Analysing Java System Properties

Implementation

Supplementary documentation to the 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

This documentation describes the implementation of the Properties Investigator. It introduces the Java Platform Debugger Architecture (JPDA), looks at the Java Debug Interface (JDI) and then presents the two main components of the proposed tool, namely the Analyser and the Visualizer.

Contents

1 Standard System Properties

From the thesis we know the various issues that can arise when using System Properties in Java. Below is a list containing the standard System Properties in Java. They are initialized when the Java Virtual Machine is launched. Apart from them, users are free to define their own custom properties.

Table 1: The standard system properties in Java 7, (Oracle, 2013)

Key	**Description of Associated Value**
java.version	Java Runtime Environment version
java.vendor	Java Runtime Environment vendor
java.vendor.url	Java vendor URL
java.home	Java installation directory
java.vm.specification.version	Java Virtual Machine specification version
java.vm.specification.vendor	Java Virtual Machine specification vendor
java.vm.specification.name	Java Virtual Machine specification name
java.vm.version	Java Virtual Machine implementation version
java.vm.vendor	Java Virtual Machine implementation vendor
java.vm.name	Java Virtual Machine implementation name
java.specification.version	Java Runtime Environment specification version
java.specification.vendor	Java Runtime Environment specification vendor
java.specification.name	Java Runtime Environment specification name
java.class.version	Java class format version number
java.class.path	Java class path
java.library.path	List of paths to search when loading libraries
java.io.tmpdir	Default temp file path
java.compiler	Name of JIT compiler to use
java.ext.dirs	Path of extension directory or directories
os.name	Operating system name
os.arch	Operating system architecture
os.version	Operating system version
file.separator	File separator ("/" on UNIX)
path.separator	Path separator (":" on UNIX)
line.separator	Line separator ("\n" on UNIX)
user.name	User's account name
user.home	User's home directory
user.dir	User's current working directory

2 The Java Platform Debugger Architecture (JPDA)

In this chapter we introduce the Java Platform Debugger Architecture (JPDA) and look at how it can be used to write debugging applications.

2.1 Introducing JPDA

The Java Platform Debugger Architecture is a debugging architecture with multiple layers. It allows developers to create applications for debugging that do not depend on the platform, the VM implementation or the JDK version.

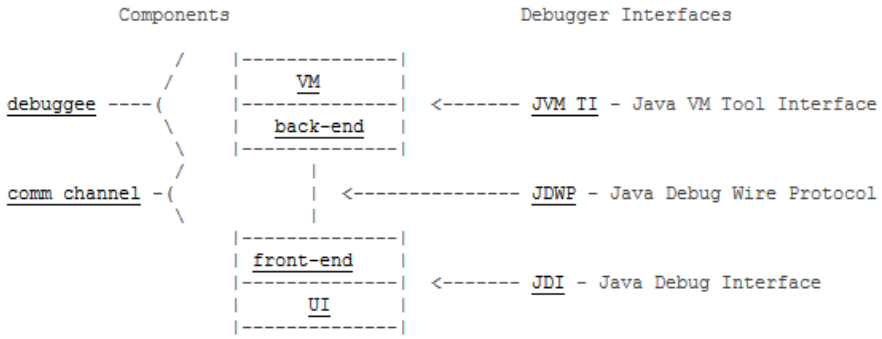

Figure 2.1: Structure overview of the Java™ Platform Debugger Architecture (Oracle, 2013)

As we can see in Figure 2.1, JPDA consists of three layers. At the back-end there is the Java VM Tool Interface (JVMTI). This interface replaces the old Java Virtual Machine Debug Interface (JVMDI) and defines the debugging services provided by the virtual machine. The Java Debug Wire Protocol (JDWP) is then responsible for the communication between the debuggee and debugger processes. Finally, at the front-end is the Java Debug Interface, the component that we use within the thesis. Its goal is to provide a high level API for accessing the Java VM Tool Interface.

Developers that need to write debugging applications can hook into JPDA at any layer. If needed, they could implement an entirely new front-end or rely on another virtual machine that implements JDWP. However, this is usually not required unless radical new features are needed. Most of the times developers only need to rely on JDI. This way, their application will automatically work with all virtual machines and platforms supported by Java.

