Analysing Java System Properties

Bachelor's thesis
at the
Software Composition Group (SCG),
Institute of Computer Science and Applied Mathematics,
University of Bern, Switzerland

By
David Wettstein
November 2013

Led by
Prof. Dr. Oscar Nierstrasz
Andrei Chiş

Abstract

In Java, System Properties are global variables in disguise: every class can access them and modify their values. Thus, they suffer from the same pitfalls as global variables: it is difficult to reason about their usage, they can introduce hidden dependencies between classes, they can cause naming conflicts, *etc*.

An analysis of several open-source Java systems revealed that System Properties are a common presence. Given the sheer size and complexity of these systems, manually reasoning about System Properties is no longer an option; tool support is required.

To address these problems in this thesis we introduce the *Property Investigator*, a tool for understanding the usage of System Properties. The tool allows developers to identify all System Properties used within an application, inspect their values and determined from where these values were set. To make it accessible to developers we have integrated it into the Eclipse IDE.

Acknowledgements

I would like to thank the people in the SCG group for providing me with feedbacks and funny moments during the seminars.

My thanks go especially to Andrei Chiş for his support and inputs during the implementation of the project.

Contents

1 Introduction

1.1 Global variables

A *global variable* is a variable with global scope. This means it can be accessed from anywhere in an application and its value can be changed at any time. They are often used to communicate between parts of a system that do not share a direct association.

Global variables can cause a wide range of problems, thus, their usage is considered a bad practice. For example, since they can be freely accessed from anywhere within an application they have a negative effect on code reuse and software testing. Furthermore, it can be difficult to understand the reason behind a global variable if its usage is scattered across the entire code base. Last but not least, global variables can cause naming conflicts when integrating modules or concurrency issues when used in multithreaded applications.

1.2 Java System Properties

In Java, *System Properties* are used to describe the current environment (*e.g.* the current user, the Java version, the operating system). They are part of the standard library and can be freely accessed through public methods. Developers are allowed to define and use their own custom System Properties or change the values of the default ones. Therefore, Java System Properties behave like global variables and inherit all their negative effects.

1.3 Challenges

Given the sheer size and complexity of today’s software systems, manually reasoning about the usage of System Properties does not scale. Tool support is needed to automate and ease this activity.

Such tools must be able to locate all System Properties used within an application. This is not a straightforward requirement as System Properties are identified using `String` objects whose values might only be known at runtime. For example, the identifier of a System Property can be computed by concatenating the results of several method calls. Therefore, this information can only be obtained by running the application.

Just knowing what System Properties are present within an application is not enough. For each one developers need to identify the code locations that reference it. Then, for each code location where a System Property is accessed, they need to know what the value of the property is. To make matters worse, during the execution of the program the same call to a System Property can return different values.

Last but not least, understanding how a System Property got to have a certain value is an important aspect. To support this, tools should allow developers to determine the previous code location that changed the value of a System Property.

1.4 A tool for reasoning about System Properties

We addressed the challenges mentioned above by implementing a dedicated tool, the *Properties Investigator*. To cover all the use cases we selected an approach based on dynamic analysis. The tool that we propose has two components. The first component, the *Analyser*, is responsible for extracting the required information. The second component, the *Visualizer*, is responsible for displaying this information to the user. These two components form an Eclipse plugin.

1.5 Outline

The remaining of this thesis is structured as follows:

- **Chapter 2** describes Java System Properties in more details, looks at their usage within various applications and discusses the problems that they can cause.
- **Chapter 3** focuses on the data need to reason about System Properties and presents the first component of the tool, the *Analyser*.
- **Chapter 4** introduces the second component of the tool, the *Visualizer*, and shows how the *Properties Investigator* can be used to understand the usage of System Properties.
- **Chapter 5** discusses how the proposed tool can be improved and **Chapter 6** concludes the thesis

2 Java System Properties

In this chapter we introduce Java System Properties. We show how they can be accessed and modified and look at their usage within real world applications.

2.1 What are Java System Properties?

In Java instances of the class `Properties` are used to maintain a collection of key/value pairs. The `Properties` class is defined in the package `java.util` and inherits from `java.util.Hashtable`. `Hashtable` implements the interface `java.util.Map<K,V>`, which allows the keys and the values to have different types. As a result of the inheritance relation the `Properties` class also inherits this interface. However, the `Properties` class discourages the usage of this interface and instead provides an alternative API, in which both keys and values are `String` values. Storing other types of values or keys will lead to errors when working with a `Properties` object. The primary use case of the `Properties` class is to manage configuration values.

The `System` class from the Java SDK is a utility class providing several important features, like access to the standard input, standard output and standard error streams, access to environment variables, *etc*. It further manages a global instance of a `Properties` object, containing information about the configuration of the current working environment. *System Properties* in Java are nothing more than properties stored in this object.

As it is stored in a *private static* field, the `Properties` object can only be accessed through methods of the `System` class, listed in Table 1.

Table 1: Public static methods for working with the 'properties' field in the class System (Oracle, 2013)

Method	**Description**
`getProperties()`	*Returns the current system properties.*
`getProperty(String key)`	*Gets the system property indicated by the specified key.*
`getProperty(String key, String def)`	*Gets the system property indicated by the specified key.*
`setProperties(Properties props)`	*Sets the system properties to the Properties argument.*
`setProperty(String key, String value)`	*Sets the system property indicated by the specified key.*

Reading the values of System Properties can be done using one of the first three *get* methods. The difference between these methods is, on the one hand, in the return type and on the other hand in their parameters. `getProperties()` returns the `Properties` object responsible for storing System Properties. The value of a specific property can be accessed based on its key using `getProperty(String key)` or `getProperty(String key, String defautValue)`. The first will return `null` if there is no property with the given key, while the second will, in the same situation, return the `defautValue` parameter.

Creating custom System Properties or changing the value of existing ones can be done in multiple ways. After the program has started, the entire `Properties` object storing the current System

Properties can be changed using the method `setProperties(Properties props)`. Changing the value of a single System Property is done with the method `setProperty(String key, String value)`.

Changes made to the `Properties` object storing System Properties are not persistent. They will not exist in a future invocation of the Java interpreter as System Properties are re-initialized each time the Java runtime starts.

A third mechanism to modify System Properties or create new ones is with the *-D* option of the Java Virtual Machine. For example, launching a Java program with the line `java -Dfoo="some string" SomeClass` will set the System Property `foo` to the value `"some string"`.

Table 2: Possible ways to set one or multiple properties

Option	**Example**
Method setProperties(Properties props)	`java.util.Properties props = new Properties();` `System.setProperties(props);`
Method setProperty(String key, String value)	`System.setProperty("custom.property", "value");`
Console -Dproperty="value"	`java -Dfoo="some string" SomeClass`

Since the `System` class is public and all methods from Table 1 are also public, System Properties can be modified from within the entire code base. This further raises security concerns, especially when working with applets, where potentially untrusted code is running on the virtual machine of a user. To eliminate these security risks, Java uses a security manager to control how System Properties are accessed at runtime. This is implemented by the class `java.lang.SecurityManager`.

