Java Wiretap

Extracting Feature Execution Models for Reverse Engineering

Julien Fierz
Supervised by: Orla Greevy
University of Bern, Switzerland
Software Composition Group
June 2007

Wiretap is a profiler tool that instruments Java applications and allows a reverse engineer to directly control the extraction of behavioral data units (features) and the level of detail of the dynamic data in a well-defined format. Wiretap captures fine-grained dynamic data such as message sends (activations), field access and instance tracking.

The extracted model can then be used by a reverse engineering platform for further analysis in reverse engineering environments like Moose [11] extended with DynaMoose [5] for Feature Analysis and Object-Flow Analysis [9, 10]). Wiretap allows the user to trace different triggerable actions of an application, each representing different features, which can then be treated as distinct feature entities when performimg feature analysis. To control the large volume of detailed dynamic information, Wiretap allows selective instrumentation of an application at package level.

Contents

List of Figures

Chapter 1

Introduction

Reverse engineering, the process of extracting high level abstractions of a software system, is very important for program comprehension [2]. It is difficult to understand large applications only by looking at the code. Furthermore, due to language features of object-oriented programs such as polymorphism and dynamic binding, a purely static view of source code does not tell us how the program behaves at runtime.

Researchers have developed many analysis techniques to model an application in several ways. Recently researchers are focusing on features of an application which represent behavioral units of an application [4,5]. We adopt the definition of a feature as a unit of behavior of an application triggered by the user [4]. By treating features as first-class entities, we can analyse these units of behavior in a model of the system.

The goal of our work is to facilitate dynamic analysis of Java programs by providing a tool that instruments Java applications and allows a reverse engineer to directly control the extraction of behavioral data units (features) and the level of detail of the dynamic data in a well-defined format. The extracted model can then be used by a reverse engineering platform for further analysis. Our target reverse engineering environment is Moose [11] extended with DynaMoose [5] for Feature Analysis and Object-Flow Analysis [9, 10]).

A key goal of our profiler tool is that it must be able to let the user trace different actions, each representing different features. These can then be treated as feature entities when performing feature analysis. Each execution trace captures large amounts of detailed dynamic information, but this means that a large amount of data must be handled. Thus another goal of our profiler tool is to the let the user control the level of detail of the dynamic information to be captured.

1.1 Reverse Engineering with Profilers

Profilers are very useful tools to analyze programs (in our case Java programs). They are built with a focus on performance analysis. Most of them do allow extraction of dynamic data, but they do not support flexible extraction or definition of the format of the data. This is the main limitation of these tools. The profilers work fine for detecting performance bottlenecks and they are able to visualize dynamic data in some way. However, for reverse engineering concerns, we need be able to extract data that we can use to create and manipulate our own models of execution data. For Feature Modelling (i.e. modelling observable activities of a system that are initiated by the user) we need to be able to control the collection of dynamic data and associate parts of it with features.

1.2 Java Wiretap (JWT)

We reviewed a range of open source Java profilers. None of these meet our requirements to provide the level of detail or flexibility to obtain the data that we need to build our behavioral model of a program. After reviewing the existing solutions, we decided that it would be better to implement our own profiler to meet our requirements. We didn't want to start from scratch so we decided to build on an existing solution. Java Wiretap (JWT) is the resulting program. It mainly observes the communication between objects at runtime and produces output in a well-defined format that can easily be interpreted and read by reverse engineering tools, in our case Moose.

Moose has already tools which supports reverse engineering. The target analysis tool of the behavioral data is *DynaMoose* (see Figure 1.1 (p.6)), a dynamic analysis capability built on top of Moose. It allows one to analyze and visualize information based on the Dynamix metamodel [5]. DynaMoose is language independent, as reverse engineering analysis is performed on a metamodel of a system. It has also been used for models extracted from Smalltalk systems. A key contribution of our work is that we extract and write the data gathered by our Java profiler to a format Moose understands (specified by the metamodel). Wiretap gathers all the information Moose needs for behavioral analysis (in paricular for feature analysis) and writes it into a MSE-format file, which can be loaded in Moose.

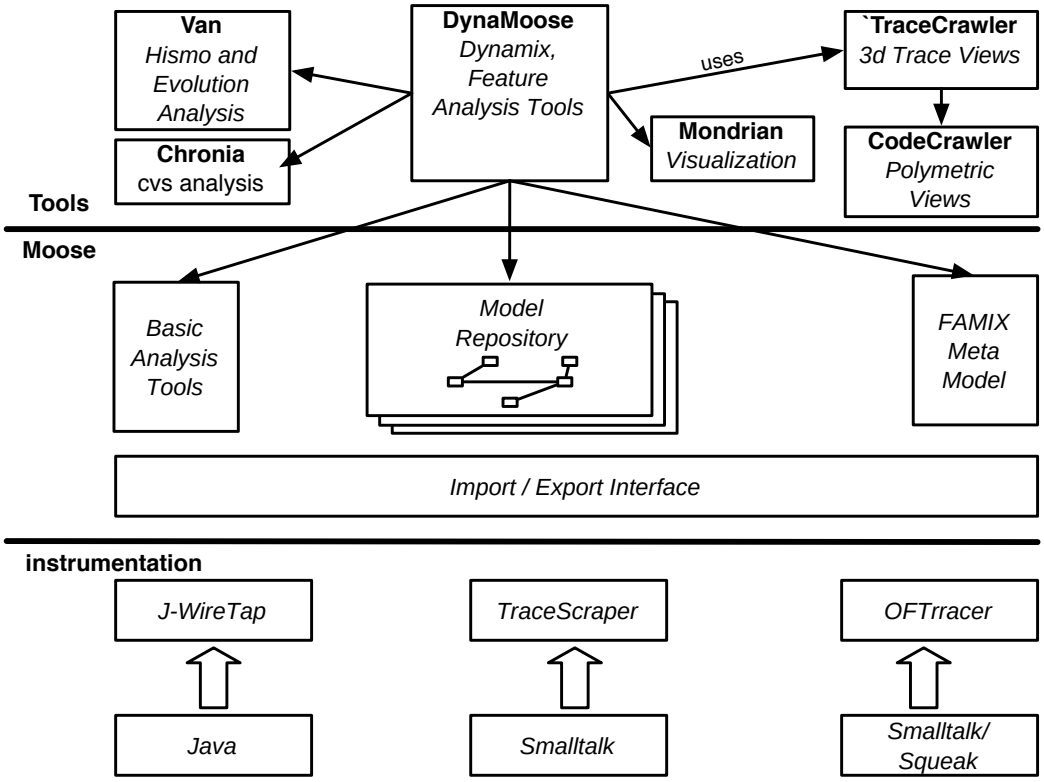

Figure 1.1. The DynaMoose Model

1.3 Goals

As mentioned, the primary goal of this work is to provide a flexible, configurable Java profiler that extracts detailed behavioral data of an object-oriented program to a well-defined format. The following

is a more detailed list of our goals:

- Let the profiler gather behavioral data: method calls, object allocation, field access, ...
- Most profilers do not track the instances, they only provide e.g. the number of allocations of a class. This means they do not provide on which instance of a class a method has been called; they only visualize a simple call graph. We wanted to keep track of all created instances. The problem here is that this is very space-consuming.
- A feature is a unit of domain functionality from the user's perspective. We wanted to give the profiler the possibility to track what events occur in a specific feature, which allows e.g. to determine which objects have been used in one feature, or which objects are shared by different features.
- A simple GUI to start and stop the profiler (as mentioned with support for feature extraction) and write the output.
- Eclipse is a widely-used developement environment which is very easy to extend. We decided to create an Eclipse plugin that launches an application directly with the profiler.
- The behavioral data should be described in MSE, as it is loadable into Moose for dynamic analysis and feature analysis.

Chapter 2

Java Profiling

The task of writing tools to abstract runtime data is not trivial [3]. There are many different techniques that address the task of collecting runtime data. As various tools focus on different goals, they each implement a specific technique that suits best for their purpose. Therefore the approach to tool development and the abstraction of dynamic data is not standardized. In the following we explain some of the common techniques that are used for dynamic analysis.

2.1 Profiling Techniques

First we need to distinguish between *sampling* and *instrumentation*. Sampling profiling captures the applications state at specified intervals, such as a defined time interval or a number of CPU cycles [1]. This means that at each interval the profiler captures some state information, e.g. the current execution stack trace. If the intervals are chosen reasonably, the application slows down not too much, and the amount of captured data is managable. On the other hand the captured data does not provide completely accurate information.
This is different with instrumentation profiling, where new code is inserted to catch all events the user is interested in. Using this technique, all details of the runtime behavior are recorded, but the application is slowed down and large amounts of data must be handled.
In the following we concentrate on instrumentation profiling, because we are more interested in detailed information than speed at runtime. Additionaly there are other possibilities to reduce the amount of data by applying filters (e.g. package filters).

2.2 Instrumentation Techniques

There are different instrumentation techniques for Java. A first possibility is to modify the source code directly. This requires that all controlled methods must be parsed and recompiled, and another recompilation is needed to restore the original methods. An example of this is provided by log4j[1].
A second approach is to instrument the virtual machine the application runs in to generate events of interest, like object allocation, method invocations, and so on. This way it isn't necessary to modify the source code. This technique is supported in Java by JVMPI (Java Virtual Machine Proifiling Interface) [6] and the newer JVMTI (Java Virtual Machine Tool Interface), which require native code programming [7].

[1] http://logging.apache.org

Another way to collect behavioral data is to instrument the bytecode of the application directly. With this method, no recompilation of the source code is required. The problem is that this technique relies on a good knowledge of the bytecode instructions used by the virtual machines. But there are libraries that allow one to modify bytecode by writing normal source code which is then compiled and inserted. This is the method that is used by Wiretap.