2.2 Working with JPDA

To understand how the Java Platform Debugger Architecture works we will look at two types of activities one can do with it: query information and place breakpoints.

Any JPDA activity is started with a *request*. Requests are issued from the debugger side. They can include queries for information (*e.g.* variables values), instructions for changing the state of the virtual machine, instructions for placing breakpoints, instructions for controlling the execution of the virtual machine, *etc.*

Some types of requests are synchronous, providing immediate results to the caller, while others are asynchronous (*e.g.* breakpoints). JPDA responds to the second type of requests by using *events*. Events originate on the debuggee side, which means they come from the remote virtual machine running the application being debugged. They provide the debugger with the requested information. For example, if the debugger issues a request to be notified of classes being loaded into the system, whenever a new class is loaded an event is send to the debugger.

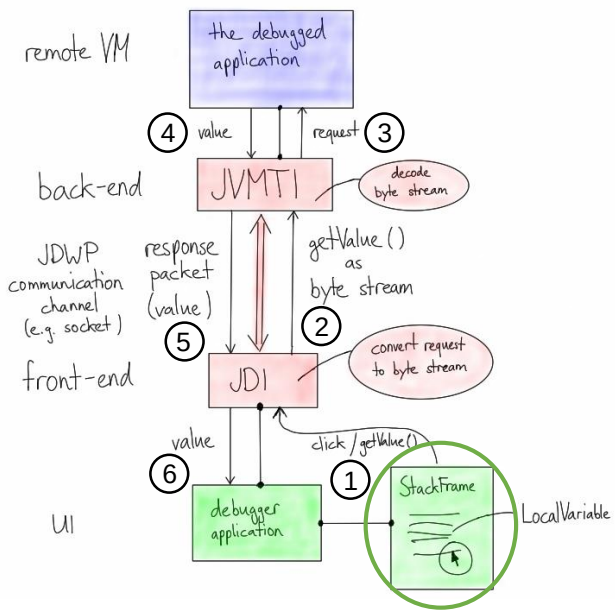

Figure 2.2: Workflow of JPDA for a request

A common use case when debugging is inspecting the values of variables from the stack (Figure 2.2 → green circle). To do this the debugger will issue a request for the desired value (1). This is achieved by calling the method `StackFrame.getValue(LocalVariable var)` on the stack frame containing the value of interest.

JDI converts the request then into a byte stream in accordance with JDWP and sends it over the defined communications channel, for example a socket, to the back-end (2). The back-end decodes the byte stream and sends the request through JVMTI to the remote virtual machine running the debugged application (3).

The remote virtual machine answers the request by returning the desired value (4). The back-end formats the response into a packet, which includes the value, according to JDWP. Afterwards the packet is sent back to the front-end through the communication channel (5). There the response packet is decoded and the value is returned as the result of the method call above (6). The debugger application can now display the requested value.

