against an API is much more productive when its Javadoc explains what the individual classes and methods do, how they integrate with the rest of the API and gives (links to) explanations of technical terms.

Why use FAMIX as a meta-model when Eclipse maintains an internal model of the Java source code already? There are several reasons not to use Eclipse's Java model. First, it is very minimalistic concerning cross-links. For example, a class knows its superclass but not its subclasses. That makes it fast to update, but also slower to query. Also, the API is not intuitive at all and poorly documented. But the most important point is that it is a model solely for Java; extending or changing it to accommodate the heterogeneous nature of JEAs is not possible — simply because there is no way of patching class definitions in Java. Therefore we decided to implement our own model. By using FAMIX, we can furthermore benefit from the know-how in the research group and potential interoperability with other analysis tools developed by the research group (*i.e.*, Moose via MSE exchange).

Lastly, we would like to mention a point about work organization. Sometimes, we felt urged to justify ourselves, which made writing the thesis very hard at some points. That is, when we realized that what we implemented is not necessarily what is best, because we had not considered some points before. At the time of writing, when we had to make our thoughts very explicit, sometimes we discovered a significant point we had not thought of before. Or there was another related work which we suddenly discovered, which forced us to re-interpret all of our work. When that happened, it was always very frustrating. When beginning to work on a new field, one can of course not know all that is relevant. A critical analysis of a more experienced peer could help at this point. But when no expert is around, one should mentally prepare for significant setbacks.

8.2 Future Work

For the meta-model extensibility, it would certainly be interesting to compare our current implementation to the "property onion" approach described above, considering both ease of extensibility and performance.

The VERA specific help contents for the Eclipse user should be improved.

Creating custom analyses for VERA still requires writing an Eclipse plug-in. We would like to further ease the creation of visualizations by providing something like the visualization scripting capabilities of Moose. Coding with the Java port of Mondrian, *jMondrian* [8], is much more verbose than with the Smalltalk version due to the static type checking of Java. Therefore, it is not suitable as a scripting language. A possible way of making visualizations scriptable in VERA would be to create an external domain specific language (DSL), for example using the interpreted language JRuby.

8.2.1 User Interaction tweaks

In this section we discuss some ideas of how VERA and its extensions could choose to interact with the user.

Problem Markers. We mentioned that displaying visualizations is *one possible* way of leveraging the information in the model and/or analysis results. Another (complementary) way would be to use problem markers. These are yellow or red icons which are displayed in the margin of the Eclipse's source code editor. They provide some text which explains why there is a marker. inCode for example uses them to mark potential design flaws. Problem markers can be produced even for results that are not directly related to a particular location somewhere in a source file. Such markers are added to the Eclipse project. Analysis results do not always have to be reflected in the user interface. They can also be artefacts like files which are then used outside Eclipse, possibly by an existing analysis tool or for documentation. Of course that defies the benefits of IDE integration, but it is possible.

Improved user interaction in visualizations. As for the existing visualizations, we think that they provide a decent user experience. Allowing the user to move visualized model objects around would certainly be an improvement. In order to improve support for as-needed software comprehension strategies, filtering capabilities could be added. This also applies to the model browser. Another helpful addition would be a search field in which model objects can be located by name.

Import scope. Vera always imports exactly one Eclipse project at a time. This makes it easy to cache imported models and to write visualizations for one project. But sometimes, people might want to analyze code from multiple related Eclipse projects at once; or the code inside a single Java package. There are work-arounds to circumvent the one-project-limitation. For multi-project import, one can create an additional Eclipse project which combines multiple projects by including all their source folders. Reducing the import scope to less than a project is not possible at the moment, but an analysis may always opt to ignore parts of the imported model. Just importing a part of the sources would speed up the import process, making VERA even more attractive as an analysis and software comprehension platform. It would probably help to add Eclipse specific entities to the meta-model, which represent, *e.g.*, projects and their dependencies, source folders, libraries, and so on.

Appendices

A Quick Start Guide

You can obtain VERA by installing it like any other Eclipse plug-in. There are two update sites available:

- Stable releases:
  `http://scg.unibe.ch/download/Vera/`
- Development snapshots:
  `https://www.iam.unibe.ch/scg/svn_repos/Students/krebs/Vera_Update_Site/`

We recommend that regular users install VERA through the stable update site. Just install "Java Enterprise Visualization" and you are ready to use VERA. All the other installable options are mainly useful for people who intend to extend VERA. Developers may want to install the latest development snapshots instead of the stable version. All plug-ins include the sources, too (except for tests). Alternatively, you can obtain the sources from our

- Subversion repository:
  `https://www.iam.unibe.ch/scg/svn_repos/Students/krebs/`

As soon as VERA is installed, go to the menu

```
Window -> Show view -> Other...
```

and select the following (type "vera" in the search field to find it faster):

```
Software Analysis -> Vera visualizations
```

The visualizations view will appear as shown in Figure 3.2 on page 19. Now select a Java project, for example in the Project Explorer or Package Explorer view. Then click on one of the icons in the visualizations view. Vera will now import the selected project and visualize it.