2.3 Instrumenting Bytecode In Java

The optimal approach (also recommended by the developers of Java) to instrument bytecode in Java is to implement an *agent* and use the `java.lang.instrument` package.

2.3.1 Agents

Agents were introduced in Java 1.5. An agent is a library that is passed to the virtual machine when an application is started. The agent must implement a static method premain(String agentArgs, Instrumentation inst) that is similar to the main() method. It is possible to add more than one agent to an application. They are added by specifing the jar-file and the class that contains the method premain() as command-line options of the virtual machine. After the virtual machine is initialized, each premain() method gets called before the main() method of the actual application. This allows one to take control of the application before it is started.

2.3.2 The java.lang.instrument Package

This package provides services that allows the Java agents to instrument applications by modifying the bytecode of the classes. The mentioned premain() method takes an argument of the type *Instrumentation*, to which can be passed an instance of a class that must implement the *ClassFileTransformer* interface. This interface contains only the transform() method. After an instance of the implementing class is passed to the instrumentation object, each time before a class gets loaded the transform() method gets called. The bytecode of the class is passed, can be instrumented and the modified bytecode is returned at the end of the method and gets loaded. In short, to instrument an application, a transformer-class that implements *ClassFileTransformer* must be written and an instance of this class must be passed to the instrumentation object in the premain() method.

Chapter 3

Implementation

3.1 The Wiretap Profiler Architecture

Wiretap is realized as an agent. A custom bytecode transformer class which implements the *ClassFile-Transformer* interface is written and set in the premain() method of the agent. For more details about how to instrument bytecode in java see Section 2.3.1 (p.9). An overview of the Wiretap architecture is shown in Figure 3.1 (p.10).

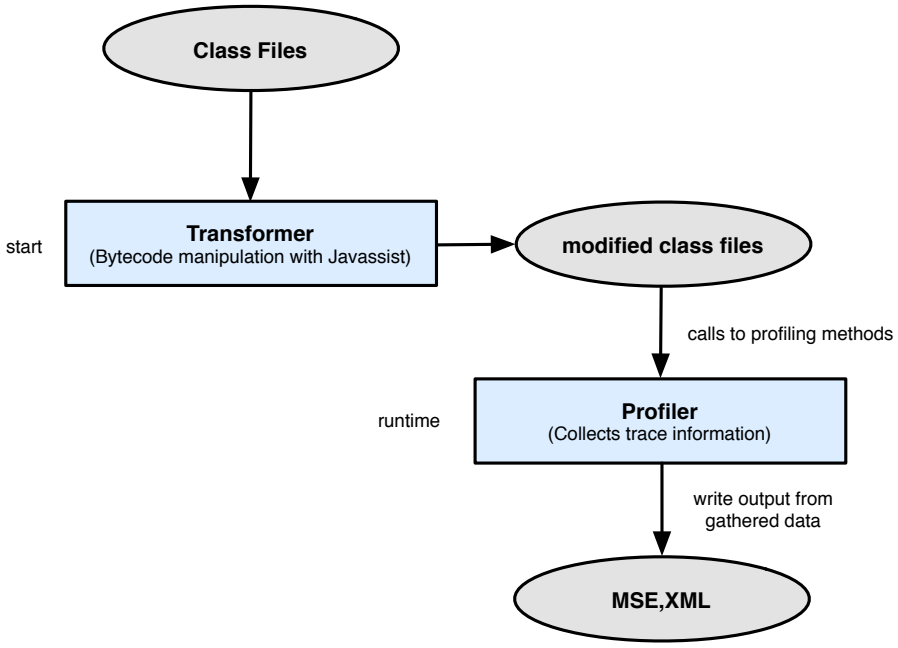

Figure 3.1. Profiling with Wiretap

Figure 3.2 (p.11) is a simplified UML of the design of the profiler application. For more information about the implementation of Wiretap and a more detailed UML, see Appendix B.

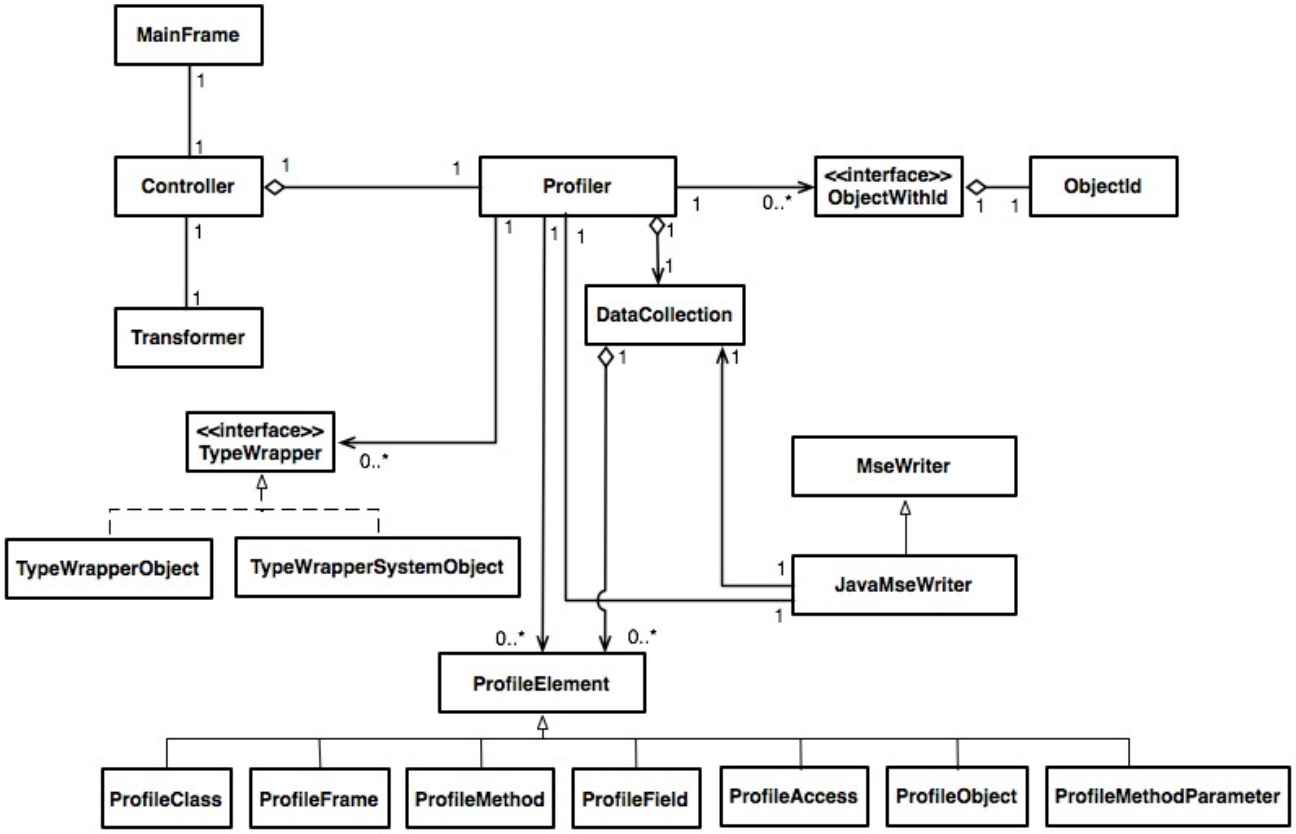

Figure 3.2. The UML Class Diagram of the Wiretap Profiler

3.2 Instrumentation Details

Instrumenting code means modifying the bytecode of a class so that we can monitor runtime events such as field access or message sends. Every time a class of the profiled application is loaded, Wiretap adds instrumentation code to it. For example, at the beginning of each method, a call to Profile.methodStart() is made, which tells our profiler that the application has entered a method. The information that our profiler wants to keep track of is passed as arguments, e.g. which object called the method, what were the arguments and so on. But not only methods are instrumented, also members are added to the class, e.g. a unique identifier (*ObjectID*) that is created when a constructor gets called. This allows us to distinguish between all instances that were created during runtime.
The instrumentation is performed using a library called Javassist [1], which parses class files and provides functions to modify the classes before they are loaded. You can add code on source-level, then Javassist compiles it and returns the modified bytecode of the class. It is also possible to directly add bytecode.

3.3 MSE Format

We provide an export of the collected behavioral data into a file that is MSE compliant. An MSE file can be imported by the reverse engineering environment Moose. It is generic and can specify any

[1] http://www.csg.is.titech.ac.jp/~chiba/javassist/

kind of data, regardless of the metamodel. In our case we export some parts of the static model, e.g. classes, and dynamic data, such as message sends. See Figure 3.3 (p.12) for an overview of the metamodel.