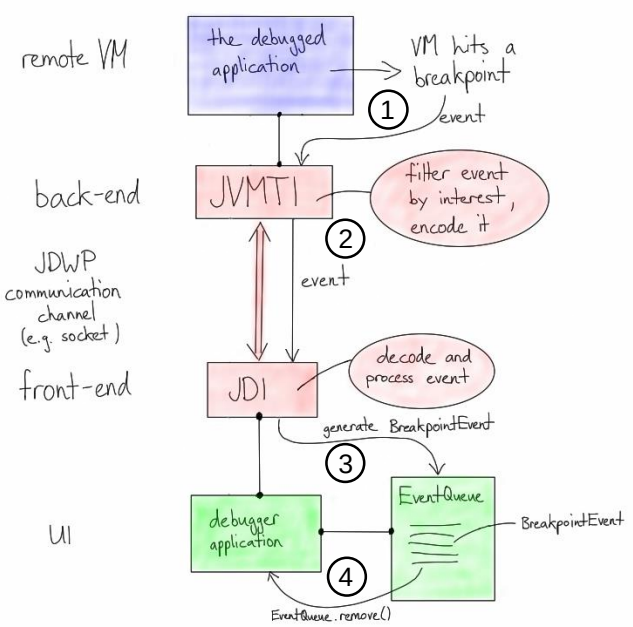

Figure 2.3: Workflow of JPDA for an event

A request for setting a breakpoint has a different workflow: when the request is made, control returns to the debugger immediately. The virtual machine continues the execution of the target program until that breakpoint is reached. When this happens the remote virtual machine sends an event through JVMTI (Figure 2.3 → 1). JVMTI, in turn, calls an event handling function set when issuing the initial request. This function filters the event, if required, queues it and afterwards converts and sends it through the communications channel according to JDWP (2). JDI then decodes it and generates a `BreakpointEvent` (3). Finally, the debugger application can get the event by extracting it from the `EventQueue` (4).

3 Implementation of Properties Analyser and Model

In this chapter we look at the implementation of the Properties *Analyser*. First we describe the package `VMUtils`, which uses the Java Debug Interface (JDI) to get the necessary data. Then we give a description of the Properties Model and look at how it is constructed and exported into an XML file.

3.1 VMUtils and JDI

The main class of the `VMUtils` package is the `VMHandler`. This class starts and control the lifecycle of the remote virtual machine.

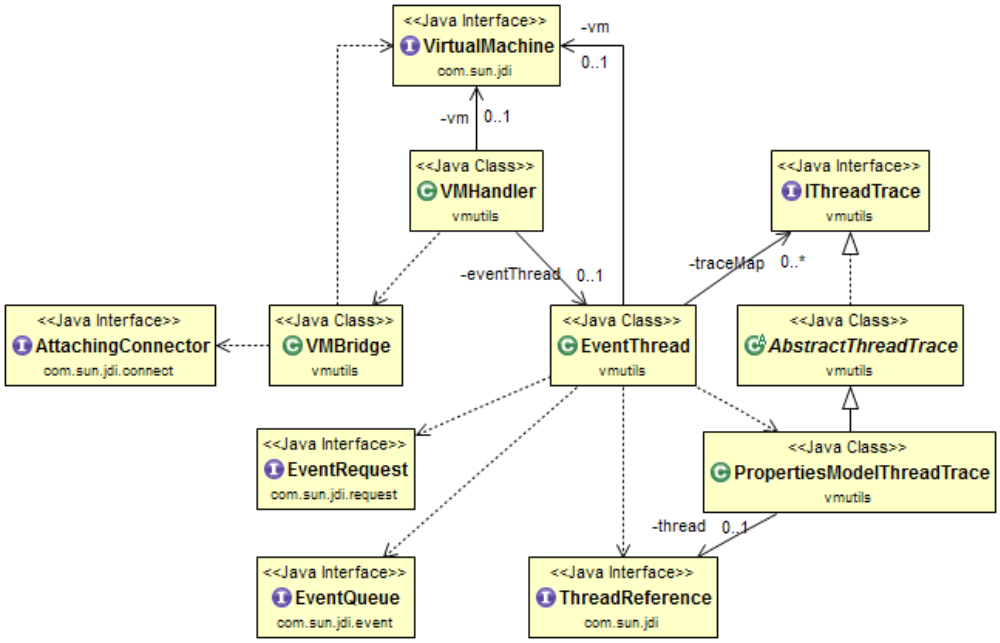

Figure 3.1: UML diagram of VMUtils with context to JDI

The `VMHandler` uses a `VMBridge` object to acquire, respectively connect to the remote virtual machine through a defined port.

```
public VMHandler(int port) throws IOException {
    vm = new VMBridge().acquireVM(port);
    generateEvents(writer);
}
```

Figure 3.2: Constructor of class VMHandler

To establish a connection to the target virtual machine, `VMBridge` uses a `Connector` object. We rely on an `AttachingConnector` connector that can be used to connect to a running VM through a socket. This connector is obtained from the `VirtualMachineManager`, an object that manages the connection to a target virtual machine. The procedure for connecting to a remove virtual machine is shown in Table 2.

Table 2: Connecting the debugger to the virtual machine (Oracle, 2013).

