MASTER THESIS

Jakub Jelínek

Output breakpoints for Java

Department of Distributed and Dependable Systems

Supervisor of the master thesis:  RNDr. Jan Kofroň, Ph.D.
Study programme:  Informatics
Specialization:  Software Systems

Prague 2012
I would like to thank RNDr. Jan Kofroň for being the supervisor of this thesis, for his guidance and valuable feedback during the time this thesis was formed.

I would like to thank my wife Eva for her patience, help and love that she manifested to me so many times during my whole study and also during work on this thesis.
I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on March 21, 2012

Jakub Jelínek
Název práce: Output breakpoints for Java

Autor: Jakub Jelínek

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Jan Kofroň, Ph.D.

Abstrakt:
Cílem této práce bylo navrhnout podporu výstupních breakpointů pro jazyk Java a implementovat navržené řešení včetně integrace do prostředí Eclipse IDE. Výstupní breakpoint představuje místo na výstupním médiu (např. soubor, standardní výstup nebo socket), kde se má vykonávání programu zastavit. Místo na výstupním médiu je zobecněno jako zastavovací podmínka – může představovat např. absolutní pozici, konkrétní řádek a znak nebo výskyt textového řetězce.

Navržené řešení je založeno na pozorování tříd odpovědných za výstup pomocí vstupních breakpointů na metodách, které jsou standardní součástí jazyka Java. Řešení je závislé na konkrétní verzi a implementaci JRE, neboť pozoruje také metody, které nejsou součástí veřejného Java API.

Klíčová slova: Java, breakpoint, výstupní soubor, debugger, Eclipse

Title: Output breakpoints for Java

Author: Jakub Jelínek

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kofroň, Ph.D.

Abstract:
The goal of this thesis was to design the support for output breakpoints in the Java language and implement the designed solution including integration into the Eclipse IDE. An output breakpoint represents a location in an output media (e.g. file, standard output or socket) where the execution of program should stop. The location in the output media is generalized as a stop condition – it can be in the form of absolute position, a particular line and a character or an occurrence of a text string.

The designed solution is based on observing the classes responsible for output using entry method breakpoints that are a standard part of the Java language. This solution is dependent on a particular version and implementation of JRE because some of observed methods are not a part of public Java API.

Keywords: Java, breakpoint, output file, debugger, Eclipse
# Contents

## Introduction 3

## 1 Background 6

1.1 Java Platform Debugger Architecture .......................... 6
   1.1.1 Java Debug Interface .................................. 7
   1.1.2 Java Debug Wire Protocol .............................. 8
   1.1.3 Java Virtual Machine Tools Interface ................... 8

1.2 Eclipse IDE .................................................. 10
   1.2.1 Eclipse Platform architecture .......................... 11
   1.2.2 Eclipse Debug Project ................................ 11

## 2 Solution design 13

2.1 Terms ......................................................... 13

2.2 Choice of solution ........................................... 13
   2.2.1 Modification of the JPDA ............................... 13
   2.2.2 Modification of debugged program ...................... 14
   2.2.3 Debugger using the JPDA only .......................... 15
   2.2.4 Overview and selection ................................ 16

2.3 Basic concept ................................................ 16
   2.3.1 Obtaining method argument values ........................ 17
   2.3.2 JRE modification ........................................ 19
   2.3.3 Output breakpoint life cycle ............................ 20

2.4 IDE selection .................................................. 22

2.5 Limitations ................................................... 22

## 3 Development documentation 23

3.1 Overview ....................................................... 23

3.2 Plugin cz.cuni.mff.kuba.breakout .............................. 24
   3.2.1 Eclipse integration ..................................... 24
   3.2.2 OutputGate interface ................................... 28
   3.2.3 StopCondition interface ................................. 29
   3.2.4 Extension points ....................................... 30
   3.2.5 Configuration composites ............................... 30
3.3 Plugin cz.cuni.mff.kuba.breakout.gate ................................. 31
  3.3.1 FileOutputStreamGate ........................................... 32
  3.3.2 SystemOutputGate .............................................. 33
  3.3.3 SocketOutputStreamGate ........................................ 34
  3.3.4 FileChannelGate ............................................... 35
3.4 Plugin cz.cuni.mff.kuba.breakout.condition ....................... 37
  3.4.1 Encoding ....................................................... 39
3.5 Plugin cz.cuni.mff.kuba.breakout.extra ............................ 40
3.6 Known bugs ......................................................... 40
3.7 Extensibility ....................................................... 41
  3.7.1 Creating an output gate ....................................... 41
  3.7.2 Creating a stop condition .................................... 42

4 Evaluation ............................................................. 43
  4.1 Performance ....................................................... 44
  4.2 Future work ....................................................... 46

Conclusion ............................................................... 47

Bibliography ............................................................ 48

List of Figures .......................................................... 50

List of Abbreviations .................................................... 51

Appendix A: Content of attached CD .................................. 52

Appendix B: User manual ................................................ 53
Introduction

Software developers spend a lot of time finding bugs. When debugging an application, it is often quite easy to identify the flaw in the application output. For example, the sixth line does not contain a bracket. However, it may not be easy to locate the particular location in the application code which is responsible for the erroneous output, so that one is clueless where to place the breakpoint to trace the code, find the reason, and fix the bug.