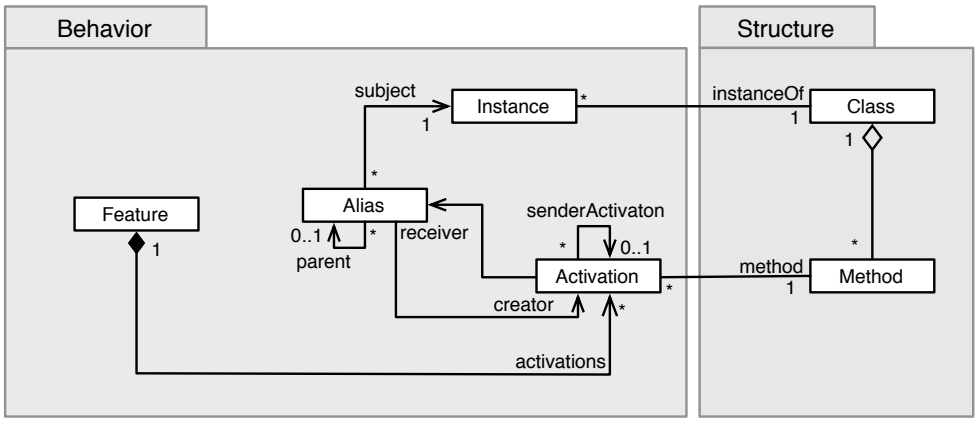

Figure 3.3. Dynamix: The Meta-Model to express Behavioral and Structural Entities of an Application

In an MSE file, entities are saved along with attributes. Each entity has a unique id and several attributes. In Figure 3.4 (p.12) theres a very simple example of an excerpt of an MSE file. In the example, there are the entities FAMIX.Class, FAMIX.Method and DynaMoose.Activation. The class has only one attribute, its name, while the method additionally has a reference to the id of its containing class (which in this case is FooClass). The activation refers to the method that was called and additionally specifies when the method started (timestamp) and when it ended.

```
(Moose.Model
      (entity
            (FAMIX.Class(id: 1)
                  (name 'FooClass')
            )
            (FAMIX.Method(id: 2)
                  (name 'foo')
                  (belongsTo (idref: 1))
            )
            (DynaMoose.Activation(id: 3)
                  (method (idref: 2))
                  (start 31845467368618)
                  (stop 31845484084988)
            )
))
```

Figure 3.4. A small MSE example

3.4 The Trace Feature Relationship

All features are traced in a single execution file so that we preserve the instance information between features. We refer to this as a marked file of feature traces. The MSE Format marks each trace so that

we can distinguish between the activations (i.e. the method calls) of each feature. In Figure 3.5 (p.13) we show a schematic representation of an execution trace.

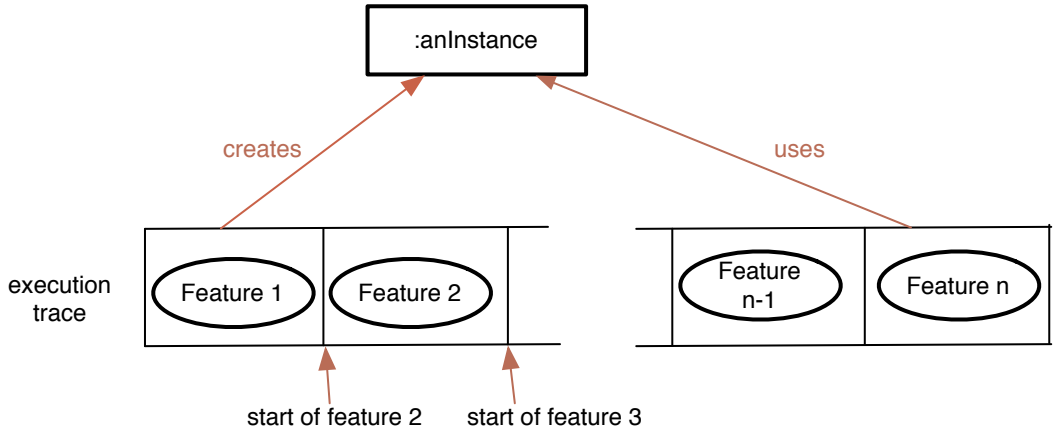

Figure 3.5. The marked execution trace showing how instance information is preserved between features

3.5 Eclipse Plugin

The Eclipse IDE is very easily extensible, because it is made up of a set of plugins. Each plugin encapsulates functional extensions to the platform and is integrated trough extension points of other plugins. Therefore Eclipse doesn't need to be recompiled when new functionality is added. The plugin is described in a manifest file which is written in XML. There the name, version, extensions, extension points and other settings are defined. The extensions must declare which extension points of other plugins they use. An example of an extension point is org.eclipse.ui.actionSet which is part of the plugin org.eclipse.ui. It is used to add GUI-elements in the menu or toolbars. In the manifest the extension of actionSet must declare a label text, an icon and a class which contains the code that is executed when this action occurs.
The Eclipse plugin architecture is easy to understand and very powerful. Hundreds of extension points in the plugins that are already integrated can be used and new ones can be declared.

Our Wiretap plugin actually consists of three eclipse plugins. One contains the agent jar-file (see section 3.1), the second defines a new launch configuration and the third is responsible for the GUI. Of course all parts could be combined into one eclipse plugin, but it is recommended to separate funcionality and GUI, because this enables developers to use the functionality of your program, but perhaps implement another GUI for it. The plugins do not declare any new extension points.
The purpose of the Wiretap plugin is to let the developer start his applications with the profiler directly from eclipse. Normally an eclipse-user starts the program with the *Run* command and creates a launch configuration for it.
Our plugin extends the point org.eclipse.debug.core.launchConfigurationTypes which lets us define our own type of launch configuration.
The UI-Plugin extends the point org.eclipse.debug.ui.launchConfigurationTabGroups, which is used to create a new tab group where the user can modify the settings for the launch of the application. In this

new tab group the common tabs are available, like the ones where the user can set the project and the main-class, but also a new tab for specific Wiretap settings. Our new type of launch configuration then starts the application as usual, but with the profiler running along.

3.6 Problems Instrumenting Java Code

Instrumenting java code is not unproblematic as there are many special cases that must be considered when inserting code. This is especially the case when modifying bytecode directly. Javassist is able to handle most of the code, but in some rare cases it fails. There are two kinds of failures. Either (1) Javassist fails to compile the code, or (2) it can be compiled but throws a VerifyError at runtime. In the first case you can catch an exception and inform the user that a method could not be instrumented. This method will then of course not show up in the resulting execution trace. But as this happens very rarely, this is a drawback we can live with.

The second failure is more problematic as it throws a `VerifyError` at runtime, which leads to a crash of the application. A `VerifyError` is thrown when the virtual machine detects some corrupted code which occurs when the compiler is not working totally (as it seems to be the case with the compiler implemented by javassist). To tackle this issue, we added an option in the Wiretap launch configuration tab to turn off the verifier of the virtual machine. But this is not an ideal solution, because with the verifier turned off you could feed the virtual machine with whatever bytecode you want, possibly performing illegal actions.

Chapter 4

Example Usage

4.1 Introduction

In this chapter we perform a case study to show how to extract features with Wiretap and perform analysis with DynaMoose [5]. The example application is Phone Simulator, which lets the user start a cellphone and do some basic actions a real phone can perform, like calling, managing contacts, setting a logo and so on. We associate features with these actions. First we trace a lot of features, and then we will have a closer look at only some of those features. In figure 4.1 you see a picture of the cellphone at

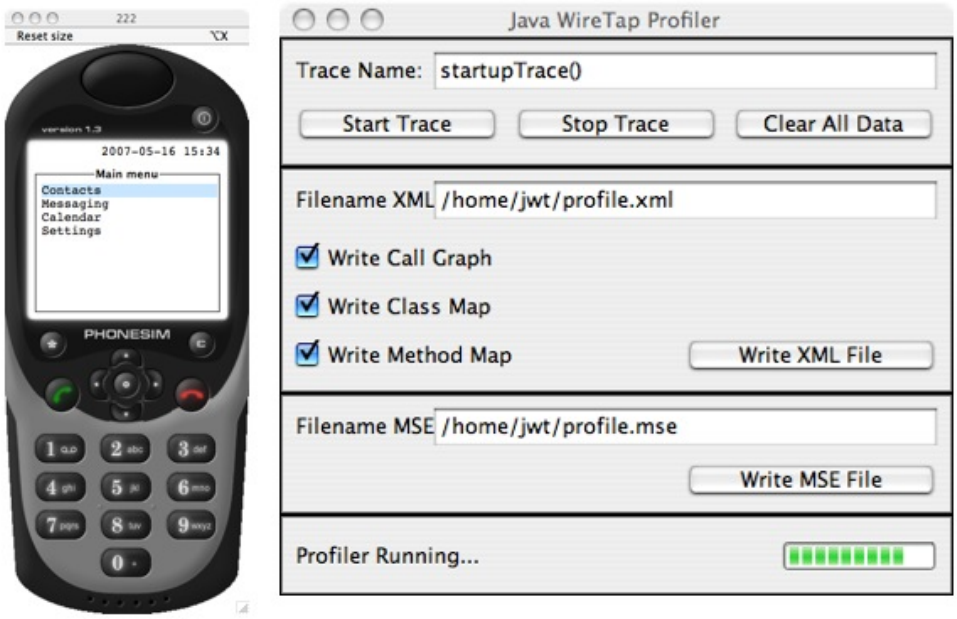

Figure 4.1. The Phone Simulator GUI and Wiretap Controller

the left, and at the right the Wiretap control interface that is used to start and stop traces to extract the desired features.

4.2 Profile Some Features

For the case study we traced the following features: *startupTrace, switchOnPhone, dialNumber, hangUp, menu, contacts, newContact, infoContact, settings, settingsRingTones, calendar, calendarEvent, messenging and inbox.*
Most of the names are self-explanatory. *startupTrace* is the feature that represents the initalization of the program and the phone, *switchOnPhone* is the action of switching the phone on, *menu* is a feature that represents clicking the OK button to enter the phone menu, and so on.

4.3 DynaMoose Analysis

Figure 4.2 (p.16) shows the overview of the traces in Moose. All traces that were created during profiling are listed and indicate how many classes were involved in a feature (NOCf) and how many calls were made (NOEvents). E.g. *startupTrace* is the trace which has the highest amount of involved classes, which makes sense because often at initialization a lot of classes are already used. On the other hand, a small feature like *hangUp* has only a few classes involved.