In the context of System Properties, the system checks if the code accessing a property has the necessary permission before performing any operation. This is done by calling the method `checkPermission(Permission perm)` of the `SecurityManager`, where a permission of the class `java.util.PropertyPermission` (extends `java.security.Permission`) is used. The possible permissions are read, write or none.

2.2 Identifying Java System Properties

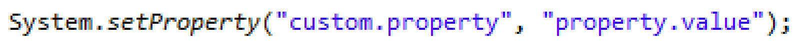
Because Java System Properties behave like global variables, they exhibit similar problems. These problems range from hidden dependencies that increase the difficulty of writing unit tests to naming conflicts. These problems can be addressed only if developers are able to understand how Java System Properties are used within an application. To achieve this they have to answer the following questions:

- From where is a System Property set?
- Where is it read?
- How can we determine where was a certain value set?
- What are all the values of a specific System Property during the execution?

In order to answer these questions, we have to locate all the places where a System Property is used. Before we can think about how we could get the necessary information, we have to look how a System Property can be specified.

There are four different possibilities to specify a custom System Property (we illustrated them using a `setProperty(..)` call):

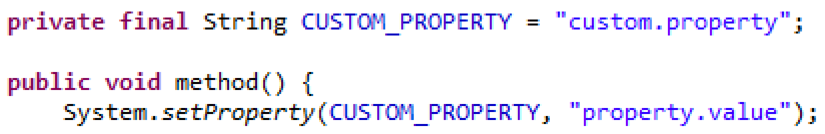
- The simplest possibility is to hard-code the key of a Property with a string literal in the method call.



```
System.setProperty("custom.property", "property.value");
```

Figure 2.1: Key of property defined by string literal

- One could also define a (static final) constant variable containing the key as a `String` object, and call the methods using this constant variable.

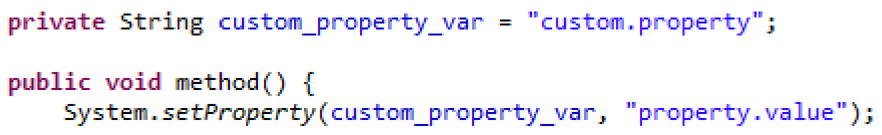


```
private final String CUSTOM_PROPERTY = "custom.property";

public void method() {
    System.setProperty(CUSTOM_PROPERTY, "property.value");
```

Figure 2.2: Key of property defined by constant variable

- The third possibility is to use a variable. This variable can either be an instance variable, a local variable or a method parameter.

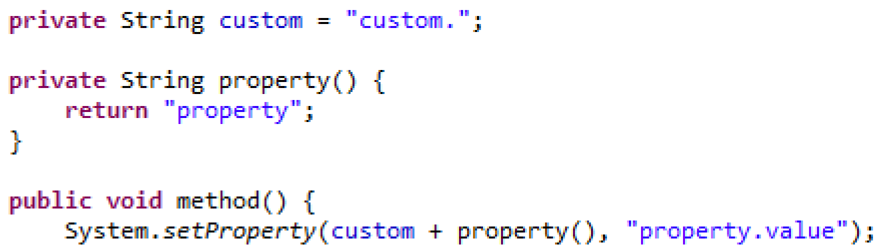


```
private String custom_property_var = "custom.property";

public void method() {
    System.setProperty(custom_property_var, "property.value");
```

Figure 2.3: Key of property defined by variable

- The fourth way is to compute the key of a property at runtime (using a method call, string concatenation, *etc.*).



```
private String custom = "custom.";

private String property() {
    return "property";
}

public void method() {
    System.setProperty(custom + property(), "property.value");
```

Figure 2.4: Key of property defined by string concatenation and method call

The keys of the properties defined in the first two ways can be known at compile time. On the other hand, the keys of the Properties defined with the third and fourth mechanism are only known at runtime. To determine these values, one has to use dynamic analysis.

2.3 Usage of Java System Properties

To learn how System Properties are used in practice, we performed an analysis on several Java projects from Pangea[1], an easy-to-setup repository for empirical studies. The analysis was done using Moose, an open-source platform for software and data analysis. The Java projects contained by Pangea were taken from the Qualitas Corpus[2] version 20120401 including 111 open-source Java systems.

The average size of the analysed systems is 100'809 NCLOC (the non-comment lines of code). They contain on average 780 classes. There are three projects, which we did not include in our analysis as

[1] http://www.moosetechnology.org/docs/pangea
[2] http://www.qualitascorpus.com

their large size caused parsing problems: *Eclipse* (two and a half million NCLOC), *gt2* (half a million NCLOC) and *Netbeans* (two million NCLOC).

For each project we identified all the places where a `getProperty(..)` or `setProperty(..)` method is called. Then we classified the calls in six categories, depending on the way in which their parameters are specified. This classification is presented in Table 3. One observation regarding this classification is that we consider a variable to be a constant if its name contains only uppercase letters. We did not explicitly check for the *final* and *static* modifiers.

Table 3: Categories used to classify the creation of a String object.

Category	**Description**
Literals	Hardcoded Strings like `"custom.property"`
Constants	Strings in Java constants like `CUSTOM.PROPERTY`
Variables	String variables that are not constants like `propertyVar`
Invocations	Method invocations returning a string like `propertyMethod()`
Concatenations	String concatenations like `"custom." + "property"` or even with methods
Constructors	Constructor calls like `new String("custom.property")` or `new Foo(a,b)`

Out of 111 projects, 107 are using System Properties. A few projects invoked the method `getProperty(…)` more than 200 times, but 75% of the projects invoke it less than 40 times. Calls to the method `setProperty(…)` were less frequent, as we can see in the diagrams below. Each bullet in the diagrams represents one of the 107 projects in Pangea that uses System Properties.

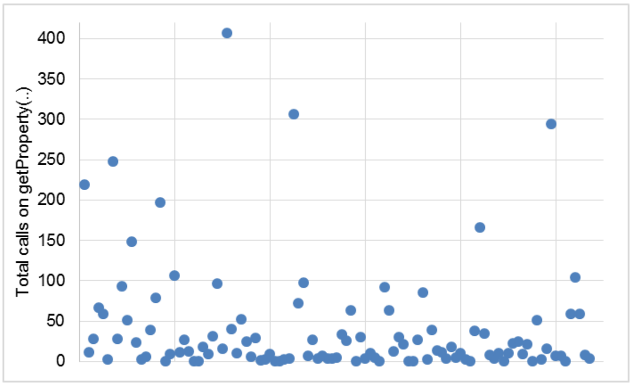


Figure 2.5: Total calls on getProperty(..) per project

Projects:	**Number of get calls:**
25% quartile	4 or less calls
median	13 or less calls
75% quartile	39.5 or less calls
100% quartile	407 or less calls