This thesis aims at solving this issue when using the Java language by introducing output breakpoints that could help in situations described above.

The need for output breakpoints increases when there is a program where a particular part of the output can be produced by multiple places in the source code. For example, a compiler produces a sequence of instructions. A single instruction (e.g. an unconditional jump) can be a part of many constructions (e.g. for cycle, switch or if-then-else). Without output breakpoints, one would have to install a line breakpoint on each source code line that causes the unconditional jump instruction in the output. However, with output breakpoints support, one adds just one output breakpoint.

Term of output breakpoint

The concept of output breakpoints extends the well-established notion of breakpoints by possibility to associate a breakpoint with a location in the program output. In the case of line breakpoint the location is obviously defined as the line number. However, in the case of program output it would be desired to allow expressing the location more heterogeneously.

In the most general case, the location can be determined by a regular expression. The program would stop when the entire output would match this regular expression. Note that using only regular expressions might not be user-friendly. Consider a location defined as “stop on 12th line, 25th character” expressed by the regular expression \((\^[\^n]\n\}\{11\}\^[\^n]\}\{25\}\). Therefore, specific types of stop conditions will be provided.
Also, different ways of program output are considered. Beside the standard output, there is the standard error output, output to files or outgoing network communication. All these ways shall be discussed and supported.

**Goals**

There are two main goals of the thesis that can be summarised as follows:

- To design an approach to detect writing into program output and be able to stop program executing on user code that caused this write. This approach should be a natural extension of current debugging means provided by the Java platform.

- To implement an extension to Eclipse IDE (that was chosen as a representative of mainstream Java IDEs) that would provide output breakpoints functionality. It should be usable in a simple way and should allow to inspect the program state as for other types of breakpoints.

These two main goals can be refined to few sub-goals and emphases that are also important and should be kept in mind when finding the solution:

- Provide support for the following types of output: standard (error) output, output to files and outgoing network communication.

- Proposed solution should not expect the user to modify the code in order to use the output breakpoints.

- If technically possible, the Eclipse extension should be realised as (a) separate plugin(s) that extend(s) only extension points provided by other plugins. It means that no modification of standard plugins responsible for Java debugging support in Eclipse would be needed.

- The Eclipse extension should be extensible itself, allowing to add a new type of output or a new type of stop condition (location definition).
Text structure

**Chapter 1** gives an overview about background technologies that are used or good to know for this solution. Mainly it covers the Eclipse platform, focusing on the debugger support of this platform, and finally the debugging platform provided by Java language.

**Chapter 2** explains principles of the provided solution, reasons for design decisions and principles that are important to understand the chosen solution. It aims more at the idea of output breakpoints itself while postponing the discussion of the integration into the Eclipse IDE to the following section.

**Chapter 3** contains the developer documentation. It covers implementation details, including the way this solution was integrated into the Eclipse IDE and describes the problems that raised up.

**Chapter 4** sums up this thesis, reviews conclusions and achieved results. It also recalls limitations of this solution and propose possible future work in this field. There is also included a report about performance testing in this chapter.

**Appendix A** describes folder structure of attached CD.

**Appendix B** contains user manual. Describes how to install plugins supporting output breakpoints into Eclipse and how to use them.
1. Background

In this chapter, it will be given an overview about background technologies that are essentials for understanding the solution proposed later. First, the Java Platform Debugger Architecture will be introduced. Second, it will be explored the Eclipse Platform and its debugging support, especially focusing on support for Java language debugging.

1.1 Java Platform Debugger Architecture

“The Java Platform Debugger Architecture (JPDA) provides the infrastructure you need to build end-user debugger applications for the Java Platform, Standard Edition (Java SE). It includes the following three-layered APIs:

- Java Debug Interface (JDI), a high-level Java programming language interface, including support for remote debugging;
- Java Debug Wire Protocol (JDWP), which defines the format of information and requests transferred between the process being debugged and the debugger front end;
- The JVM(TM) Tools Interface, JVM TI. This is a low-level native interface that defines the services a Java virtual machine provides for tools such as debuggers and profilers.” [9]

The official Oracle documentation [10] gives following recommendation concerning a debugger development: “A debugger developer may hook into JPDA at any layer. Since the JDI is the highest level and easiest to use we encourage developers to use this interface. Suppose a company develops a debugger using JDI. They can use it with the reference implementation and it will automatically work with the VMs and platforms Sun supports. It can also work, for example, with the reference implementation front-end and a debuggee running another company’s VM that implements JDWP (which might use or by-pass JVM TI).

Some debuggers are built on top of lower layers, JDWP (for example if the front-end is not written in the Java language) or JVM TI (for specialized debuggers which need low-level functionality).”
To sum up, the Figure 1.1 gives an overview about the possibilities a developer can hook into JPDA. The debugger A represents use of the high-level JDI layer. The debugger B represents use of JDWP layer. The debugger C stands for a debugger using the JVM TI only. The variant A is preferred and can be used because the Eclipse IDE is written in Java.