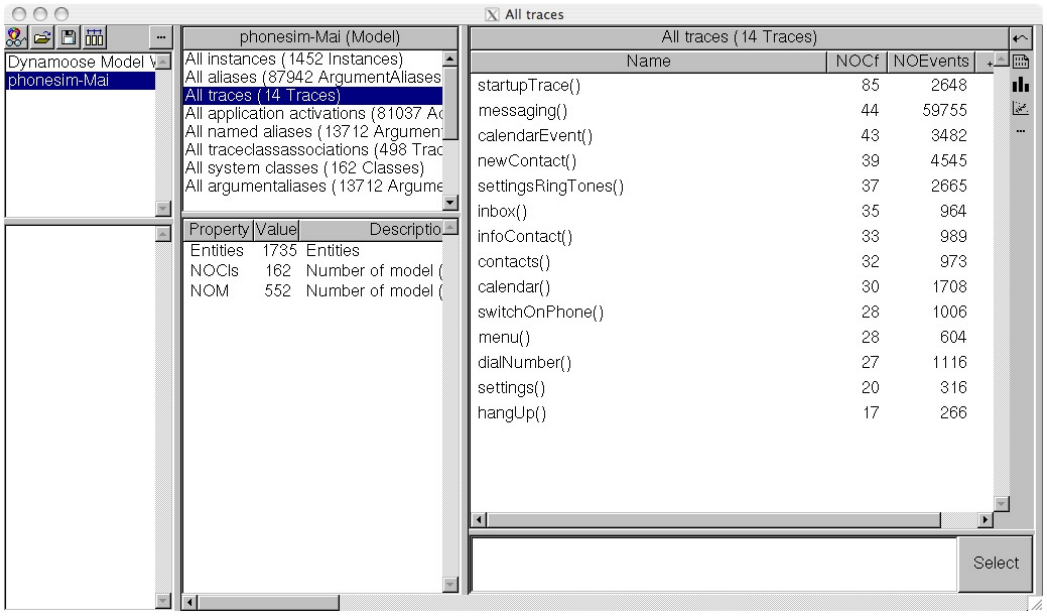

Figure 4.2. The DynaMoose Analysis of 14 Features of Phone Simulator

Moose provides a lot of tools to create different views of the behavioral data. In the following we will show and explain some of these views.

4.3.1 Feature Views

DynaMoose enables us to visualize the features of our model as feature views [5]. The feature affinity measurement quantifies the relevancy of a source entitiy (package, class, method) to a feature. In Figure 4.3 (p.17) we show feature views as groupings of participating methods. Each large rectangle represents a feature captured using Wiretap and each small rectangle of the feature view represents a

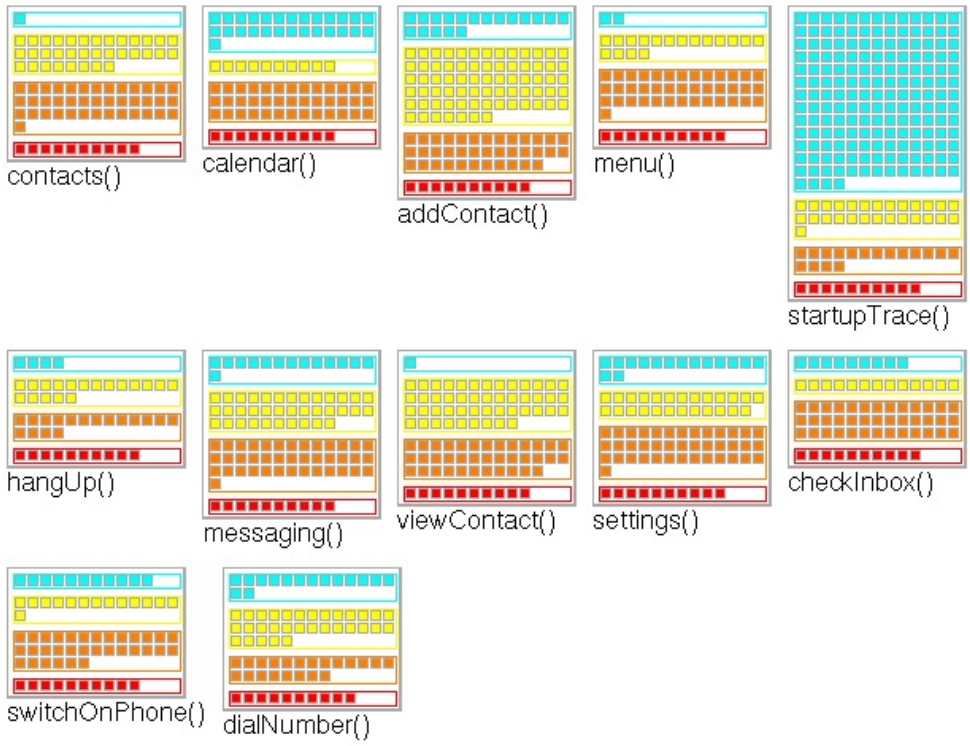

Figure 4.3. The Phone Simulator Features Views showing Feature Affinity Values of Methods

participating method. The colors of the method rectangles indicate the level of participation in a set of features. Red implies that a method is *hot* (referred to as an *infrastructural feature* method) as it is used in all the features and at the other extreme, cyan implies that a method is *cold* (referred to as a *single feature* method) as it is participating in only one of the features of the model. Feature affinity also differentiates between methods that participate in most of the features (*high group* methods shown in orange participate in more than half of the features) and methods that participate in some (*low group* feature methods shown in yellow participate in half or less of the features).

In Figure 4.3 (p.17) we see that the feature *startupTrace* contains a high number of *single feature* methods. This is because this feature is traced when the Phone Simulator application is launched. It is responsible for initialization code that is executed only once.

4.3.2 Feature Relationships

DynaMoose enables us to view relationships between features based on shared source entities. In Figure 4.4 (p.18) we visualize the relationships between the features based on the degree of similarity between the low group feature classes. The visualization uses grayscale to represent the relationships (black means that two features are completely related, white means that there is no relationship). We highlight the relationship between the *Contact* and *infoContact* features. Our analysis reveals that these features are tightly related as they share a high proportion of classes. This result makes sense as both features are accessing the classes that implement the contact and address book functionality of the Phone Simulator application.

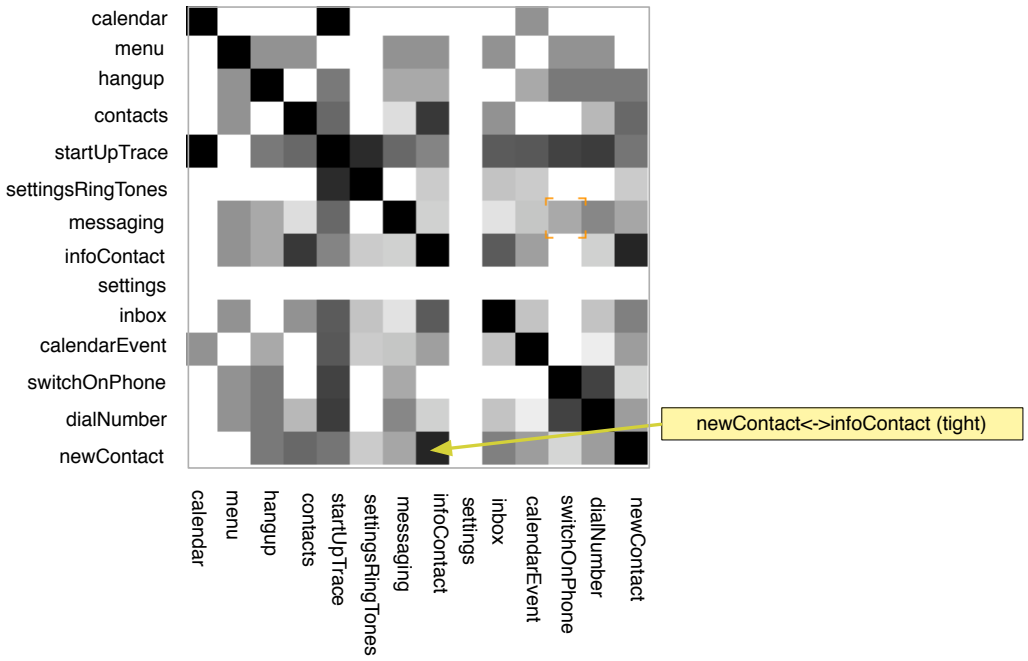

Figure 4.4. The Static Feature Relationship View of 14 Features of Phone Simulator

4.3.3 Instance relationships

Wiretap captures the instance relationships between individual features. We illustrated this schematically in Figure 3.5 (p.13) by showing how instances created in one feature may also be used by other features. This establishes what is described as a *dynamic relationship dependency* between features [5]. We show an example of this relationship from our case study application Phone Simulator in Figure 4.5 (p.18).

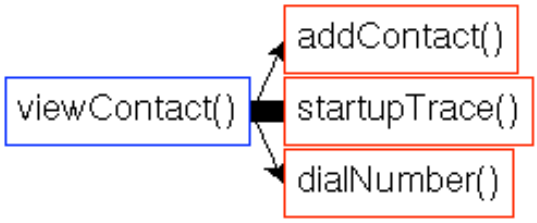

Figure 4.5. The Instance Dependency Relationship View of the *viewContact* Feature

This shows a feature as a node and the edges between the nodes represent the dynamic dependency relationships between the features. The visualization maps the *number of instances referenced* measurement to the width of the edge to reflect the strength of the dependency. We see that the feature *viewContact* depends on a large number of instances (185), that have been created during the *startupTrace* feature.