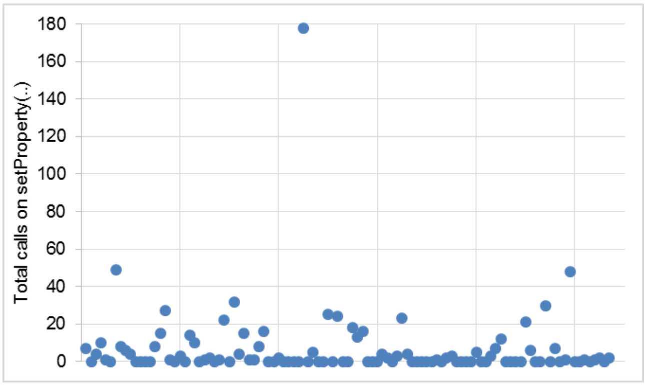


Figure 2.6: Total calls on setProperty(..) per project

Projects:	**Number of set calls:**
25% quartile	0 or less calls
median	1 or less calls
75% quartile	6.5 or less calls
100% quartile	178 or less calls

Projects making heavy use of System Properties include: Apache Tomcat, Apache Ant, Apache Hadoop, Azureus, JBoss.

Table 4: Results of analysis

Method	**Total**	**Literals**	**Constants**	**Variables**	**Invocations**	**Concatenations**	**Constructors**
getProperty(..)	4353	3886	239	200	6	19	3
setProperty(..)	739	572	122	42	2	1	0

Table 4 presents the results of the analysis. Out of all the calls of interest, 4353 were `getProperty(..)` calls (788 specified a default value in case the property did not exist). Because the last three categories had a shared percentage of less than one percent, we combined them into one category named *others*.

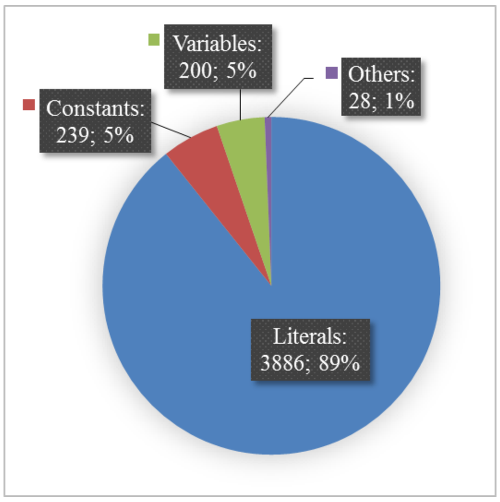


Figure 2.7: Calls to getProperty(..)

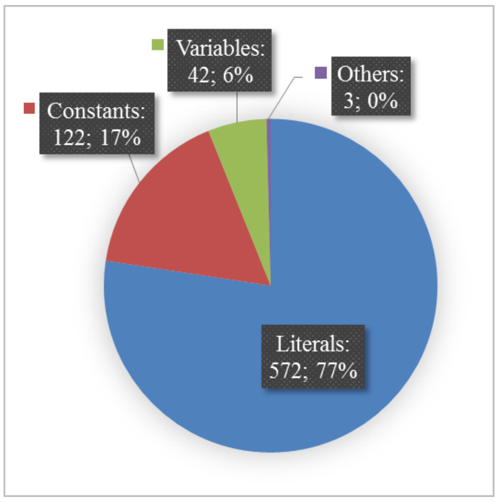


Figure 2.8: Calls to setProperty(..)

Figures 2.7 and 2.8 show the distribution of the results. We can see that, in most cases, Java System Properties are identified using string literals or constant variables. The value of these String objects is known at compile time. However, there are also cases, when the value can only be determined at runtime. As our aim in towards a complete analysis, we want to be able to also cover those cases.

To conclude, this analysis reveals that System Properties are used within Java systems. It further shows that they are identified not only using string literals and constants but also using String objects stored in variables or returned by method calls. This motivates the need to use an approach based on dynamic analysis to help developers reason about them.

3 Getting the data

This chapter looks at what kind of data developers need to successfully reason about System Properties. It then provides a short description of the Java Debug Interface (JDI), the infrastructure selected for extracting these data along with some alternatives. The second part of this chapter is about the Properties *Analyser*, the first half of the proposed tool. The chapter concludes by presenting some performance measurements.

3.1 What is the necessary data?

Before we can start implementing a tool that analyses calls on System Properties, we have to figure out what data can help developers reason about these properties.

First, to map a property call to a property, we need its *name* (the key used in the call). Then we need to know the *type of the call* (get/set) and the *value* of the property at that call. We also need the *code location* of a call: class, method and line number. Furthermore, we need a *chronology* of these calls, so that we can determine what call set a value and what calls used that value.

After identifying the necessary data, we need to decide if we use a static or a dynamic approach to extract it, in other words we need to decide if we want to run the application or not. With a static approach we could find out the locations of the calls. We would also get the properties defined in a static way (string literal or constant). This means that the keys created, for example by using a method or a string concatenation, may not be obtained. Getting the values of System Properties using a static approach would be much harder.

From Chapter 2 we already know that the keys used to identify System Properties are also defined dynamically. With a static analysis of the properties we would not get these calls. Although there are not many, these properties could be crucial. Therefore, we decided to use a dynamic approach for analysing the calls on Java System Properties based on the Java Debug Interface (JDI), a component of the Java Platform Debugger Architecture (JPDA).

3.2 Java Debug Interface (JDI)

3.2.1 What is JDI?

The Java Debug Interface (JDI) is a high-level application-programming interface (API) belonging to the Java SE platform. It is the top layer of the Java Platform Debugger Architecture (JPDA).

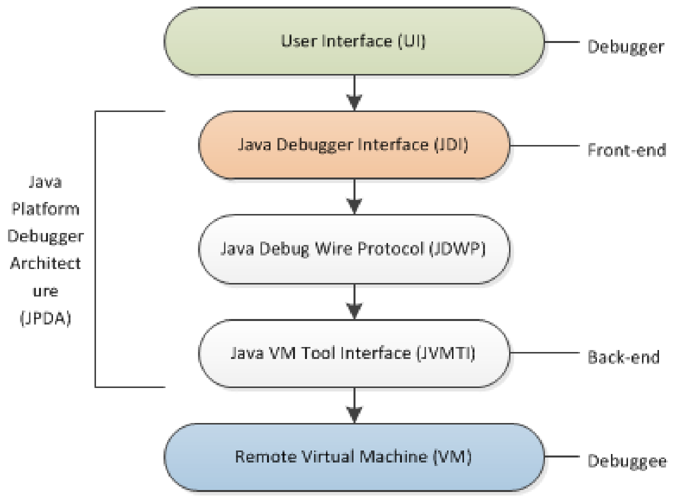

User Interface (UI)
Debugger
Java Debugger Interface (JDI)
Front-end
Java Platform Debugger Architect ure (JPDA)
Java Debug Wire Protocol (JDWP)
Java VM Tool Interface (JVMTI)
Back-end
Remote Virtual Machine (VM)
Debuggee

Figure 3.1: Overview of JPDA layers

The Java Debug Interface offers a mechanism for controlling and querying a Java virtual machine (VM). This functionality is useful for applications like debuggers that have to interact with a running system. Furthermore, JDI provides introspective access to the VM. This means that with JDI we have access not only to the state, but also to the VM's classes, arrays, interfaces, primitive types and also the instances of all those types.