1.1.1 Java Debug Interface

The Java Debug Interface (JDI) is a high-level Java API that enables a debugger to connect to a virtual machine, to set up breakpoints and to receive breakpoints events, including access to the running state of the virtual machine.

To get an idea about the JDI, its packages will be shortly described. The JDI API consists of five packages as follows:

- **com.sun.jdi** – defines mirrors (classes that represents objects in the target VM) for values and types (primitive, arrays and objects/classes), and for the target virtual machine, threads and stack frames. These classes provides the access to the running state of the virtual machine, including getting and setting values of visible variables.

- **com.sun.jdi.connect** – provides connectors that can launch a virtual machine for debugging (called launching connectors) or those that can connect to a running instance of VM (called attaching connectors).

- **com.sun.jdi.connect.spi** – intended to for developers of a new transport service for connections between a debugger and a target VM.
• **com.sun.jdi.request** – contains the `EventRequestManager` class that allows to create breakpoint requests and classes that represent different types of these requests. An instance of the event request manager can be obtained from the `VirtualMachine` class.

• **com.sun.jdi.event** – contains the `EventQueue` class that stores incoming breakpoint events and classes that represents events. For each request class exists a corresponding event class, for example `ExceptionRequest` – `ExceptionEvent`.

There are listed all JDI events in the Table 1.1. Descriptions are taken from its JavaDoc [7]. Note that for the later proposed solution the **method entry event** will be used.

### 1.1.2 Java Debug Wire Protocol

The Java Debug Wire Protocol (JDWP) is the protocol used for communication between a debugger and the debugged Java virtual machine. This protocol does not define transport, but only format and layout of messages. Therefore, it allows different ways of transport (e.g. sockets or shared memory). For more information about transports see [14], for protocol description see [8].

### 1.1.3 Java Virtual Machine Tools Interface

“The JVM tool interface (JVM TI) is a native programming interface for use by tools. It provides both a way to inspect the state and to control the execution of applications running in the Java virtual machine (JVM). JVM TI supports the full breadth of tools that need access to JVM state, including but not limited to: profiling, debugging, monitoring, thread analysis, and coverage analysis tools.

JVM TI was introduced at JDK 5.0. JVM TI replaces the Java Virtual Machine Profiler Interface (JVMPI) and the Java Virtual Machine Debug Interface (JVMDI) which, as of JDK 6, are no longer provided.” [13]
<table>
<thead>
<tr>
<th>Event class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessWatchpointEvent</td>
<td>Notification of a field access in the target VM. Field modifications are not considered field accesses.</td>
</tr>
<tr>
<td>ModificationWatchpointEvent</td>
<td>Notification of a field modification in the target VM.</td>
</tr>
<tr>
<td>WatchpointEvent</td>
<td>Notification of a field access or modification.</td>
</tr>
<tr>
<td>ClassPrepareEvent</td>
<td>Notification of a class prepare in the target VM.</td>
</tr>
<tr>
<td>ClassUnloadEvent</td>
<td>Notification of a class unload in the target VM.</td>
</tr>
<tr>
<td>BreakpointEvent</td>
<td>Notification of a breakpoint in the target VM. This is the “line” breakpoint.</td>
</tr>
<tr>
<td>ExceptionEvent</td>
<td>Notification of an exception in the target VM.</td>
</tr>
<tr>
<td>MethodEntryEvent</td>
<td>Notification of a method invocation in the target VM. This event occurs after entry into the invoked method and before any code has executed. Method entry events are generated for both native and non-native methods.</td>
</tr>
<tr>
<td>MethodExitEvent</td>
<td>Notification of a method return in the target VM. This event is generated after all code in the method has executed, but the location of this event is the last executed location in the method. Method exit events are generated for both native and non-native methods. Method exit events are not generated if the method terminates with a thrown exception.</td>
</tr>
<tr>
<td>MonitorContendedEnteredEvent</td>
<td>Notification that a thread in the target VM is entering a monitor after waiting for it to be released by another thread.</td>
</tr>
<tr>
<td>MonitorContendedEnterEvent</td>
<td>Notification that a thread in the target VM is attempting to enter a monitor that is already acquired by another thread.</td>
</tr>
<tr>
<td>MonitorWaitedEvent</td>
<td>Notification that a thread in the target VM has finished waiting on a monitor object.</td>
</tr>
<tr>
<td>MonitorWaitEvent</td>
<td>Notification that a thread in the target VM is about to wait on a monitor object.</td>
</tr>
<tr>
<td>StepEvent</td>
<td>Notification of step completion in the target VM.</td>
</tr>
<tr>
<td>ThreadDeathEvent</td>
<td>Notification of a completed thread in the target VM.</td>
</tr>
<tr>
<td>ThreadStartEvent</td>
<td>Notification of a new running thread in the target VM.</td>
</tr>
<tr>
<td>VMStartEvent</td>
<td>Notification of initialization of a target VM.</td>
</tr>
<tr>
<td>VMDepthEvent</td>
<td>Notification of target VM termination.</td>
</tr>
<tr>
<td>VMDisconnectEvent</td>
<td>Notification of disconnection from target VM.</td>
</tr>
</tbody>
</table>

Table 1.1: JDI events overview

9
1.2 Eclipse IDE

Since the Eclipse IDE was chosen to be enriched by output breakpoints support, it will be introduced in this part, starting with general description and then focusing on the support for the Java language debugging. For reasons that led to the selection of Eclipse IDE see chapter 2.4 IDE selection.