Scenario	Description
Debugger attaches to previously-running VM	1) Target VM is launched using the options `-agentlib:jdwp=transport=xxx,server=y` 2) Target VM generates and outputs the tranport-specific address at which it will listen for a connection. 3) Debugger is launched. Debugger selects a connector in the list returned by `attachingConnectors()` matching the transport with the name "xxx". 4) Debugger presents the default connector parameters (obtained through `Connector.defaultArguments()`) to the end user, allowing the user to fill in the transport-specific address generated by the target VM. 5) Debugger calls the `AttachingConnector.attach(java.util.Map)` method of the selected connector to attach to the target VM. A VirtualMachine mirror is returned.

The `AttachingConnector` that we are using must have the name "`com.sun.jdi.SocketAttach`". Connectors having this name implement a socket based communication with a VM on top of TCP/IP. By using the manager and the described connector, we can get a mirror of the target virtual machine, which is returned to the `VMHandler`. It generates then a new `EventThread`.

The `EventThread` sets the event requests and enables them. Here we also add a class filter, since we are only interested in calls on the class `java.lang.System`. This thread is then used to handle the incoming events. In the `run()` method of this thread we get the JDI `EventQueue` (see Figure 3.3). `EventQueue` is a manager of incoming debugger events from a target VM (A particular virtual machine is assigned one instance of an `EventQueue`).

```
@Override
public void run() {
    EventQueue queue = vm.eventQueue();
    while (connected) {
        try {
            EventSet eventSet = queue.remove();
            EventIterator it = eventSet.eventIterator();
            while (it.hasNext()) {
                handleEvent(it.nextEvent());
            }
            eventSet.resume();
```

Figure 3.3: Snippet of the method run in class EventThread

The incoming events are always grouped in `EventSets`. To handle the events we iterate through each event in an `EventSet`. After all events have been handled, the `EventSet` is resumed, since events cause the target VM to suspend. If this is not done the target VM will hang. After the resumption, we proceed with the next `EventSet`.

Out of all possible types of events we are only interested in events of type `MethodExitEvents`. Once we get these events, we have to further filter them as we get an even for every call to a method of the class `java.lang.System`. However, we are only interested in methods that work with System Properties.

The data from the filtered events is then stored in a `PropertiesModel`, at which we will look in Chapter 3.2. In the class `PropertiesModelThreadTrace` the Analyser creates a new `PropertiesModel` and a new `CallSiteBuilder`.

In the method `methodExitEvent(..)` of the `PropertiesModelThreadTrace` we determine the location of a call and also the name of the called property. We use then the model to get the corresponding `Property` object. If this object does not yet exist, the model will create a new one.

The task of the `CallSiteBuilder` is to extract the necessary data from the event (code location, property value) and store it into a property object. This procedure will be clearer after reading Chapter 3.2.

```
Property property = this.model.getProperty(name);
callSiteBuilder.addEvent(event, location, property);
```

Figure 3.4: Snippet of the method methodExitEvent in class PropertiesModelThreadTrace

3.2 The Properties Model

As we saw above, the `PropertiesModel` class handles the creation of `Property` objects. Before creating a new `Property` object, the `PropertiesModel` looks for a property with the same name. This check is necessary to avoid having multiple `Property` objects referring to the same property. Thus, the model ensures that each called property has a unique `Property` object that stores data about its usage.