Moreover, JDI lets users control a virtual machine's execution explicitly. One can suspend and resume threads, set breakpoints or watchpoints. If a thread is suspended, its state, local variables and stack trace can be inspected. Users can also be notified when different kinds of events are raised. For example, events are triggered when exceptions are raised, classes are loaded or threads are created.

We are using JDI to connect to a remote virtual machine and enable event requests for calls accessing System Properties. Additionally, we use JDI for processing the incoming events from the remote virtual machine and extracting the necessary data out of them.

3.2.2 Alternative solutions

3.2.2.1 Aspect Oriented Programming, AspectJ

As another possibility of dynamic analysis we looked at AspectJ, the implementation of Aspect Oriented Programming (AOP) for Java. Aspect Oriented Programming is a programming paradigm aiming to increase the modularity of a system by allowing the separation of cross-cutting concerns. A cross-cutting concern affects other concerns in an application and often cannot be decomposed from the rest of the application when using object-oriented or procedural programming. This can lead to

code duplication or increased dependencies. A common cross-cutting concern is, for example, data logging. By using AOP, it is possible to encapsulate such cross-cutting concerns into aspects.
AspectJ extends Java with a new concept: *join points*. It further adds several new constructs: pointcuts, advices, inter-type declarations and aspects. Pointcuts and advices influence the execution flow, whereas inter-type declarations can alter the class hierarchy of a program. These constructs are then encapsulated into aspects. To understand AspectJ better, the constructs are briefly described in the Tables 5 and 6.

Table 5: Description of the constructs in AspectJ.

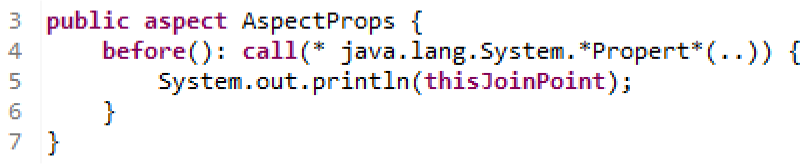
Type	**Definition**	**Example**
Join Point	Well-defined point in the program flow	A method call
Pointcut	Selects each join point with a specified declaration	A method call on the method `ClassA.methodA(int)` of class `ClassA` with one integer parameter
Advice	Piece of code that is executed when a join point is matched by a pointcut	See Table 6 and Figure 3.2 (Advice is between {…})
Inter-type declaration	Add custom methods or fields to an existing class from within an aspect	Add method `someMethod()` to `SomeClass`

There are three different types of advices. The difference between them is the moment at which the defined piece of code is being executed.

Table 6: Types of advices in AspectJ

Type	**Definition**	**Example**
Before advice	Code is executed before the program continues with a selected join point	Before calling `System.getProperty(..)`, do something
After advice	Code is executed after the program has continued with a selected join point	After calling `System.getProperty(..)`, but before the return value is returned, do something
Around advice	Code is executed when a join point is selected and has explicit control over the resumption of the join point	When calling `System.getProperty(..)`, do something and control resumption of the call

We could use AspectJ to get the required data by capturing calls on System Properties using pointcuts and aspects to extract the actual data.



```
public aspect AspectProps {
    before(): call(* java.lang.System.*Propert*(..)) {
        System.out.println(thisJoinPoint);
    }
}
```

*Figure 3.2: Before advice when the pointcut selects the join point of calling a method *Propert* of class `java.lang.System`*

We decided not to use AspectJ, since we could not get all required data. On the one hand, we do not get calls from reflection. For example, a pointcut `call(void run())`, won't select a call using reflection, like `((Method) run).invoke(args)`. Therefore, we would have to implement a workaround. On the other hand, we also do not get calls from external libraries with the usual pointcuts. There are some approaches to solve this problem, for example changing the byte code of the library or using load time weaving, but they are not trivial to implement.

3.2.2.2 *Instrumenting class java.lang.System*

A second alternative consists in instrumenting the class `System` from the Java Standard Library. We did not use it, as it would have required shipping our adapted JDK with the tool. This could end up causing compatibility or update problems. Furthermore, instrumenting core Java classes could cause additional problems, since the Java Quick Starter preloads some of the classes during the start-up of the operating system (Haase, 2007).

3.3 The Properties Analyser

After identifying the necessary data and the means to extract it, we are able to look at the Properties *Analyser*. The Analyser is the first half of our tool.