In fact, the term “Eclipse IDE” is not exact. We use this term here to refer to the Eclipse SDK, therefore, to be precise, let us define:

- “The **Eclipse Platform** contains the functionality required to build an IDE. However, the Eclipse Platform is itself a composition of components; by using a subset of these components, it is possible to build arbitrary applications. The Eclipse Rich Client Platform (RCP) is one such subset of components ... The RCP is being used to build arbitrary applications that have nothing to do with software development in diverse areas that include banking, automotive, medical, and space exploration.” [6]

- The **Java development tools** (JDT) project “provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself.” [2]

![Figure 1.2: Eclipse SDK (taken from [4])](image)
• The **Eclipse SDK** “is a combination of the efforts of several Eclipse projects, including Platform, Java Development Tools (JDT), and the Plug-in Development Environment (PDE).” [6] The Figure 1.2 shows the structure of the Eclipse SDK.

To sum up, we are going to extend the Java development tools (JDT) that is a part of the Eclipse SDK.

### 1.2.1 Eclipse Platform architecture

“The Eclipse platform is structured around the concept of *plug-ins*. Plug-ins are structured bundles of code and/or data that contribute functionality to the system. Functionality can be contributed in the form of code libraries (Java classes with public API), platform *extensions*, or even documentation. Plug-ins can define *extension points*, well-defined places where other plug-ins can add functionality.” [4]

### 1.2.2 Eclipse Debug Project

Different plugins for Eclipse platform are developed under Eclipse projects that deal with different areas of interests – for full list of projects see [3].

There is one interesting project in view of Java debugging – the Eclipse Debug Project. In fact, this project consists of two sub-projects. First of them is Platform Debug project that offers general, language independent support for debugging. The second one is JDT Debug project that provides Java specific extension based on the Platform Debug means.

**Platform Debug**

The Platform Debug project offers language independent support for debugging that includes mechanisms for:

• “Launching programs
• Source lookup
• Defining and registering breakpoints
• Event notification from programs being debugged
• A language independent debug model
• A language independent debug UI

The Debug component of the platform defines interfaces for a language independent debug model, which abstract common debugging features of many languages. The Debug component of the platform does not provide an implementation of a debugger, it is the duty of other plug-ins to provide language specific implementations of debuggers.” [1]

In the case that someone would like to write a brand new debugger based on the Platform Debug, or for an overview what the Platform Debug offers and how it can be used, reading of the article [16] is recommended.

**JDT Debug**

“JDT Debug implements Java debugging support and works with any JDPA-compliant target Java VM. It is implemented on top of the language independent debug model provided by the platform debugger.

JDT debug provides the following debugging functionality:

• Launching of a Java VM in either run or debug mode
• Attaching to a running Java VM
• Expression evaluation in the context of a stack frame
• Scrapbook pages for interactive Java code snippet evaluation
• Dynamic class reloading where supported by Java virtual machine” [1]
2. Solution design

This chapter describes the proposed solution of output breakpoint support. After defining a few important terms, there are explained possible approaches to solve this problem and also explained which approach was chosen and why. The chosen solution is then described in detail from the basic idea to several technical details. There are also revealed limitations of this solution.

2.1 Terms

It will be useful to define two terms used throughout this chapter. These terms describe two parts that each output breakpoint is made from:

- **Output gate** represents kind of output medium, a place where the output goes through. For example, a file is a kind of the output gate. A concrete file name is then configuration of this output gate. This concrete file is an instance of the file output gate.

- **Stop condition** represents the generalized location in the output where a program should interrupt. Different stop conditions can offer different means to express the location. Note that there can be a stop condition that can cause stopping the program more that once (e.g. whenever you see "ERROR" string).

2.2 Choice of solution

It was identified three ways to support output breakpoints. Each of them will be described and its advantages and disadvantages will be shown. The main question concerning the solution is how to observe the output.

2.2.1 Modification of the JPDA

The first approach is to reflect the concept of output breakpoints through the entire Java Platform Debugging Architecture. On the JDI layer the EventRe-
questManager interface would be extended by a method `createOutputRequest` that creates a output breakpoint request. Also, related request (`OutputRequest`) and event (`OutputEvent`) classes would be created. The `OutputRequest` class would have appropriate methods to define an output gate and a stop condition.