B Scaffold for the example importer

In the extension example in Chapter 5 we introduce a new importer. Here is a scaffold of its implementation.

```
public class AnnotationInstanceImporter extends JavaASTImportVisitor {

    /** The model repository to which we add newly created model objects. */
    private IProjectModelRepository repo;

    /** A non-argument constructor, required for automated instantiation by Eclipse */
    public AnnotationInstanceImporter() {}

    /** Invoked by Vera every time before an AST is visited. */
    @Override
    public void setWorkingModel(IProjectModelRepository modelRepository) {
        this.repo = modelRepository;
    }

    @Override
    public boolean visit(NormalAnnotation node) {
        createAnnotation(node);
        return true;
    }
    @Override
    public boolean visit(MarkerAnnotation node) {
        createAnnotation(node);
        return true;
    }
    @Override
    public boolean visit(SingleMemberAnnotation node) {
        createAnnotation(node);
        return true;
    }

    private void createAnnotation(Annotation node) {
        // create a new instance of AnnotationInstance and add it to this.repo
    }

}
```

Listing 11: An example scaffold for the annotation instance importer from the example chapter.

C Licensing of Plug-ins and Features

All our sources are subject to an open-source license. Vera itself we put under a BSD 3-clause license. The drawing framework jMondrian remains under its original BSD 2-clause license. Fame also keeps its dual license (GPL + LGPL). Of course every one who uses Vera should be informed of these licenses. Eclipse gives a plug-in developer the possibility to include a license text which will be displayed to the user during the installation process. Installation can only be completed when the user agrees to the licenses of all plug-ins that are to be installed. This includes licenses of plug-ins that are automatically installed due to plug-in dependencies. There are four steps to publishing a plug-in in a properly licensed fashion.

1. Wrap your plug-in in a feature. There is no way of attaching license text directly to a plug-in. Of course a file containing the license text should be included in the plug-in. But that will not be shown to the user during the installation process. Only features can do that.

2. Add your license text to the feature. A fresh feature project contains exactly two files: the `build.properties` and the `feature.xml`, where the latter contains all the meta-information. The Eclipse PDE provides a dedicated editor for this special file, so you do not have to know its XML structure at all. Copy the license text into `Information -> License Agreement` and you are done. Consider filling in all the other fields of the feature configuration for a good end user experience.

3. Make sure that your plug-in can only be installed through the feature containing the license agreement. This is only an issue when you have multiple features on the same update-site. When another feature requires your plug-in, you should specify this by configuring that other feature to depend or include your *feature*. If you just added your plug-in directly to the other feature, the plug-in would be published under that feature's license agreement. But when a feature depends on or includes other features, the license agreement of all involved features are respected, *i.e.*, shown to the user during installation.

4. When you change license texts or dependencies between features, you should always increase their version number. Only then is the update-site's meta-data updated properly. When you are just experimenting during development, constantly increasing the version numbers can get really tedious. In that case you might want to create the update-site from scratch instead. Keep a backup copy of the `site.xml` as it was *before* the first build of the update-site. In this initial state, the `site.xml` does not yet contain timestamps in the version numbers of plug-ins and features, but a "`.qualifier`" postfix instead. That placeholder causes Eclipse to build the corresponding feature as if it was the first time ever it is built. When you change the `site.xml` at a later point, copy over the backup version, then modify, then back it up again, then build the update-site.

Also think of a good strategy of how you increase the version numbers of all your plug-ins and features. We like to keep the version number of a feature synchronized with its plug-ins. As a guideline, keep in mind that changes will only be fetched by users who installed something from your update-site when you increase the version numbers of all involved plug-ins and features.

Bibliography

[1] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, and Eric Jendrock. The J2EE 1.4 tutorial, December 2005.