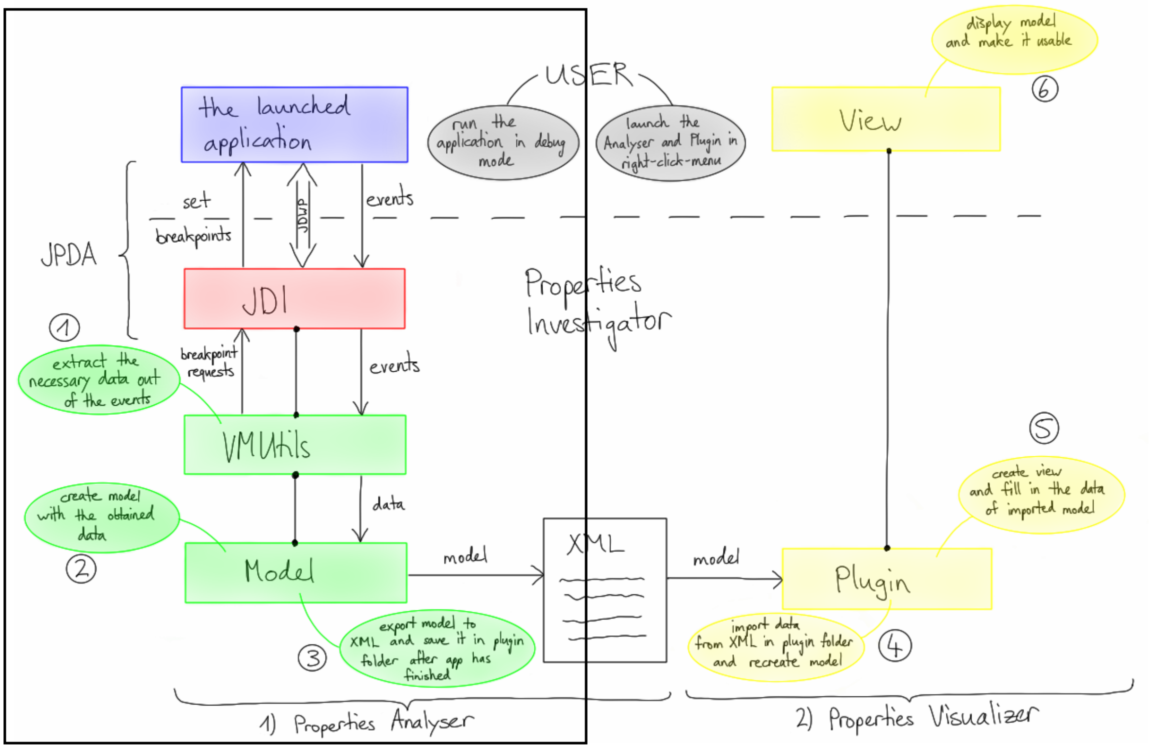

USER
run the application in debug mode
launch the Analyser and Plugin in right-click-menu
the launched application
set breakpoints
JDWP
events
JPDA
JDI
Properties Investigator
①
extract the necessary data out of the events
breakpoint requests
events
VMUtils
data
create model with the obtained data
②
Model
model
XML
model
③
export model to XML and save it in plugin folder after app has finished
1) Properties Analyser
display model and make it usable
⑥
View
⑤
create view and fill in the data of imported model
Plugin
import data from XML in plugin folder and recreate model
④
2) Properties Visualizer

Figure 3.3: Sketch of the whole workflow of our tool with focus on the first half, the Analyser

The Analyser performs three main tasks:

- It connects to a remote virtual machine and gets the events of interest (calls to methods accessing/altering System Properties) by using the Java Debug Interface (Figure 3.3 → 1).
- It extracts the required data from these events and stores it in the Properties Model (2).
- It persists the created Properties Model in a XML file (3).

After connecting to the virtual machine running the target application in debug mode, the Analyser uses JDI to get notified when the events of interest occur. This is achieved by creating breakpoint requests. Because we are only interested in events from the class `System`, a class filter is used.

During the execution of the program, the Analyser receives the events of interest. While JDI requests come from the debugger side, events originate on the debuggee side. Events inform the debugger about changes of the state in the debugged application (*e.g.* application hits a breakpoint). Out of these events the Analyser gets the data about the calls on System Properties. The Properties Model, at which we will look next, is then populated using these data.

3.3.1 The Properties Model

To store the required data we designed a model consisting of a list of `Property` objects stored by an instance of a `PropertiesModel` class. For each System Property, called at least once, we create a new `Property` object. A `Property` object has two attributes, a name and a list of `CallSite` objects. A `CallSite` has a `Location`, identifying the class, method and line number of the associated call, and a list of `PropertyValue` objects. Each `PropertyValue` object stores a value of the corresponding `Property` at that call location. A list is required, as during execution, the value of a System Property can change at a specific call location.

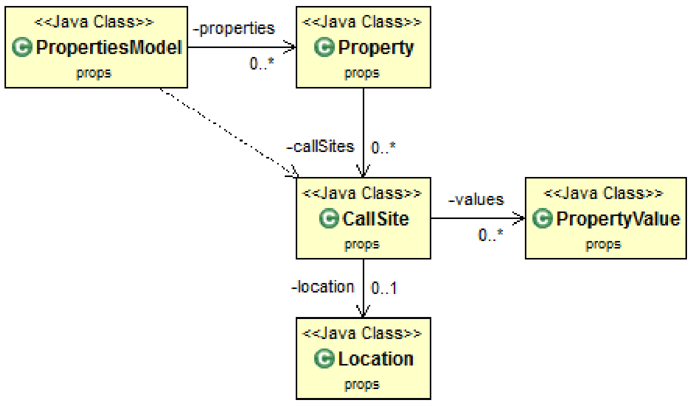


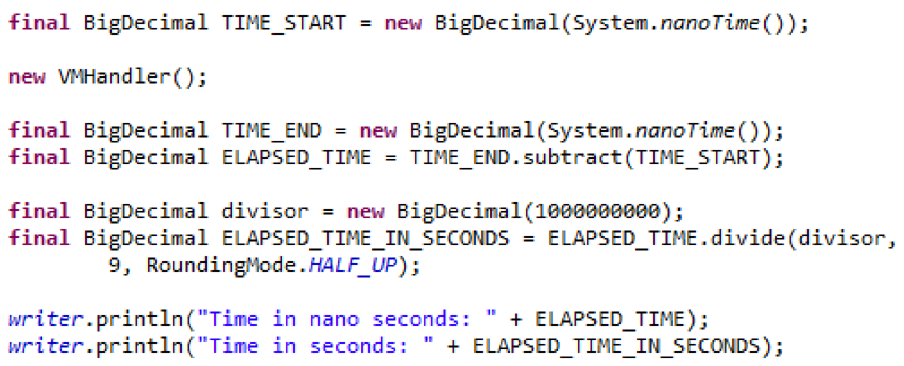
Figure 3.4: UML diagram of important objects in Properties Model

To order the calls to a System Property, we use a static integer variable, `id`. Every time a new `CallSite` or a `PropertyValue` is created, the `id` is being increased. This approach works, as when an event is triggered, JDI block the entire virtual machine; thus the order is preserved even in multithreaded contexts.

After getting the data and creating the model, we export it into an XML file. To simplify this step, our tool uses the Java Architecture for XML Binding (JAXB) to marshal and unmarshal objects.

3.4 Measurements

To determine the speed of the Analyser, we performed a series of measurements on several simple examples. The measurements were done using the features provided by the class `java.lang.System`; the elapsed time was computed by using `BigDecimal` numbers.



```
final BigDecimal TIME_START = new BigDecimal(System.nanoTime());

new VMHandler();

final BigDecimal TIME_END = new BigDecimal(System.nanoTime());
final BigDecimal ELAPSED_TIME = TIME_END.subtract(TIME_START);

final BigDecimal divisor = new BigDecimal(1000000000);
final BigDecimal ELAPSED_TIME_IN_SECONDS = ELAPSED_TIME.divide(divisor,
        9, RoundingMode.HALF_UP);

writer.println("Time in nano seconds: " + ELAPSED_TIME);
writer.println("Time in seconds: " + ELAPSED_TIME_IN_SECONDS);
```

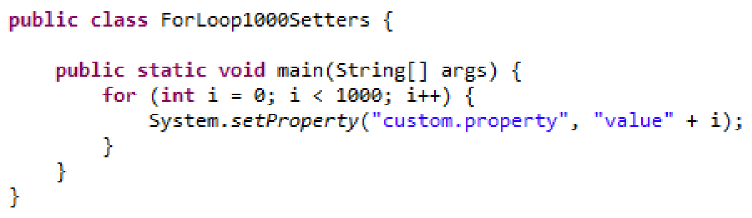
Figure 3.5: Snippet of the main method in the class VMHandler

The code above measures the time it takes for the whole workflow of the Analyser to complete. This includes starting the tool, connecting to the remote virtual machine, constructing the model and finally exporting the model to an XML file. To have a reference point, we did a first series of measurements without using the Analyser.