Similarly, the Java Debug Wire Protocol (JDWP) would be extended by commands allowing to express output breakpoint requests, its configuration and also commands to indicate breakpoint hit. Also the JVM Tool Interface would be extended by methods concerning output breakpoints. Here, in the target VM would also be realised the observation itself (somehow).

Advantages of this approach are small requirements for debugger clients, small traffic between the debugger and the deuggee – importance of this increases when debugging remotely.

As a main disadvantage we can see is wide range of necessary changes that have to be made in several API layers. Also, it would require to modify each JVM implementation to support this feature. Therefore, this seems to be a cumbersome solution unless the Java defining authority would decide to officially change the API in the way of supporting output breakpoints.

### 2.2.2 Modification of debugged program

In general, this approach consist of modification the debugged program so that the program itself would detect the moment when to suspend. This modification can be done as preprocessing before compilation, or as bytecode modification after the compilation.

Since all classes that are needed to be modified (those that are responsible for output writing) are a part of the JRE, they could be modified manually, once compiled and then used as part of the JRE.

For example, the `write(byte[])` method from the `FileOutputStream` class would be modified like this:

```java
public void write(byte b[]) throws IOException {
    OutputBreakpointSupport.processWrite(b);
    writeBytes(b, 0, b.length); // the original method code
}
```
Then, in the `OutputBreakpointSupport` class would be a line where the program reaches only when the stop condition is fulfilled. The only thing that a debugger client would have to do is to install one line breakpoint on this line that indicates stop condition meeting.

For the bytecode modification the AspectJ could be used. The AspectJ is a aspect-oriented extension to the Java programming language. An aspect can add a source code before, after or around defined method. Then, using the AspectJ compiler can be aspects added to a JAR like this (note that this adding is called weaving):

```sh
ajc -inpath rt.jar -outjar rt_woven.jar MyAspect.java
```

Advantages of this approach are small requirements for debugger clients (one like breakpoint on a predetermined location indicating fulfilling the stop condition) a small traffic between the debugger and the debugee (only when hit).

### 2.2.3 Debugger using the JPDA only

This approach is based on using abilities of JPDA only and does not add nothing to the debugged program nor to the target JVM. Conditional breakpoints known in Eclipse (and other IDEs as well) do the same – they use line breakpoints that are offered by JPDA and enrich them by evaluating a condition. When this condition is fulfilled, debugger suspends the program. Otherwise, the debugger resumes the program so that for user it seems that the program has not stopped at all.

For output breakpoint support, method entry breakpoints can be used. These method entry (and exit) breakpoints already are a part of JPDA. When a method entry breakpoint is hit, the debugger can inspect values of method arguments. Based on these values, the debugger decides whether to suspend the program or to resume it.

The greatest disadvantage of this solution is frequent communication between debugger and debugee since debugee waits for debugger decision during each write.
2.2.4 Overview and selection

Table 2.1 gives an overview of possible solutions and includes information about their requirements concerning debugger-debugee traffic and changes needed to be done with JRE or JVM.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Traffic</th>
<th>JRE changes</th>
<th>JVM changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modification of the JPDA</td>
<td>small</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Modification of debugged program</td>
<td>small</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Debugger using the JPDA only</td>
<td>large</td>
<td>rarely</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 2.1: Solution variants overview

The first alternative that modified the JPDA is really cumbersome since it requires a lot of modification. The other two solutions are usable. Since the third alternative has less modification needs, it was chosen to be explored and implemented. It will be described in detail in following sections.

However, after doing performance testing (see section 4.1 Performance) one should also pay attention to the second alternative in future because it probably could have better performance.

2.3 Basic concept

The proposed approach is based on observation that there is usually one or few classes from Java runtime library (rt.jar) that are responsible for writing into particular output.

Let us imagine that there is an output medium and the class ResponsibleClass that is responsible for writing into this output medium. Let assume further that this class has two methods open(String name) and write(byte b). Once the open method is called, output medium with name name is opened and ready to be written into. Also, the instance of ResponsibleClass where the open method was called is associated with the specified output. All following writes are done by calling the write method on this instance:

```java
ResponsibleClass output = new ResponsibleClass();
output.open("/output/file.txt");
output.write('A');
```
The Java Platform Debugging Architecture (see [9]) provides method entry breakpoints that are used to observe methods that are called. Thus, by installing a method entry breakpoint on `ResponsibleClass` one can be informed whenever any method on this class is called (see Figure 2.1).

![Figure 2.1: Basic concept of output breakpoints](image)

First, we need to identify the correct instance of `ResponsibleClass`. That can be achieved by observing calls of the `open` method. By inspecting the value of the `name` method argument can be decided whether the name matches. If so, subsequent calls of the `write` method on this instance are also processed – by obtaining the parameter `b` value and passing this value to a stop condition that can decide about program interruption.

By this approach, program can be stopped even before it actually attempts to write on watched location so that user can identify who causes the output.

### 2.3.1 Obtaining method argument values

As described above, we observe certain methods just in time when they are called. We also need to obtain the actual values for method arguments. When processing a method entry event, there is an instance of the `MethodEntryEvent` class that represents this event (see Figure 2.2).