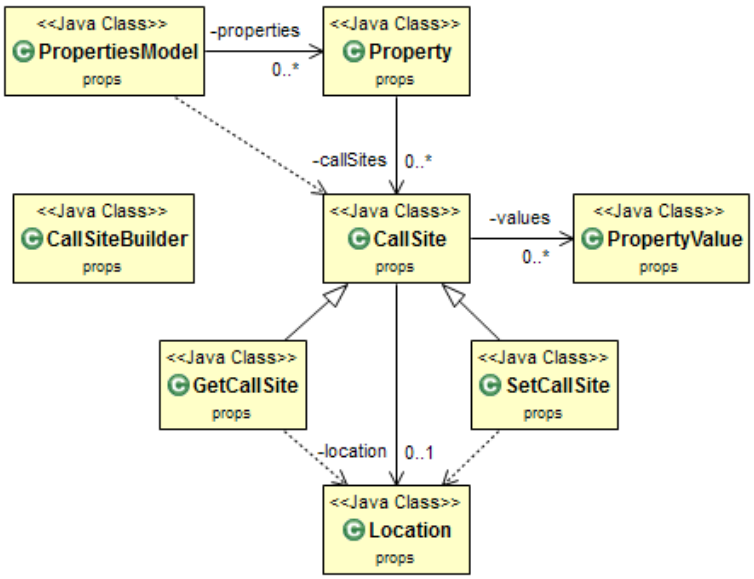

Figure 3.5: UML diagram of Properties Model

Information about the usage of a property is stored using a list of `CallSite` objects. A `CallSite` encapsulates a code location that uses a particular property, along with all the values the property had at that location. The location is captured using a `Location` object; the values using a list of `PropertyValue` objects. A list is required as the same call to a property can return different values at various points in time. To preserve the chronological order of those different values each `PropertyValue` object stores, alongside the value, an `id`, uniquely identifying that value. Its value comes from a static integer variable `id` in the class `CallSite`. Each time a `PropertyValue` has to be added to a `CallSite`, the `id` is increased and given to that `PropertyValue`. This means the order, in which the values occurred in the program, can be obtained by sorting the values based on the id.

The class `CallSiteBuilder` uses the functionality previously described to add the extracted data from the event to a `CallSite` and a `Property`. This class also determines the type of a call (get or set) and creates a call site of the corresponding type (`GetCallSite` or `SetCallSite`).

```
// Type getProperty
returnValue = event.returnValue().toString();
// Type setProperty
setValue = event.thread().frame(0).getArgumentValues().get(1).toString();
```

Figure 3.6: How the CallSiteBuilder gets the value for the PropertyValue object

```
PropertyValue value = new PropertyValue(returnValue);

CallSite callSite = property.locateCallSite(methodName, location);
if (callSite == null) {
    callSite = new GetCallSite(location); // or new SetCallSite(location)
    property.addCallSite(callSite);
}
callSite.addValue(value);
```

Figure 3.7: How the CallSiteBuilder adds the extracted data to a `CallSite`

To work with the elements of a properties model we are using the visitor design pattern. The advantage of this pattern is that we can add new operations on the model without having to modify its classes.

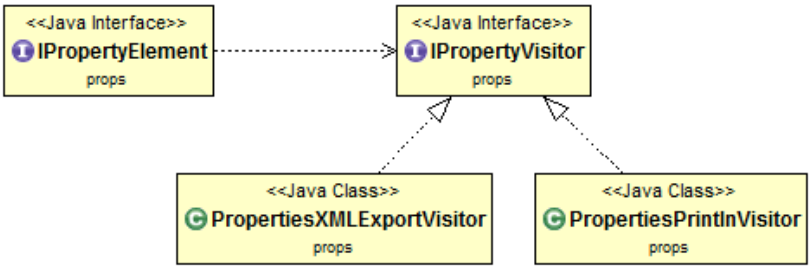

Figure 3.8: Visitor pattern of Properties Model

For our requirements we implemented two visitors. The first simply prints the model in the console. The second visitor, the `PropertiesXMLExportVisitor`, exports the model to an XML file. To implement this functionality we use the Java Architecture for XML Binding (JAXB). With JAXB we can easily map objects to an XML representation. Since we only need a simple export/import of Java objects into and from XML, we are using directly the class `JAXB` of the package `javax.xml.bind`, which defines the methods `marshal(..)` and `unmarshal(..)`, for creating an XML representation for an object, or recreating an object from its XML representation.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<propertiesModel>
    <property>
        <call>
            <location>
                <class>org.apache.jmeter.NewDriver</class>
                <codeindex>2</codeindex>
                <linenumber>40</linenumber>
                <method>&lt;clinit&gt;</method>
                <sourcepath>org\apache\jmeter\NewDriver.jav
            </location>
            <type>props.GetCallSite</type>
            <values>
                <count>1</count>
                <id>34</id>
                <value>;</value>
            </values>
        </call>
        <call>
            <location>
```

Figure 3.9: Example of an xml file created by JAXB.

In JAXB, the root and the elements of the model are defined using annotations. The root class of the model, in our case the `PropertiesModel`, needs the annotation `@XmlRootElement`. The other elements of the model, for example the `Property` objects, need the `@XmlElement(name = "elementName")` annotation. These annotations have to be placed on the *get* methods returning those elements. For example, the `PropertiesModel` object has a list of `Property` objects. This is accessed through the method `getProperties` having the annotation `@XmlElement(name = "property")`. In this way we indicate to JAXB that the nodes of the properties model, which is the tree root, are property elements. Apart from these annotations all classes used by JAXB have to define a constructor with no parameters.