Table 7: Results of the measurements in seconds
(System: CPU i7-2600K @ 3.4GHz, RAM 16 GB DDR3 1333 MHz, HD 1TB SATA/600 7200 RPM, Windows 7)

Method	**With JDI:**	**Without JDI:**
1000 Get Calls	3.3768 s	0.0005 s
1000 Set Calls	4.8402 s	0.0027 s
1000 Get and Set Calls	6.9876 s	0.0035 s
Junit Test (1000 Get Calls with an assert)	15.0524 s	0.0015 s
Demo App	2.5386 s	0.0036 s

The results of the measurements are shown above in Table 7. For each one we run the application five times and took the lowest value.



```
public class ForLoop1000Setters {

    public static void main(String[] args) {
        for (int i = 0; i < 1000; i++) {
            System.setProperty("custom.property", "value" + i);
        }
    }
}
```

Figure 3.6: Simple application, which does 1000 set calls in a for-loop

As the results show, there is a significant decrease in performance when the tool is used. A possible explanation for this degradation is that we use two JDI events, `MethodEntryRequests` and `MethodExitRequests`, although in the end we are using only `MethodExitRequests`. Additionally, by using JDI the Java virtual machine switches to interpreted-only mode (like when using the JVM option `-Xint`). As a result, compilation to native code is disabled and all bytecodes are executed by the interpreter. Therefore, the performance decreases. In his bachelor's thesis, Andreas Elsner from the University of Paderborn in Germany got similar results (Elsner, 2004).

Another reason for the big times with JDI could be the filtering of events. Although we filter the events with a class filter (`java.lang.System`), the virtual machine will still send events for all classes to the Java Debug Wire Protocol (JDWP) agent. The agent filters then the events and sends the filtered results to JDI.

4 The Properties Investigator

In this chapter we present the Properties Investigator, the proposed tool for reasoning about System Properties. We first look at some possible use cases of the Properties Investigator. We then describe the second half of the tool, the *Visualizer*. Finally we explain how the tool can be used.

4.1 Use cases

4.1.1 What System Properties am I using in my application?

When working with a large number of System Properties one could forget which names are already used. To avoid naming conflicts one would have to go through the whole code of the application and list the names.

Our tool provides a convenient way to automate this task. It performs an analysis that reveals the names of all System Properties used at least once in the application.

4.1.2 What values do System Properties have in my application?

Another common question that appears when working with System Properties is *'What are the runtime values of a given Property'*. Unfortunately, this information is not always obviously from the code. Without a dynamic analysis, one has to collect it manually.

Our tool provides this information out of the box. Every call to a System Property is augmented with all the values that occur at runtime. Furthermore, for a specific System Property, one can display an overview showing the previous and the next value of that property (if the property had more than one value).

4.1.3 From where are System Properties called within my application?

After the analysis, the plugin shows an overview of all calls and allows developers to go directly into the source code to see the locations. Furthermore, the tool makes it possible to navigate chronologically through the calls on a System Property.

4.2 The Properties Visualizer

The goal of this component is to present the extracted data to users and help them understand the usage of System Properties. It is implemented as an Eclipse plugin.

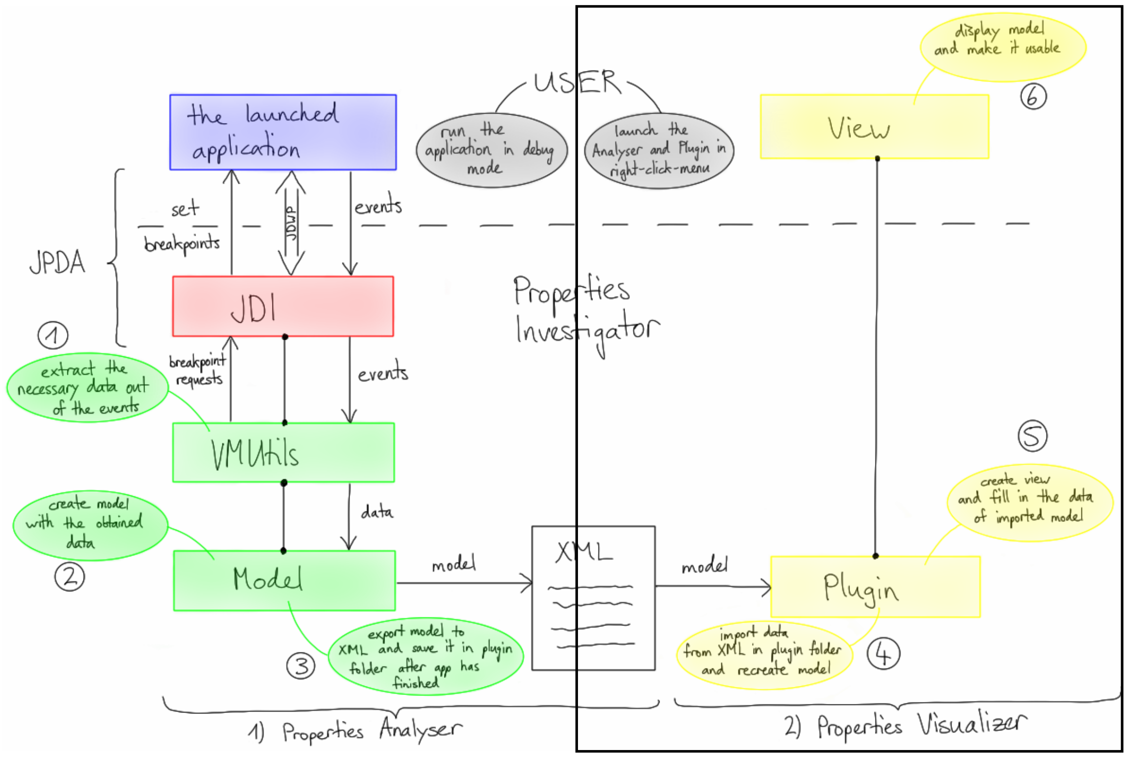
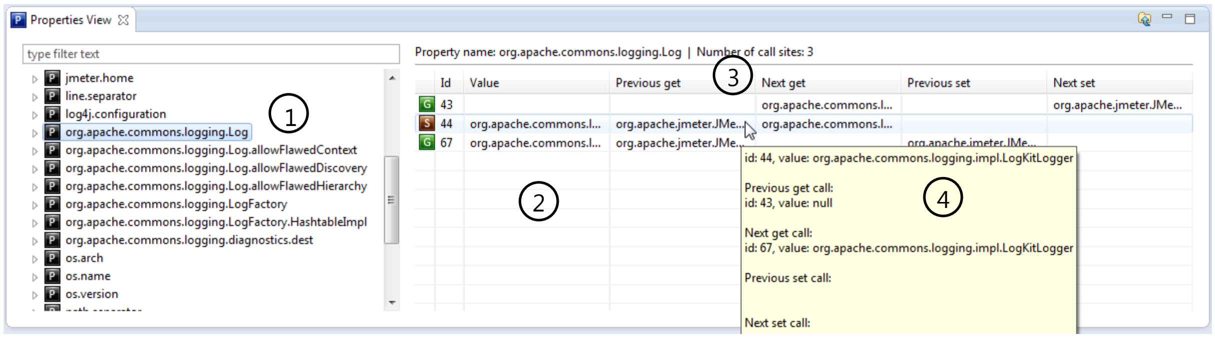