In this context, each method is identified by a `name` and `signature`. Note that this signature is a string encoded as defined in the JNI documentation [12].

Based on the name and signature we can identify the method. Now, there are two ways to obtain the method argument value. The value can be get by
argument name or argument position. Since the name is not always present (depends not debug level\textsuperscript{1} that was used when compiling the deuggeee), getting the value by position is preferred.

Figure 2.2: The MethodEntryEvent class

To get the value of argument on pos position from instance of MethodEntryEvent named event can be done in two ways. The first one uses the getArgumentValues method of the StackFrame class that is there since Java 1.6:

```java
ThreadReference thread = event.thread();
StackFrame frame = thread.frame(0);
Value value = frame.getArgumentValues().get(pos);
```

The second way uses the arguments method of the Method class and the getValue method of the StackFrame class:

```java
List args = mee.method().arguments();
LocalVariable var = (LocalVariable) args.get(pos);
Value value = mee.thread().frame(0).getValue(var);
```

In fact, the first mentioned method getArgumentValues uses internally typically the arguments method. Therefore, limitations of the second method will be inherited by the first one.

And indeed, there is a limitation for the arguments method that is described in its JavaDoc \cite{15}: "AbsentInformationException is thrown if there is no

\textsuperscript{1}Using the javac compiler, there is -g: option with possible values lines (for line numbers), vars (for variable names) and source (for full source code)
variable information for this method. Generally, local variable information is not available for native or abstract methods, thus they will throw this exception.”

Consequently, a trouble comes if a method that we would like to observe is native. This really happens in case of the java.io.FileOutputStream class and its write(int) method (in Java 6, in Java 7 is this method no longer native). It this case, Java runtime environment modification is to be done.

2.3.2 JRE modification

In the case of the write(int) method of the java.io.FileOutputStream class it is possible to declare this method as non-native and provide implementation that delegates this call to another also native method that can do its work. Originally declared method:

public native void write(int b) throws IOException;

is rewritten like this:

public void write(int b) throws IOException {
    byte[] array = { (byte) b };  
    writeBytes(array, 0, 1);
}

Note that the native method writeBytes(byte[], int, int) is called. This method is private and thus not part of (public) API. For output breakpoint only public API methods are interesting to observe because these methods clients have to call to get what they want (to write something in this case).

Now comes a question how to impart this modified class into debugee JRE. First option is to replace the class file directly in rt.jar. This is quite easy because JAR archives are just a renamed ZIP file.

Another way is to use java command option -Xbootclasspath/p:path that "specifies a semicolon-separated path of directories, JAR archives, and ZIP archives to prepend in front of the default bootstrap class path. Note: Applications that use this option for the purpose of overriding a class in rt.jar should not be deployed as doing so would contravene the Java 2 Runtime Environment binary code
license.” [11] When using Eclipse, you just add this option with correctly set up the path in Run Configurations > Arguments tab > VM arguments field. For example, when the patch class files are in /my/space/patch folder, following is to be entered:

- Xbootclasspath/p:/path/to/patch  (Unix-like path)
- Xbootclasspath/p:C:\\path\\to\\patch  (Windows-like path)

### 2.3.3 Output breakpoint life cycle

Each output breakpoint has a life cycle that is goes through. First of all, the user decides what output breakpoint to create – he chooses an output gate and a stop condition. Then, each launch of the debuggee has three phases:

- initialization
- waiting for opening the gate (looking for responsible instance)
- listening to written data (evaluating of stop condition)

![Figure 2.3: Initialization of output breakpoints](image)

Initialization should happen before the debuggee starts (debuggee is launched with -Xrunjdwp:suspend=y option). During this phase each output breakpoint has an opportunity to create one or more breakpoint requests. These requests typically observe methods responsible for opening the output medium. After
all output breakpoints have created needed breakpoints requests, target virtual machine is resumed. See Figure 2.3 for illustration.

When later a breakpoint (the installed method entry breakpoint, not the output breakpoint) is hit, the output breakpoint that has created corresponding breakpoint request is asked for handing this event. Output breakpoint reference is saved as custom property of event request. In general, there are two results of handling – (1) decision whether actually stop at this breakpoint or not and (2) newly created breakpoint requests.

Usually, breakpoint requests created during initialization phase are used to identify the instance of responsible class, then are created breakpoint requests to observe data written by this instance. See Figure 2.4 for illustration.

Figure 2.4: Handling breakpoint events

\(^{2}\)The `com.sun.jdi.request.EventRequest` class has the `getProperty(Object key)` and `putProperty(Object key, Object value)` methods to save custom user data.
2.4 IDE selection

In the beginning it was necessary to make a decision about the IDE where the support for output breakpoints should be implemented. There were two main candidates – Eclipse and NetBeans. Both are well established IDEs and both offer comparable possibility of extensibility by plugins.

Therefore, personal author’s experience and practice with these IDEs played a role. Since author’s preferred IDE is Eclipse, the Eclipse was chosen as the platform for output breakpoints support. But certainly a substantial part of this solution could be used also for extending the NetBeans IDE or any other debugging UI.