```
@XmlElement(name = "property")
public List<Property> getProperties() {
    return this.properties;
}
```

Figure 3.10: Declaration of XmlElement property

4 Implementation of Properties Visualizer

In this chapter we look at the implementation of the Properties Visualizer. Before we do this, we give a general description about the implementation of Eclipse plugins.

4.1 Implementing an Eclipse plugin

The easiest way to start implementing an Eclipse plugin is to begin with one of the sample plugins provided by Eclipse.

A new plugin project consists of the usual folders, `src` and `bin`, but had also two additional folders `icons` and `META-INF`. The `META-INF` contains the manifest file, `MANIFEST.MF`, used to define some important information about the plugin, like its dependencies and extensions. Further information about the extensions is stored in a file named `plugin.xml` in the root folder.

The code of the plugin itself consists of at least one class. This class is set as the activator in the `MANIFEST.MF` file and controls the plugin life cycle. The activator extends the abstract subclass `AbstractUIPlugin` of the class `Plugin`, which implements the `BundleActivator` interface from the OSGi™ Framework (see Figure 4.1).

The OSGi™ Framework is a service platform that allows Eclipse to start, stop or update modules or components without requiring a reboot: it defines a dynamic component system for Java.

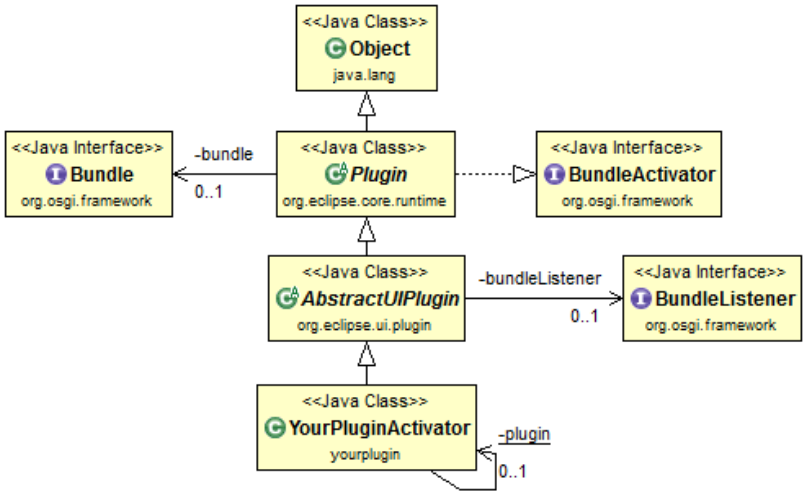

Figure 4.1: Relation of a plugin to Eclipse and the OSGi™ Framework

The `AbstractUIPlugin` class is the *"abstract base class for plug-ins that integrate with the Eclipse platform UI"*. This class provides services for managing UI resources, for example the `ImageRegistry` containing information about various images in the folder `icons`, of the plugin. Furthermore, subclasses of `AbstractUIPlugin` are also able to handle preferences and dialog windows (see Eclipse API[1]).

[1] http://help.eclipse.org/kepler/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/plugin/AbstractUIPlugin.html

4.2 The Properties Visualizer

The Properties Visualizer has several dependencies to other Eclipse plugins. The most important one is to the Properties Analyser plugin, as this plugin provides the functionality for recreating a Properties Model from an XML file.

When starting the Properties Visualizer, the activator initializes the `ImageRegistry` with the icons used by its views. The plugin creates then the main view.

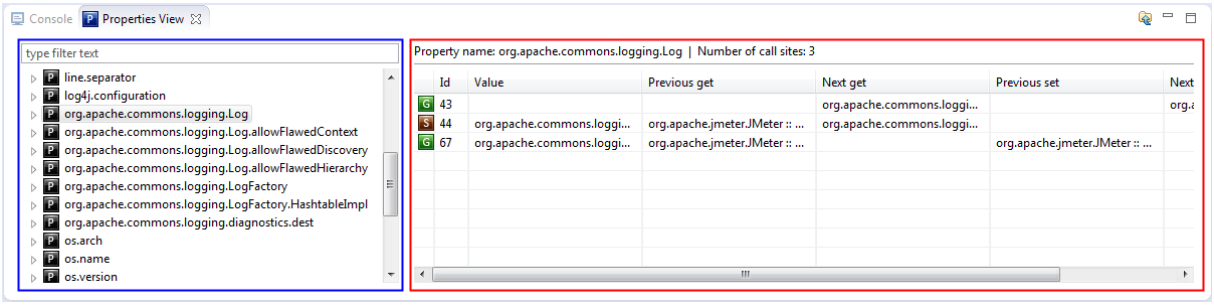

Figure 4.2: The view of the Properties Visualizer