4.3.4 Other Analysis Representations

The main challenge of dynamic analysis is the huge volume of data, making it difficult to extract high level views. Kuhn and Greevy introduced a novel approach representing entire traces as signals in time [8]. This approach allows the reverse engineer to visualize feature traces as time plots and to annotate the signals with color to represent measurements such as the *feature affinity* value of the methods represented in the signal trace.

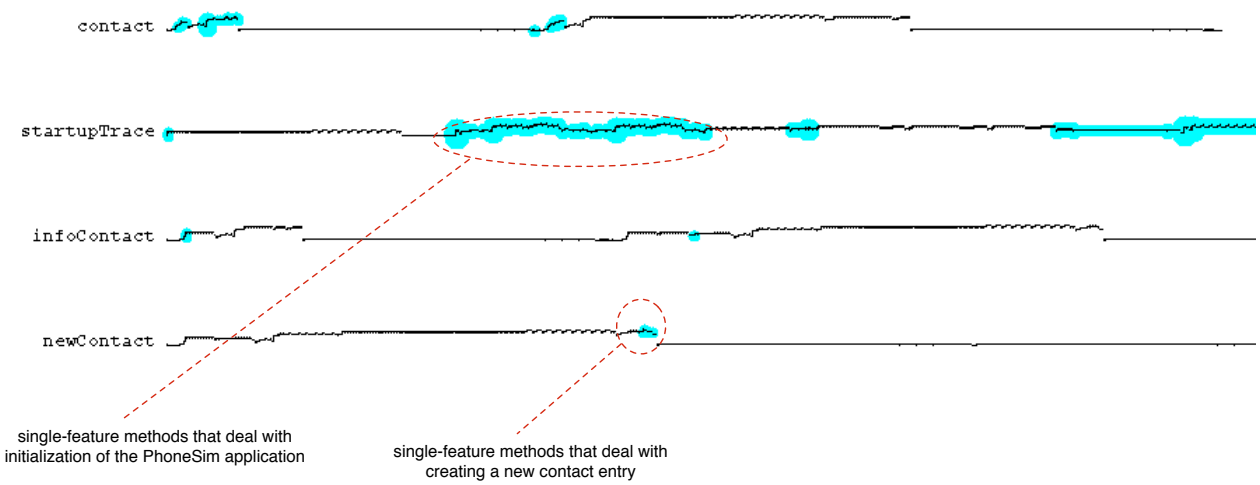

Figure 4.6. The DynaMoose Feature Signal Analysis of 4 Features of Phone Simulator

Figure 4.6 (p.19) shows signal representations of four of the features of our Phone Simulator application. We annotated the trace to highlight the *single-feature* methods. We show which parts of the features represent feature specific activity.

4.4 Conclusion

Looking at the visualisations in this case study, we can see that the results we obtained from feature analysis make sense for our Phone Simulator application. Features we expected to be similar really are, as the contact example showed. Based on these results, we can see that Wiretap gathers the behavioral data most of the analysis tools need and we verified that the MSE output is written correctly. In the future, Wiretap could be extended to extract even more runtime data as there are other tools in Moose that need more information to perform their analysis.

Chapter 5

Conclusion

In the first chapter, we outlined the goals of this work. Then we showed how the model of behavioral data we extract can be used to perform a variety of feature analyses. We have extracted the features of a case study application Phone Simulator, and analyzed the resulting Feature Model of the system using *DynaMoose*. Thus we have reached the goals that we stated at the outset. Our profiler, Wiretap, gathers behavioral data, including the details of instance tracking. Due to the interactive GUI, it provides a way to trace features by allowing the reverse engineer to mark the start and end of a feature. The resulting behavioral data is written in a well-defined format, namely MSE which is understood by our reverse engineering environment Moose. Our Wiretap profiler is implemented as an Eclipse plugin. This means that it is neatly integrated into the developer's familiar working environment. The user can easily start the profiler directly from eclipse.

5.1 Lessons learned

One of the key lessons learned during this project was that instrumenting Java bytecode can be very tricky. It is very easy to cause applications to crash if you are not careful in modifying the code. There are many special cases to be considered. Several times the profiler worked perfectly on small projects with only a few classes. However, when Wiretap was tested with a larger application, it crashed because of such special cases. For example, in an early version, we didn't check if a method we wanted to instrument was abstract. In the test project there were no abstract methods, so it worked. But in a larger project there were abstract methods, and as it is not possible to instrument abstract methods, the application threw an error.
Another lesson was that it is difficult to write Eclipse plugins that will work over time, because when a new version of Eclipse is released, it is not guaranteed that the plugin will still work, e.g. if class names are changed in the eclipse core plugins.

5.2 Future Work

In this section we outline future work and extensions to the functionality of Wiretap.

- *Profiling System Classes.* Currently the profiler instruments only classes of the application being profiled; system classes are not considered. In most cases this is not a problem, because when considering the runtime behavior of a system you are mainly interested in the classes that participate

in the behavior. (E.g. most of the time you are not interested in the call tree of the system classes). In the current implementation of the profiler, it is not possible to track instances of the class *String* for example. So one possible extension would be to also track instances of system classes.

- *Shortcomings of the current Javassist library.* The few instrumenting problems of Wiretap could be handled by awaiting a newer version of Javassist or using an alternative library. Another approach would be to write the bytecode directly (either with Javassist or another library). This would add more flexibility to the profiler, but it would become more complex too, because the developer has to have excellent knowledge of bytecode.

- *Speed and Space Management.* The performance of Wiretap is not optimized, as our emphasis in this work was to have a clean design to keep the model flexible rather than to speed things up. For most Java applications this works fine, but if e.g. someone wanted to profile a real-time application, the current implementation needs to be enhanced. And as space is concerned, it would be better to have some other ways to reduce the amount of dynamic data. With the current implementation, we have only implemented a mechanism to selectively filter packages from the instrumentation step.

- *User Interface.* Another issue is the user interface, which could be heavily improved. In particular, the selection of the packages is quite basic, e.g. inclusive/exclusive filters for packages would be a nice improvement.

- *Standalone Wiretap.* There should be a standalone version of Wiretap, because as we have seen if a new Eclipse version is released, the plugin might not work anymore due to incompatibility problems. A standalone version would require an extension of the GUI.

Appendix A

User Guide

A.1 Installing the plugin

The releases of the Wiretap plugin can be found in our subversion repository:

`https://www.iam.unibe.ch/scg/svn_repos/fierz/profiler/releases`

To install the Wiretap plugin, just download the latest zipfile and copy the zipfile into the root folder of eclipse and extract it. It should unzip three folders into the plugin folder of eclipse. If you want to reinstall Wiretap, e.g. to update it, it is recommended to delete those folders from the plugin folder first. The Wiretap folders are named as follows:

- ch.unibe.iam.scg.profiler_<version number>
- ch.unibe.iam.scg.profiler.launcher_<version number>
- ch.unibe.iam.scg.profiler.launcher.ui_<version number>

You have to restart Eclipse so the plugin will be loaded.

A.2 Running from Eclipse 3.2

You can run Wiretap much like any other run configuration (e.g. debug configuration) directly from Eclipse by right-clicking on the top node of the project and selecting *Run As* and then *Run...* from the context menu.

An Eclipse window (as shown in Figure A.1 (p.23)) is displayed with a list of the different launch configuration types. Right click on *JWT Profiler* and select *New* from the context menu. This results in a new Wiretap configuration being created with your selected project already set. On the first tab of the configuration you have to specify the main class. To start the application with the profiler, click *Run*. If you want to configure the profiler, you can move to the tab named *JWT* before you run the application.