2.5 Limitations

In the ideal case, all observed methods would be a part of the public Java API and none of them would be native in any JRE implementation. In that case there would be no need for JRE modification (because of missing native methods) and this solution would be implementation independent.

However, this conditions are not met. Therefore, in the cases where private or protected methods or fields are used (either for observation or for getting a value) this solution becomes dependent on this non-public knowledge. Primarily, Oracle (former Sun) Java 6 and Java 7 are supported. In some cases, this solution can work also with other JRE implementations but with no warranty. JRE implementation dependencies will be discussed in detail in chapter 3 – Development documentation.

Taking into account the fact that provided Eclipse plugins depend on other plugins from Eclipse platform, this solution is also limited by Eclipse version that it is made for – Eclipse 3.7 Indigo. It also could work with another versions but with no warranty.
3. Development documentation

This chapter covers development documentation above scope of generated JavaDoc documentation that can be found on the attached CD in /doc/javadoc folder.

3.1 Overview

The provided solution is delivered as a set of Eclipse plugins. Therefore, this documentation will be divided into parts according these plugins. The solution is made up of four plugins (see Figure 3.1).

![Figure 3.1: Plugins dependencies](image)

The `cz.cuni.mff.kuba.breakout` plugin is the core plugin. It defines contracts for output gates and stop conditions (both Java interfaces and Eclipse extension points) and provides integration into Eclipse IDE.

The `cz.cuni.mff.kuba.breakout.gate` plugin extends the core plugin by providing output gates implementations. Note that this plugin is not dependent on any other plugin but the core plugin.

The `cz.cuni.mff.kuba.breakout.condition` plugin extends the core plugin by providing stop conditions implementations. Note that also this plugin is not dependent on any other plugin but the core plugin.

Finally, the `cz.cuni.mff.kuba.breakout.extra` plugin extends context menu of the text editor in Eclipse to allow adding output breakpoints upon opened file.
The stop condition settings are derived from the position or the selection in the editor.

3.2 Plugin cz.cuni.mff.kuba.breakout

The cz.cuni.mff.kuba.breakout plugin is the core plugin. It defines contracts for output gates and stop conditions (both Java interfaces and Eclipse extension points) and provides integration into Eclipse IDE.

In the beginning, there was a question how to extend the Eclipse environment. As the ideal result it was considered to extend this platform using its extension points and using classes only from exported packages. On the other hand, if this way would not be possible, modification of entire org.eclipse.jdt.debug or org.eclipse.jdt.debug plugins would be needed. The first possibility was preferred.

Finally, a solution was found in the form of an extension using the extension points but several classes from non-exported packages had to be used – in these cases the source code contains the @SuppressWarnings("restriction") annotation that denotes using a class from such packages. Namely these are the following three classes:

- org.eclipse.jdt.internal.debug.core.breakpoints.JavaBreakpoint
- org.eclipse.jdt.debug.ui.JDIModelPresentation
- org.eclipse.jdt.internal.debug.ui.propertypages.JavaBreakpointPage

3.2.1 Eclipse integration

Concerning Eclipse integration fulfilling the following goals was required:

1. To enable the user to choose, configure and add the output breakpoint. Similar approach as when adding an exception breakpoint was chosen – via a button in the Breakpoints view (see Figure 5.2).

2. To ensure that the added breakpoint will appear in the Breakpoints view with a correct label and icon. Output breakpoints have a property page that can be displayed via the context menu.
3. To ensure that this output breakpoint will be installed when a Java application is launched. That means that this output breakpoint can install breakpoints that are needed for its work.

4. To ensure that the debugee will stop and the Eclipse IDE will show its state when the stop condition of this output breakpoint is fulfilled.

**Adding output breakpoint**

The button for adding an output breakpoint is defined as a view action, declared in `plugin.xml` as follows (note that `toolbarPath` defines place on the toolbar where the button for adding an exception breakpoint already is; therefore this button will be placed next to it):

```xml
<extension point="org.eclipse.ui.viewActions">
  <viewContribution
    id="cz.cuni.mff.kuba.breakout.AddBreakpointViewAction"
    targetID="org.eclipse.debug.ui.BreakpointView">
    <action
      class="cz.cuni.mff.kuba.breakout.ui.action.AddJavaOutputBreakpointAction"
      icon="icons/breakout.png"
      id="cz.cuni.mff.kuba.breakout.AddJavaOutputBreakpointAction"
      label="Add Java Output Breakpoint"
      style="push"
      toolbarPath="javaExceptions">
    </action>
  </viewContribution>
</extension>
```

The `AddJavaOutputBreakpointAction` class provides implementation of this action. It shows the `AddJavaOutputBreakpointDialog` dialog. After this dialog is closed, by calling the `getOutputGate` method the output gate and its stop condition chosen by the user is retrieved. This output gate is passed to the `JavaOutputBreakpoint` constructor. As usual for other Java breakpoints, this constructor (beside others) calls the `register` method of the `JavaBreakpoint` class that registers this instance to the debug platform breakpoint manager.
Displaying Breakpoints view

Once an output breakpoint is registered to the breakpoint manager, it is displayed in the Breakpoints view. However, an effort is needed to display the correct icon and label.