USER
run the application in debug mode
launch the Analyser and Plugin in right-click-menu
display model and make it usable
6
the launched application
View
set breakpoints
JDWP
events
JPDA
JDI
Properties Investigator
1
extract the necessary data out of the events
breakpoint requests
events
VMUtils
5
create view and fill in the data of imported model
create model with the obtained data
2
data
Model
model
XML
model
Plugin
export model to XML and save it in plugin folder after app has finished
3
import data from XML in plugin folder and recreate model
4
1) Properties Analyser
2) Properties Visualizer

Figure 4.1: Sketch of the whole workflow of our tool with focus on the second half, the Visualizer

Like the *Analyser*, the *Visualizer* has to accomplish three tasks:

- After the Analyser has finished the analysis and exported the data into an XML file, the Visualizer imports it and recreates the Properties Model (Figure 4.1 → 4).
- Then it creates an Eclipse view and populates it with data from the Properties Model (5).
- Once this is completed it allows user to explore and navigate through the data (6).



jmeter.home
line.separator
log4j.configuration
org.apache.commons.logging.Log
org.apache.commons.logging.Log.allowFlawedContext
org.apache.commons.logging.Log.allowFlawedDiscovery
org.apache.commons.logging.Log.allowFlawedHierarchy
org.apache.commons.logging.LogFactory
org.apache.commons.logging.LogFactory.HashtableImpl
org.apache.commons.logging.diagnostics.dest
os.arch
os.name
os.version

Property name: org.apache.commons.logging.Log | Number of call sites: 3

	Id	Value	Previous get	Next get	Previous set	Next set
G	43			org.apache.commons.l...		org.apache.jmeter.JMe...
S	44	org.apache.commons.l...	org.apache.jmeter.JMe...	org.apache.commons.l...		
G	67	org.apache.commons.l...	org.apache.jmeter.JMe...		org.apache.jmeter.JMe...	

id: 44, value: org.apache.commons.logging.impl.LogKitLogger

Previous get call:
id: 43, value: null

Next get call:
id: 67, value: org.apache.commons.logging.impl.LogKitLogger

Previous set call:

Next set call:

Figure 4.2: Overview of the Visualizer

The view of the Visualizer consists of two parts. On the left there is a tree list of all System Properties encountered during the analysis, along with their call-sites. If a developer is only interested in specific properties or call-sites, he can use the filter from the top of the property list.
After selecting a property or a call-site (Figure 4.2 → 1), the corresponding values are shown in the table from the right part of the view (2). Furthermore, the table also shows the previous and the

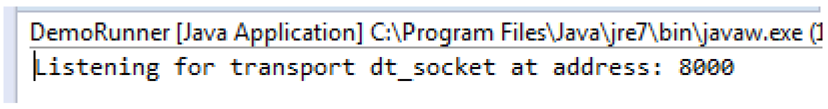
following *get* and *set* calls on the selected property, if present (3). By hovering on a value, the plugin view will display the previous or next values directly in the popup (4).

4.3 How can we use the plugin?

The Visualizer, together with the Analyser, forms the Properties Investigator. In this section we will look at how the tool can be used, as a whole. The first step is to launch the application, for which the usage of the System Properties has to be traced, in debug mode. To do this, the following arguments have to be passed to the Java Virtual Machine (VM) that will run the application:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=y
```

With these arguments the remote VM runs the application in debug mode with the JDWP transport option “dt_socket” on port “8000”. Furthermore, the application is suspended until the debugger has connected and resumed the remote VM.



```
DemoRunner [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (1
Listening for transport dt_socket at address: 8000
```

Figure 4.3: Status info from the VM of the launched application

Now the plugin can be started by right-clicking on the corresponding project in the Project Explorer or Package Explorer and selecting the command “Analyse System Properties Calls”. This resumes the VM and the Analyser starts collecting the data.

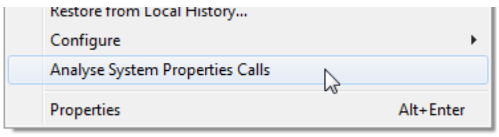

Restore from Local History...
Configure
Analyse System Properties Calls
Properties
Alt+Enter

Figure 4.4: Bottom of the right-click menu of a Java project

When the application has finished executing the plugin automatically opens the Properties View[3], showing the obtained data, if it is not already opened. It is also possible to open the view manually from the register “Window → Show view → Other…”.

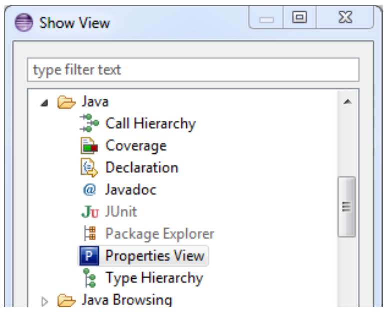


Figure 4.5: Java section of Eclipse views

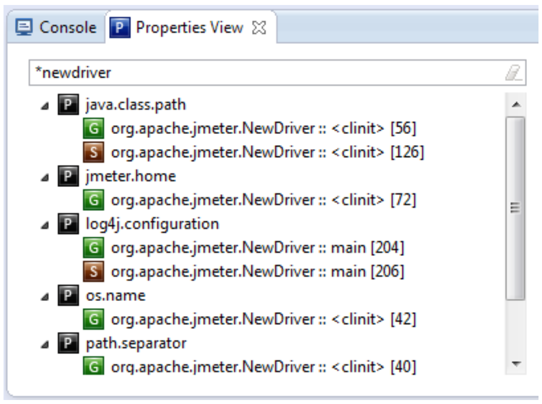
Henceforth, the data about the obtained calls on Java System Properties is displayed in the view of the Visualizer, alphabetically sorted by the names of the properties. As we already saw, the list containing

[3] All the screenshots were taken after an analysis of the application Apache JMeter.

the property names is implemented as a tree, so it is possible to click on a specific property to see its call-sites. Additionally, to see a call-site directly in the code, simply double-clicking on it will open the source file, when available.

In this list we find all the names of the properties and the locations from where they were called. As mentioned in Chapter 4.1, two of the use cases were about getting this information.

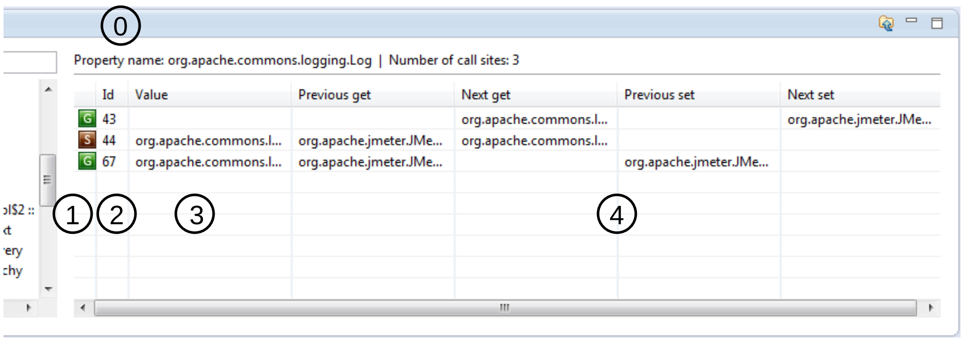
The text field at the top of the list of property names, can be used to filter the properties using regular expressions with a '*' for any string or a '?' for any character. This filter is useful when searching for properties with a specific name or when trying to locate properties called from a specific class or method.