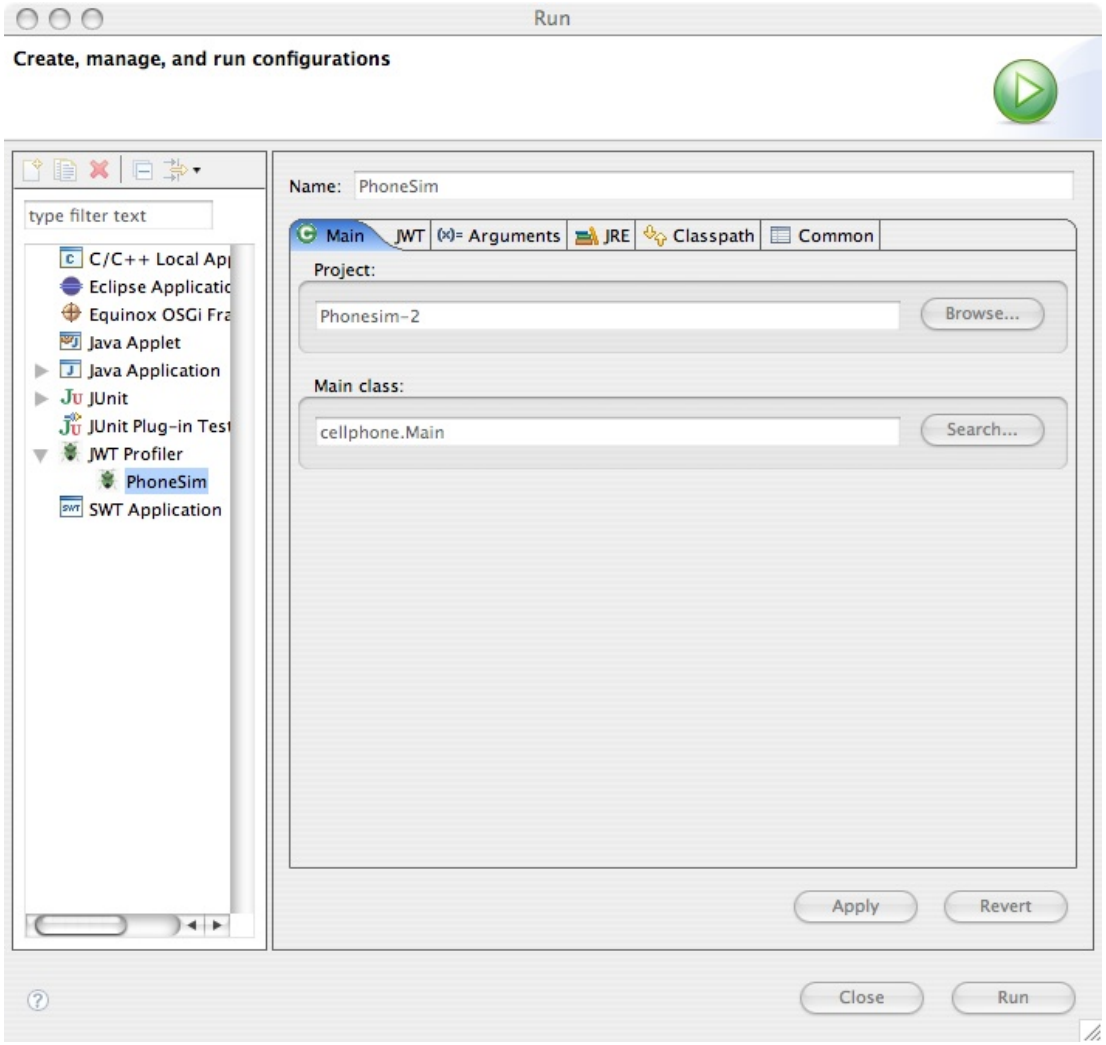

Figure A.1. The Eclipse Run Configurations Window showing the Wiretap Configuration extension

A.2.1 Profiler Settings

Here's a description of the options you can set. The descriptions correspond to the numbers shown in Figure A.2 (p.24):

1. *Profiler running from beginning:* If set, the profiler will be running from the moment the application is started.

2. *Instrument field access:* If set, field access is instrumented and will show up in the resulting execution trace. As this generates another large amount of data that is not always needed, you have the possibility to turn it off.

3. *Turn off bytecode verifier*: If set, the bytecode verifier of the virtual machine is turned off. By default this option is not set, because it should only be used if the profiler has problems instrumenting some code. If you get an unexpected java.lang.VerifyError when the profiler is running, you should turn this option on. For more information on this issue see 3.6.

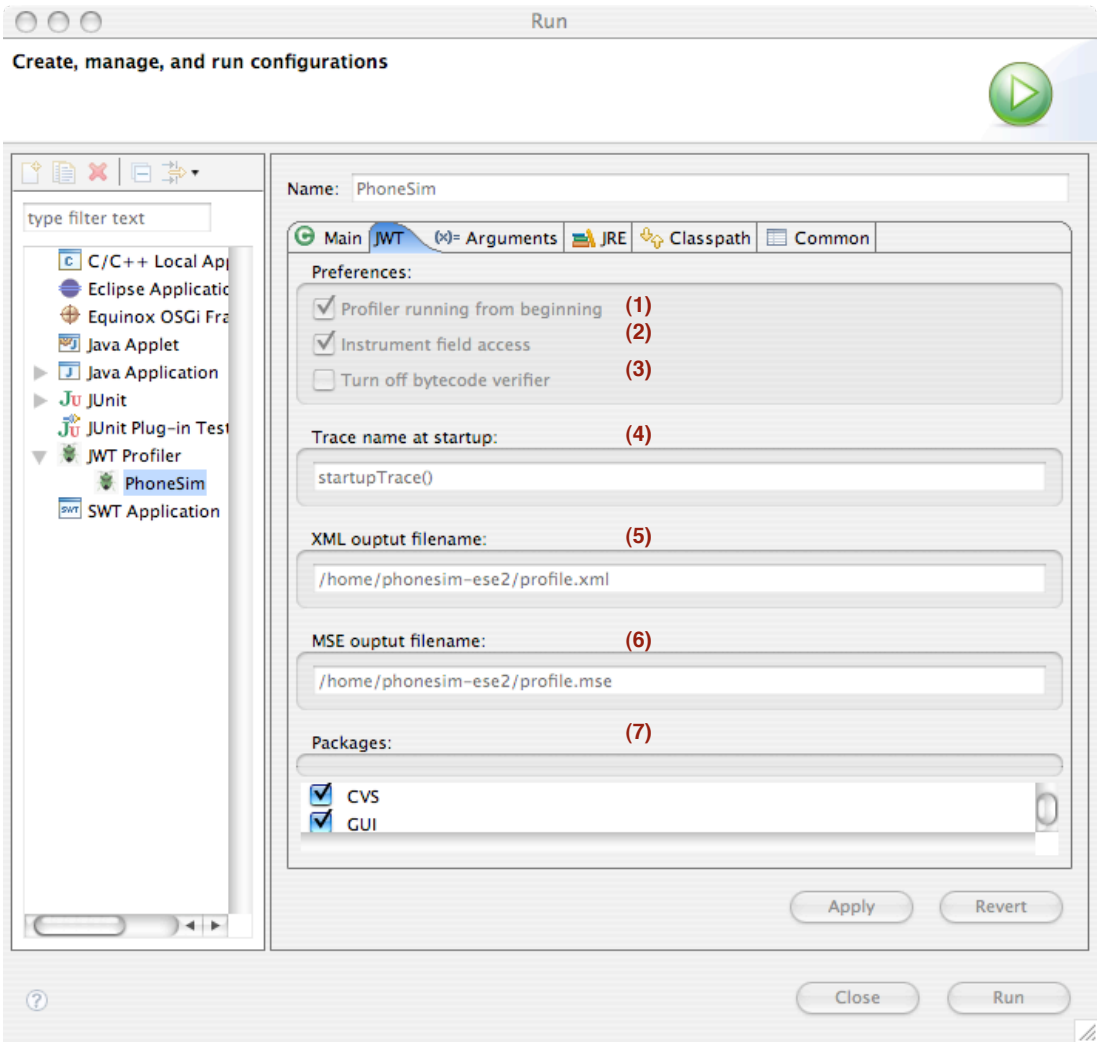

Figure A.2. The Wiretap Configuration Options

4. *Trace name at startup:* If the profiler is running from the beginning (see option above), this will be the name of the first trace.
5. *XML output filename:* The name of the XML output file.
6. *MSE output filename:* The name of the MSE output file.
7. *Packages:* This is a list of the packages the project contains. The selected packages will be instrumented. By default all packages are selected, but sometimes you only want to profile a set of packages to reduce the produced amount of data. So if there are packages that are not of interest, deselect them.

A.2.2 Runtime

After the application is started, the Wiretap profiler window appears on the screen as shown in Figure A.3 (p.25). There are four sections in this window.

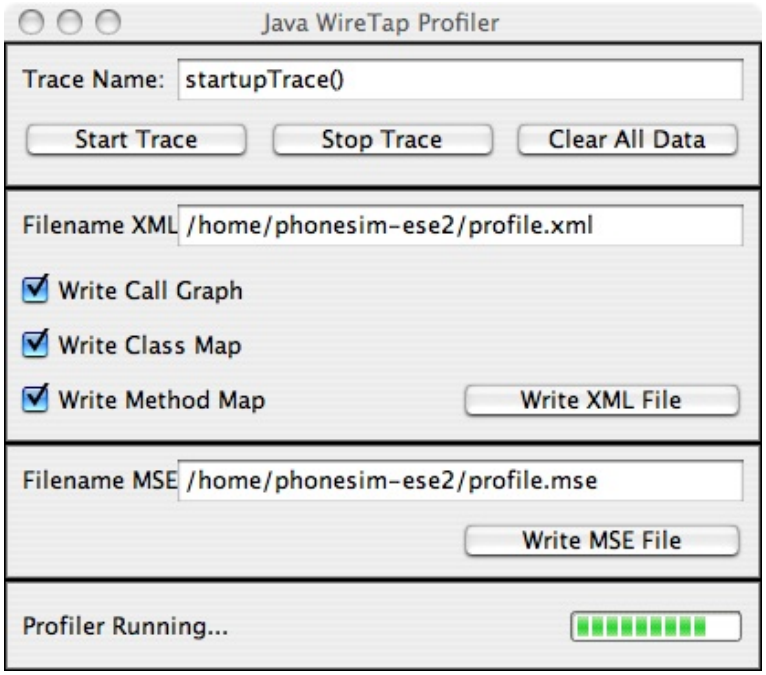

Figure A.3. The Wiretap profiler window

Control Panel. The top section lets you control the profiler. If the profiler is not running, you can start it. That means you will start a tracing a feature with the specified name to uniquely identify the feature. Once the feature execution is completed, you press the *Stop Trace* button to mark the end of the execution of a feature. If you press *Clear all data*, all the execution data you have profiled up until this point in time will be thrown away. Be careful: if you do this, we recommend you first save the execution data to a file beforehand.

XML Output. The section below the control panel is the panel for specifying an XML output file. You can specify the filename and there are three options which let you control what execution data should be written to the file. Press the *Write XML File* button to dump the gathered execution data to the specified XML file in XML format.

MSE Output. This is the panel for specifying the MSE output file. Press the *Write MSE File* button to dump the gathered execution data into the specified MSE file.

Profiler Status. In the section at the bottom indicates the status of the profiler. If the status bar is shown in white, this indicates that profiler is currently not collecting any execution data from the running application. If, on the other hand, it is animated by is displaying a series of flashing green lines, this indicates that the profiler is active and is collecting execution data.

A.3 Running as Standalone

Wiretap was intentionally designed to be run from within Eclipse. We did not invest much effort in the standalone version. With the current version, starting Wiretap from the command line is currently not particularly user-friendly, as you have to specify all the information the Wiretap profiler needs at startup as arguments to the virtual machine. For example you would have to pass all names of the packages you

want to profile. There is currently no standalone GUI to manage this.