The main view of the Properties Visualizer is composed of two parallel composites. The left composite shows all `Property` objects along with their `CallSite` objects. The right composite shows the values of the selected `CallSite` or `Property`.

Each class of the Properties Visualizer, except the `Utils` class, extends a class or implements an interface from Eclipse. The central part of the implementation is the class `PropertiesViewPart`. This class initializes the two composites and provides the displayed data.

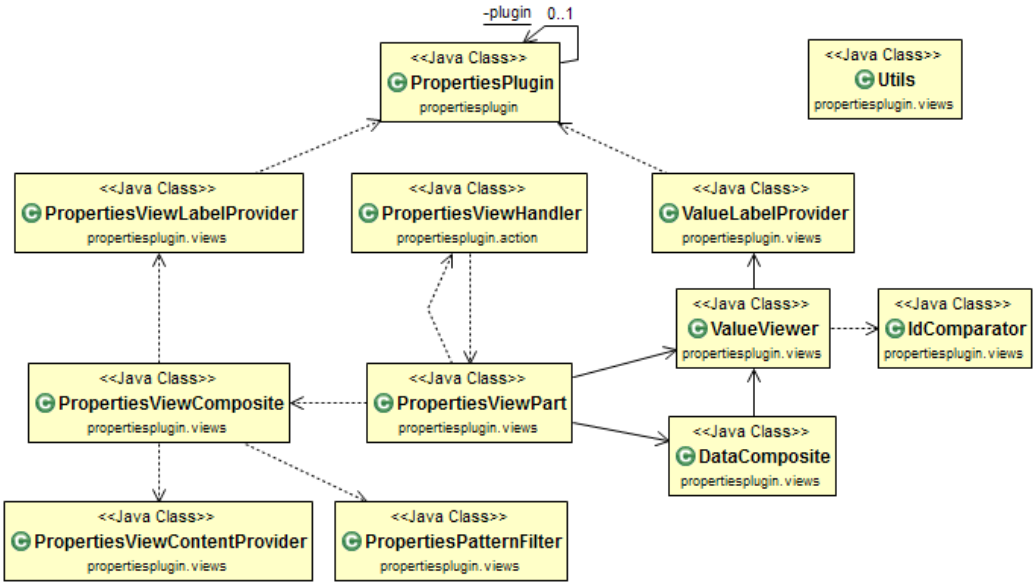

Figure 4.3: UML diagram of the PropertiesPlugin

Both composites, the `PropertiesViewComposite` and the `DataComposite`, extend the class `Composite`. Instances of this class are containers, which are capable of storing other containers, where a container is the abstract superclass of all window components in Eclipse.

4.2.1 PropertiesViewComposite

For the `PropertiesViewComposite` we are using a `FilteredTree`. This class is a simple composite that provides a text widget (for the filter input) and a `TreeViewer`. It is constructed with a given `PatternFilter`, which does a pattern matching on the tree's children. Since we need a slightly different behaviour, we implemented the class `PropertiesPatternFilter`, which extends the `PatternFilter` and overrides the methods `isParentMatch(..)` and `isLeafMatch(..)`.

The `PropertiesViewContentProvider` is responsible, as the name suggests, for the content of the `PropertiesViewComposite`. In the method `getElements(..)` of this class a new `XMLImporter` is created and used to import the `PropertiesModel` from the XML file. The `Property` objects from the model are then returned in an `Object[]` array. Each object in this array (properties) and its children (call-sites) are displayed by the methods `getText(..)` and `getImage(..)` from the `PropertiesViewLabelProvider`, which extends the class `ColumnLabelProvider` from Eclipse.

4.2.2 DataComposite

The viewer for the `DataComposite`, the `ValueViewer`, extends the class `TableViewer` from Eclipse. The `ValueViewer` sets up the table including the columns. Furthermore it implements the functionality for double-clicking on a table item.

For the content we use an `ArrayContentProvider`. This provider is used for handling a collection of elements. The elements are then displayed with the `ValueLabelProvider` through its methods `getColumnText(..)` and `getColumnImage(..)`. This class extends the `ColumnLabelProvider` and also implements the interface `ITableLabelProvider` by overriding the two mentioned methods. Since we are using the same label provider for every column, we have to implement all the different cases for the column index (*e. g.* column "Value" or column "Previous get").