```
*newdriver
◢ P java.class.path
      G org.apache.jmeter.NewDriver :: <clinit> [56]
      S org.apache.jmeter.NewDriver :: <clinit> [126]
◢ P jmeter.home
      G org.apache.jmeter.NewDriver :: <clinit> [72]
◢ P log4j.configuration
      G org.apache.jmeter.NewDriver :: main [204]
      S org.apache.jmeter.NewDriver :: main [206]
◢ P os.name
      G org.apache.jmeter.NewDriver :: <clinit> [42]
◢ P path.separator
      G org.apache.jmeter.NewDriver :: <clinit> [40]
```

Figure 4.6: Filtered list of properties by class name

After selecting a property, the values of all call-sites from where that property is accessed are shown in the right half of the view. By selecting a specific call-site in the properties list, the view will only show the values for that call-site.



Property name: org.apache.commons.logging.Log | Number of call sites: 3

	Id	Value	Previous get	Next get	Previous set	Next set
G	43			org.apache.commons.l...		org.apache.jmeter.JMe...
S	44	org.apache.commons.l...	org.apache.jmeter.JMe...	org.apache.commons.l...		
G	67	org.apache.commons.l...	org.apache.jmeter.JMe...		org.apache.jmeter.JMe...	

Figure 4.7: The table with the values of a selected property

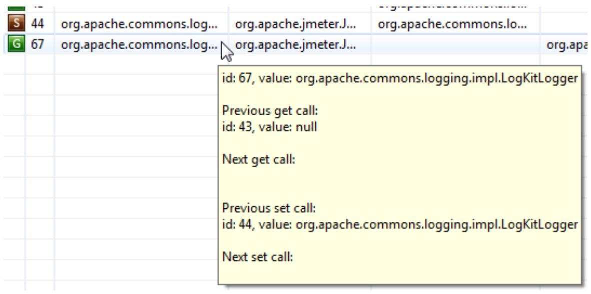
In a label right above the table (Figure 4.7 → 0) we can see either the name of the currently selected property and the number of call-sites from where it was accessed, or the location of the currently selected call-site.

The first column of the table shows the type of the call-site (1): get or set. In the second column the *id* of the value is shown (2). As we already know, the *id* is a static variable and is increased for every new value. The values of the selected property are displayed in the third column (3). To see the location of a value directly in the source code, it is possible to double-click on the value.

The remaining columns show the previous, respectively the next calls on the selected property (4). These columns show the corresponding call-site. When double-clicking on one of these call-sites the view opens and displays it in the table of values and also opens the source file, if available. This implies that it is possible to navigate through the call-sites of a property.

Knowing the order in which the values were set, is also a use case we defined above. Navigating through the call-sites can help developers understand how a property got to have a certain value.

Last but not least, a tool tip is displayed when hovering over an item. In the properties list the tool tip shows information about the call-site, like the class in which the property was called. In the right side of the view, where the values are displayed, the tool tip shows directly the previous and next values of the hovered value. Using tooltips this information is accessible without switching the call-site.



S	44	org.apache.commons.log...	org.apache.jmeter.J...	org.apache.commons.lo...	
G	67	org.apache.commons.log...	org.apache.jmeter.J...		org.apa

```
id: 67, value: org.apache.commons.logging.impl.LogKitLogger

Previous get call:
id: 43, value: null

Next get call:


Previous set call:
id: 44, value: org.apache.commons.logging.impl.LogKitLogger

Next set call:
```

Figure 4.8: Tool tip of a Property Value

5 Future work

5.1 Properties Analyser

The Properties Analyser has some limitations as we only trace calls to the methods `getProperty(..)` and `setProperty(..)` of the `System` class. However, System Properties can also be accessed using the methods `getProperties()` and `setProperties(..)`, working with the entire `Property` object. For example, importing System Properties from a file would most likely use these methods.

A second problem with the current implementation is that it can take a long time to extract the necessary data. To address this, the Analyser should be optimised. The question here is how far the tool could be optimised, since we are limited by the speed of the Java Debug Interface.

5.2 Properties Visualizer

On the side of the Eclipse plugin, the process of starting the tool should be simplified. Users should not have to manually launch the application in debug mode. This could be addressed by providing a dedicated launch configuration.

Another feature that could improve the user experience is a tooltip showing up directly in the code editor of Eclipse, when hovering over a call involving a System Property. When reading code developers would immediately see what value was returned by a `getProperty(..)` call, or what value was overridden by a `setProperty(..)` call, after an analysis of their application.

6 Conclusions

System Properties are a common presence within Java applications. Since they behave like global variables, they are difficult to understand. This problem cannot be manually addressed given the complexity of today’s systems. Apart from that, identifying the System Properties used from a method or a class is not always possible without running the program.

Therefore, developers need tool support to understand the usage of Java System Properties. In this thesis we have implemented such a tool, the *Properties Investigator*. To get the necessary information we used an approach based on dynamic analysis implemented using the Java Debug Interface. We then display this information to the user by using an Eclipse plugin. Using it developers can view the values and navigate through the call-sites of System Properties. As a result, they are able to better understand the usage of Java System Properties within the traced applications.

7 References

Eclipse. (2013, 09 08). *Introduction to AspectJ*. Retrieved from Eclipse AspectJ Docs: http://eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

Elsner, A. (2004, 09). Selektive Aufzeichnung von Laufzeitinformationen bei der Ausführung von Java-Programmen. *Bachelor's Thesis*, 41-42. Paderborn, Germany. Retrieved 10 28, 2013, from http://www.fujaba.de/uploads/tx_sibibtex/BachelorAElsner.pdf

Haase, C. (2007, 05). *Consumer JRE: Leaner, Meaner Java Technology*. Retrieved 10 26, 2013, from Oracle Technology Network, Articles, Java Platform: http://www.oracle.com/technetwork/articles/javase/consumerjre-135093.html

Oracle. (2013). *Java™ Platform Debugger Architecture (JPDA)*. Retrieved 09 05, 2013, from Oracle Java™ SE 7 Documentation: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/index.html

Oracle. (2013, 09 16). *Java™ SE 7 API class System*. Retrieved from Oracle Java™ SE 7 API: http://docs.oracle.com/javase/7/docs/api/java/lang/System.html

Wikipedia. (2013, 08 30). *Global variable*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Global_variable