Below we show an example of how to start a Java application with our Wiretap from the command line:

```
java -javaagent:"/home/jwt/lib/profile.jar"
       -Drunonstart="true"
       -Dstartuptrace="startupTrace()"
       -Dtrackfieldaccess="false"
       -Doutputxml="/home/jwt/project/profile.xml"
       -Doutputmse="/home/jwt/project/profile.mse"
       -Dpackages="package;package.subpackage" /home/jwt/project/Main
```

This command runs an application with the Mainclass *Main*, with the Wiretap profiler actively colllecting execution data from the start of execution, no field access is tracked, the MSE-output file is called profile.mse and so on. Furthermore this example only traces methods invoked from two packages, *package* and *package.subpackage*. Typically you want to profile a lot more packages, so you have to list all of them on the command line.

Appendix B

Programmers Manual

B.1 Setup Project with Eclipse

The following actions were performed with Eclipse 3.2.2.
This manual requires that you have already downloaded the source code of the projects that are needed to compile the profiler. The sources are on our subversion repository:

```
https://www.iam.unibe.ch/scg/svn_repos/fierz/profiler
```

There are four projects (The agent project and three plugin projects):

- *jwt*, the main profiler project (the agent)
- *plugin_jwt*, the plugin project that contains the agent jar created in the first project
- *plugin_launcher*, the plugin project that creates a launch configuration for Wiretap
- *plugin_ui*, the plugin project that contains the UI for the Wiretap launch configuration

The first step is to import the Wiretap projects into Eclipse. We'll begin with the plugin projects. Omit these steps if you only want to use the Java agent without the Eclipse plugin. The steps are the same for all three projects, so we only describe how to import one here, namely the launcher-plugin.

- In Eclipse select *File -> New -> Project*. From the wizard list, select *Java Project* and then press the `"Next >"` button (Although this is a plugin, don't select Plugin Project!).
- Type in a project name (for this case you could name it *plugin_launcher*). In the content section (see Figure B.1 (p.28)), select *Create project from existing source*, and choose the directory of the launcher plugin and then click `"Next >"` button.
- The next step in the wizard is java settings. In the tab *Source*, make sure that in the field *Default output folder*, the folder is *$yourpluginname$/bin*, where *$yourpluginname$* is the name of the project you chose at the first step. Then click `"Next >"` button
- Move to the tab *Libraries*. Because this is a plugin, you have to add the plugin dependencies, so click *Add Libary....* In the *Add Library* window (as shown in Figure B.2 (p.29)), select *Plug-in Dependencies* and click *Finish*.

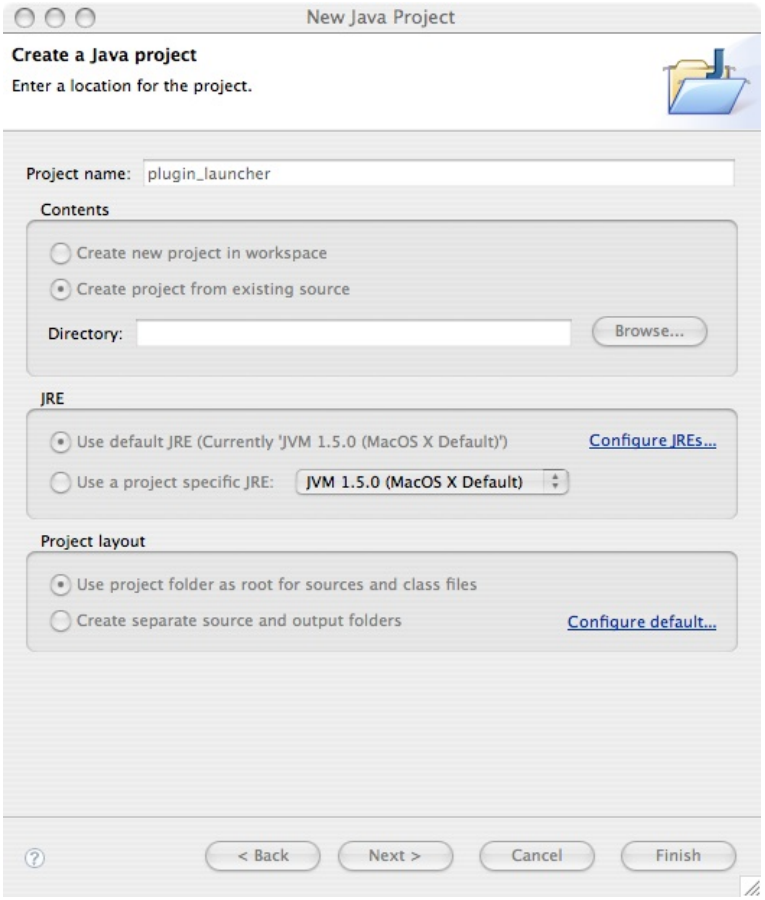

Figure B.1. The Eclipse Project Wizard

- Now click *Finish* in the java settings window too, the project is created.
- You may get errors, because you haven't added all plugins yet. After all three plugin projects are imported, you shouldn't get any more errors. If you still get errors, clean all projects (*Project -> Clean...*).

Now we will import the *jwt* project. You can do this almost like you did with the plugins. The only difference is that you don't have to add the plugin dependencies. After the import the *jwt* project is available in Eclipse.

A Java agent is passed as an argument to the virtual machine. So you have to create a jar file each time you change something. There is an ant-buildfile which does this for you. It creates the jar and copies it directly into the plugin-folder where it will be found when you start the plugin. But before you run the ant script the first time, you have perform the following settings.

- Right-click on the file build.xml in the *jwt* project. Select *Run As -> Ant Build...* The ant window appears as shown in Figure B.3 (p.30).
- In the tab *Main*, you have to specify the base directory, which is simply the directory of your profiler project.

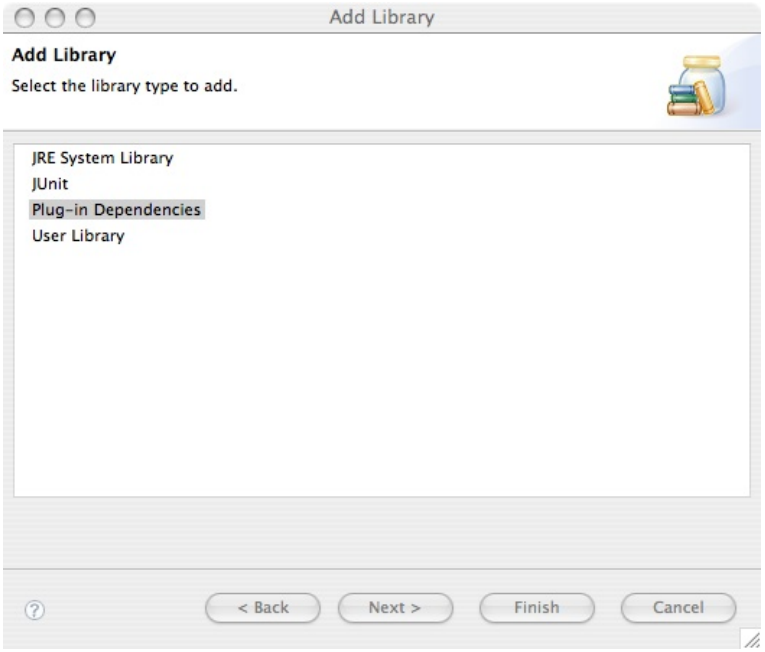

Figure B.2. The Eclipse Project Dependencies

- Move one to the tab *Targets*, and make sure that *dist* is selected.
- Move on to the tab *Properties*. Deselect the checkbox on the top and click *Add Property*.
- Type in *pluginpath* in the first field. In the second field you have to specify the path of the library-folder of the main plugin project. For example, if you have installed the *plugin_jwt* project in */home/jwt/plugin_jwt/*, then you would have to enter */home/jwt/plugin_jwt/lib/*.

Now you can start the build. Each time you make changes to the profiler you have to execute this script, so the plugin uses the jar file that is up to date.

To test the plugin, doubleclick on the file *plugin.xml* in one of the plugin projects. The plugin overview window is opened. Click *Launch an Eclipse Application*. This will start eclipse with your plugins in another workspace, now you can test them. To export the plugin, use the *Export Wizard* that you find also in the plugin overview window.

B.2 Model

Here we explain the important classes in the model. See the UML diagram Figure B.4 (p.32) for further information on how the classes are related.

Controller is the central point of the program. The function premain(), which is the entry function of the program, is located there. In premain() everything is initialized and the MainFrame is created and shown before the actual application is started.