4.2.3 Other functionality

The `PropertiesViewPart` is also responsible for updating the content of the `DataComposite`. To fulfil this task it implements the interface `ISelectionChangedListener` and adds itself to the viewer of the `PropertiesViewComposite`. This listener is then used to update the viewer of the `DataComposite` with the values of the selected `CallSite` or `Property` from the properties tree.

To sort our input by id in both viewers we are using the class `IdComparator`, which extends the class `ViewerComparator` from Eclipse. The task of this class is, on the on hand, to sort the properties by name and on the other hand, to sort the values of the call-sites by the id.

For the tooltips of the viewer's we are overriding the corresponding methods from the extended class `ColumnLabelProvider` in each label provider.

Last but not least, the `Utils` class contains various methods that are often used by the other classes. These methods should always do the same so all attributes and methods are defined as static. This is the common way to implement such a class. In the `Utils` class are, for example, methods to navigate

through the data tree, like from a `PropertyValue` to the containing `CallSite` and from this to the containing `Property`, but also methods to open the source file or to highlight a predefined line.

For launching the Analyser and importing the data into the plugin, we used an extension point to the right-click-menu of Eclipse and implemented an action handler.

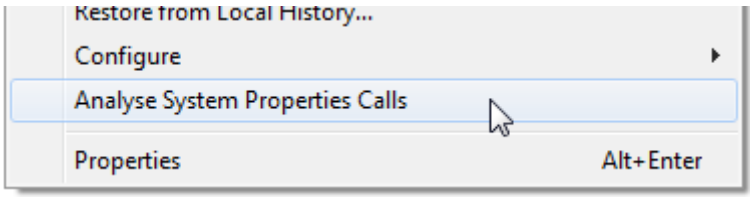

Figure 4.4: Bottom of the right-click menu of a Java project

5 References

Oracle. (2013, 11 30). *Interface VirtualMachineManager*. Retrieved from JDI API Specification: http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/com/sun/jdi/VirtualMachineManager.html

Oracle. (2013, 09 05). *Java™ Debug Interface (JDI)*. Retrieved from Oracle Java™ SE 7 Documentation: http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html

Oracle. (2013, 11 30). *Java™ Debug Interface (JDI) API*. Retrieved from http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html

Oracle. (2013, 09 05). *Java™ Debug Wire Protocol Transport Interface (JDWP)*. Retrieved from Oracle Java™ SE 7 Documentation: http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwpTransport.html

Oracle. (2013, 09 05). *Java™ Platform Debugger Architecture (JPDA)*. Retrieved 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