[2] M. Cherubini, G. Venolia, and R. DeLine. Building an ecologically valid, large-scale diagram to help developers stay oriented in their code. In *Visual Languages and Human-Centric Computing, 2007. VL/HCC 2007. IEEE Symposium on*, pages 157–162, 2007. Available from: `http://dx.doi.org/10.1109/VLHCC.2007.19`, `doi:10.1109/VLHCC.2007.19`.

[3] Joseph W. Davison, Dennis M. Mancl, and William F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, pages 44–54, 2000.

[4] Tom DeMarco and Tim Lister. Programmer performance and the effects of the workplace. In *Proceedings of the 8th international conference on Software engineering*, ICSE '85, pages 268–272, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press. Available from: `http://dl.acm.org/citation.cfm?id=319568.319651`.

[5] Erich Gamma and Kent Beck. *Contributing to Eclipse*. Addison Wesley, 2003.

[6] Canfora Gerardo and Di Penta Massimiliano. New frontiers of reverse engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society. `doi:10.1109/FOSE.2007.15`.

[7] inCode. inCode — eclipse plugin for code analysis, 2009. http://www.intooitus.com/inCode.html. Available from: `http://www.intooitus.com/inCode.html`.

[8] JMondrian. JMondrian — Java implementation of the Mondrian information visualization framework, 2009. http://loose.upt.ro/reengineering/research/jMondrian. Available from: `http://loose.upt.ro/reengineering/research/jMondrian`.

[9] Adrian Kuhn and Toon Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, pages 57–66, 2008. Available from: `http://scg.unibe.ch/archive/papers/Kuhn08cFame.pdfhttp://www.comp.lancs.ac.uk/~bencomo/MRT/MRT2008Proceedings.pdf`.

[10] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003. Available from: `http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf`, `doi:10.1109/TSE.2003.1232284`.

[11] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. Available from: `http://www.springer.com/alert/urltracking.do?id=5907042`.

[12] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM. `doi:10.1145/1134285.1134355`.

[13] Mircea Lungu and Michele Lanza. Softwarenaut: Exploring hierarchical system decompositions. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 351–354, Los Alamitos CA, 2006. IEEE Computer Society Press. `doi:10.1109/CSMR.2006.52`.

[14] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma: An integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 77–80, 2005. Tool demo.

[15] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press. Available from: `http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf`, `doi:10.1145/1148493.1148513`.

[16] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, 2005. ACM Press. Invited paper. Available from: `http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf`, `doi:10.1145/1095430.1081707`.

[17] Klaus Ostermann. personal communication, 2012.

[18] Fabrizio Perin. 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)*, pages 291–294. IEEE Computer Society Press, October 2009. Available from: `http://scg.unibe.ch/archive/papers/Peri09aEnablingevolutionOfJEAs.pdf`, `doi:10.1109/WCRE.2009.45`.

[19] Fabrizio Perin, Tudor Gîrba, and Oscar Nierstrasz. Recovery and analysis of transaction scope from scattered information in Java enterprise applications. In *Proceedings*

of International Conference on Software Maintenance 2010, September 2010. Available from: http://scg.unibe.ch/archive/papers/Peri10aTransactionRecovery.pdf, doi:10.1109/ICSM.2010.5609572.

[20] Dario D. Salvucci and Peter Bogunovich. Multitasking and monotasking: the effects of mental workload on deferred task interruptions. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 85–88, New York, NY, USA, 2010. ACM. Available from: http://doi.acm.org/10.1145/1753326.1753340, doi:10.1145/1753326.1753340.

[21] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 21–. IBM Press, 1997. Available from: http://dl.acm.org/citation.cfm?id=782010.782031.

[22] Margaret-Anne Storey, Casey Best, and Jeff Michaud. SHriMP Views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[23] Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44:171–185, 1999.

[24] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[25] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167. IEEE Computer Society Press, 2000. Available from: http://scg.unibe.ch/archive/papers/Tich00bRefactoringMetamodel.pdf, doi:10.1109/ISPSE.2000.913233.

[26] A. von Mayrhauser and A.M. Vans. From code understanding needs to reverse engineering tool capabilities. In *Computer-Aided Software Engineering, 1993. CASE '93., Proceeding of the Sixth International Workshop on*, pages 230 –239, July 1993. doi:10.1109/CASE.1993.634824.

Erklärung

gemäss Art. 28 Abs. 2 RSL 05

Name/Vorname: .....

Matrikelnummer: .....

Studiengang: .....

Bachelor ☐ Master ☐ Dissertation ☐

Titel der Arbeit: .....

.....

.....

LeiterIn der Arbeit: .....

.....

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe o des Gesetztes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

.....

Ort/Datum

.....

Unterschrift