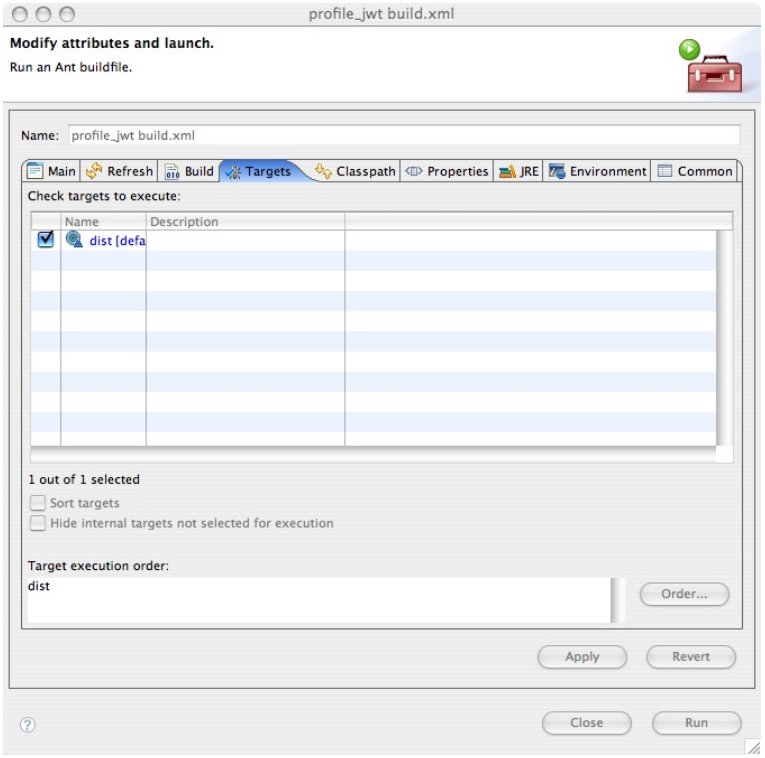

Figure B.3. Building the Profiler Projects with Ant

The class *Profiler* is responsible for the actual profiling. The start(), stop() and clear() methods re used to control the profiler. The most important methods are objectAllocation(), methodStart(), methodEnd(), fieldAccess(). They get called by the instrumented code when one of the corresponding event occurs, e.g. when a new object is instantiated, objectAllocation() gets called. Each one of those methods gets several parameters which hold the information about the event. In the case of objectAllocation(), the instance of the object that was created is passed. The profiling methods then process the data that is passed from the instrumented code. E.g. every class, every method, and so on is saved in a collection. Additionally, there are some helper methods that are mostly used by the instrumented code.

The *DataCollection* class contains all collections. It acts as the holder of the data the profiler collects. But what exactly is saved in the collections? In the case of the method collection, we could simply have saved the full method name (i.e. the name of the package, the name of the class and the name of the method). Another possibility was to save the reflection-object of the method. So as there are more than one ways to do it, we decided to keep it flexible, that means we came up with an own class for each *component* that is profiled, so if someone wants to save the data in another way than we did, he can simply rewrite the implementation of those classes. The classes are *ProfileClass, ProfileField, ProfileMethod, ProfileMethodParameter, ProfileFieldAccess, ProfileFrame* and *ProfileObject*. They all derive from the abstract class *ProfileComponent*. For all of those classes it should be clear what they stand for. Only the class *ProfileFrame* maybe is not that obvious and is the most complex of those classes. It represents a method call and holds lot of information about it, for example the start and end time, the arguments passed and so on.

Transformer is the class that is responsible for the instrumentation. When a class is loaded, the method transform() gets called, and the bytecode of the loaded class is passed. The instrumentation is done with javassist and the modified bytecode is returned. The following instrumentations are done:

- Each class gets a new field of the type *ObjectID* which is used to distinguish between all instances that occur during runtime. Internally, the *ObjectID* is simply using a counter to create a new id. Further the interface *IObjectWithID* and its only method ch_iam_unibe_scg_wiretap_getID() are added to the class. This method simply returns the *ObjectID*.
- At the beginning and at the end of each method, code is added which calls the methods of the class *Profiler* and passes the required data. Constructors additionally add code that creates the *ObjectID* that we mentioned above.
- Before each write field access, code is added that calls methods of the class *Profiler*.

See the javassist manual for details on how to add code to a class.

The *ITypeWrapper* interface along with the classes *TypeWrapperObject* and *TypeWrapperSystemObject* that implement it, is used to solve the problems that Java distinguishes between referenced and primitive types, and that there are uninstrumented classes (e.g. all classes of the system library are not instrumented). Hava a look at this method with two parameters: doIt(MyObject obj, long timestamp). Our profiler keeps track of the instances, and so if the method doIt() gets called, it creates a *TypeWrapperObject* with the *ObjectID* of the parameter obj. But for the second parameter, there is no instance because it is a primitive type. To facilitate things, we decided to create a *TypeWrapperSystemObject* which simply holds the name of the type, as no instance can be found. It is not called TypeWrapperPrimitiveType, because it includes classes that are not instrumented. E.g. if you pass a String (which is not instrumented), a *TypeWrapperSystemObject* is created and the name java.lang.String is saved in it. In short, all references to variables are of the type *ITypeWrapper*.

MseWriter and *JavaMseWriter* are used to create the MSE-output file. *MseWriter* provides methods to write a well formed MSE file in general. The important code is in *JavaMseWriter*. There is a counter which is used to create a unique identifier for each element that is written in the output file. Each *ProfileComponent* gets an identifier, and they are saved in Maps to keep track of those identifiers. Note: In the output, a method call is referred to as an activation, while in the program during profiling, a method call is called frame.

MainFrame contains all GUI components. It uses the class *Controller* to interact with the model. Most of the code should be self-explaining.

B.3 UML Diagram

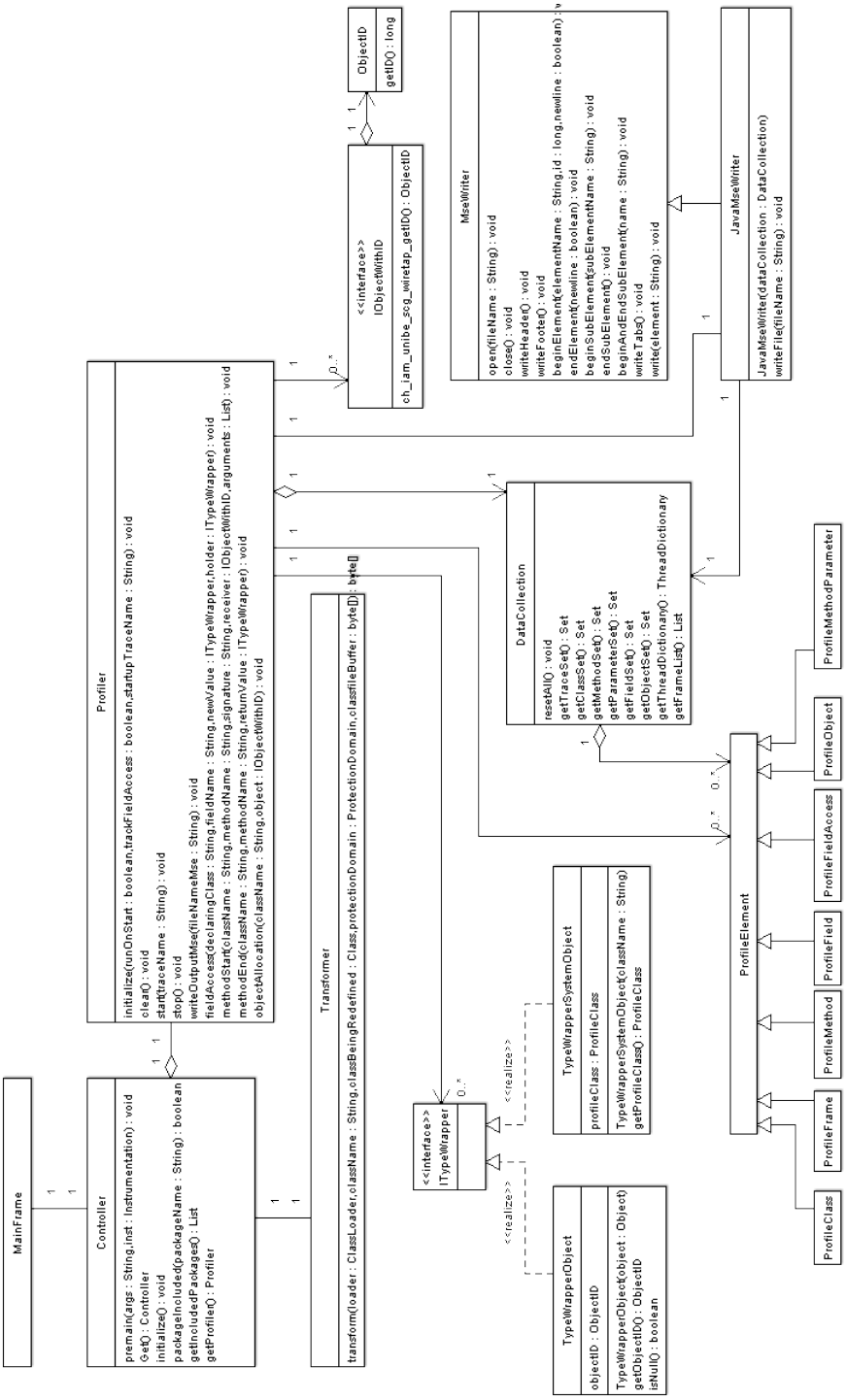

Figure B.4. Detailed UML Class Diagram of Wiretap

Bibliography

[1] W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Proceedings of The Third Asian Symposium on Programming Languages and Systems (APLAS-2005)*, volume 3780 of *LNCS*, pages 178–194, Tsukuba, Japan, nov 2005.

[2] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. In R. S. Arnold, editor, *Software Reengineering*, pages 54–58. IEEE Computer Society Press, 1992.

[3] M. Denker, O. Greevy, and M. Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, Mar. 2003.

[5] O. Greevy. *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, University of Berne, May 2007.

[6] Sun Microsystems, inc. jvm profiler interface (jvmpi).

[7] Sun Microsystems, inc. jvm tool interface (jvmti).

[8] A. Kuhn and O. Greevy. Exploiting the analogy between traces and signal processing. In *Proceedings IEEE International Conference on Software Maintainance (ICSM 2006)*, Los Alamitos CA, Sept. 2006. IEEE Computer Society Press.

[9] A. Lienhard, S. Ducasse, T. Gîrba, and O. Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[10] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC 2007)*, 2007. to appear.

[11] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.