Each breakpoint is associated with a debug model. An identifier of this debug model is returned from the `IBreakpoint.getModelIdentifier` method. When Eclipse looks for a breakpoint icon and label, first it obtains the breakpoint debug model ID and according this ID looks for debug model presentation that is associated with this ID.

In the case of Java breakpoints, the debug model ID is `org.eclipse.jdt.debug` and the debug model presentation is provided by the `JDIModelPresentation` class (from the `org.eclipse.jdt.internal.debug.ui` package).

Output breakpoints have to return `org.eclipse.jdt.debug` as their debug model ID in order to be properly installed when launching a Java debugee. However, at the same time, `JDIModelPresentation` is not able to provide a correct icon and label for output breakpoints.

Therefore, a tricky implementation of the `getModelIdentifier` method is provided – when this method is called with intention of getting an icon or label, `cz.cuni.mff.kuba.breakout.model.id` is returned. This model ID is associated with `cz.cuni.mff.kuba.breakout.core.ui.OutputBreakpointModelPresentation` that returns the correct label and icon. In all other cases, the `getModelIdentifier` method returns `org.eclipse.jdt.debug`. The full listing of this method follows:

```java
public String getModelIdentifier() {
    for (StackTraceElement e : Thread.currentThread().getStackTrace()) {
        if ("getLabel".equals(e.getMethodName()) || "getImage".equals(e.getMethodName())) {
            return OutputBreakpointModelPresentation.ID;
        }
    }
    return JDIDebugModel.getPluginIdentifier();
}
```

Once this is solved, let us focus on getting the actual icon and label. The icon is returned just according the breakpoint state – `●` (in red color) icon for an
enabled breakpoint and \(\ion{O}{out}\) (in gray color) for a disabled breakpoint. To get the label, the output breakpoint instance is asked to provide a label by calling the `getDescription` method. It is notable that the breakpoint delegates this call to the output gate’s method `describe` which can use the stop condition’s method `describe` as well.

**Installing breakpoint during debugee launch**

When launching a Java application, the `JDIDebugTarget`\(^1\) class ensures installing breakpoints into debugee virtual machine, namely its `initializeBreakpoints` method. Only the breakpoints registered in the breakpoint manager that fulfil following two conditions are installed:

- Their `getModelIdentifier` method returns value `org.eclipse.jdt.debug`.
- They inherit the `JavaBreakpoint`\(^2\) class.

The `JavaOutputBreakpoint` class fits these conditions – except for the case when displaying label or icon (in detail described above), the `getModelIdentifier` method returns value `org.eclipse.jdt.debug`. It also inherits the `JavaBreakpoint` class.

For breakpoints that fit these conditions, the `addToTarget` method from the `JavaBreakpoint` class is called that calls further the `createRequests` method. `JavaOutputBreakpoint` provides own implementation of the `createRequests` method where the `OutputGate.init` method is allowed to create breakpoint requests. The event request manager is accessible from `JDIDebugTarget` as follows:

```java
com.sun.jdi.VirtualMachine vm = target.getVM();
EventRequestManager man = vm.eventRequestManager();
```

Note that each created breakpoint request has to be registered by calling the `JavaBreakpoint.registerRequest` method.

\(^1\)from the `org.eclipse.jdt.internal.debug.core.model` package
\(^2\)from the `org.eclipse.jdt.internal.debug.core.breakpoints` package
Handling breakpoint events

When a Java breakpoint is hit, the breakpoint that created the request that has caused this event is responsible to handle this event. The `handleBreakpointEvent` method is called, passing the event and the thread reference.

By delegating to output gate, it is decided whether to suspend JVM or not. Then, the `suspend(thread, suspendVote)` method is called, beside other it notifies breakpoint listeners that this breakpoint wants to suspend the thread. However, if there is no breakpoint listener registered for this breakpoint of all registered listeners returned `DONT_CARE` flag, suspension is not allowed. There are no breakpoints listeners by default.

Therefore, `JavaOutputBreakpointListener`\(^3\) is registered for each output breakpoint. This listener does not care (in sense of `DONT_CARE` flag) about any event but the breakpoint hit. In this case it asks the output breakpoint for the actual decision by calling `shouldSuspend(IJavaThread)`. The output breakpoint instance remembers the last suspend decision for each thread to be able to answer.

### 3.2.2 OutputGate interface

In the `core` plugin, two important interfaces are defined – namely the `OutputGate` and `StopCondition` interfaces. On Figure 3.2, their relation including the relation to the `JavaOutputBreakpoint` class is shown.

![Figure 3.2: OutputGate and StopCondition interfaces](image)

The `JavaOutputBreakpoint` class holds a reference to the output gate. The output gate holds the stop condition configuration. When the output gate observes that an output was opened, it creates a new instance of the stop condition. Since the observed output can be opened many times during a single program run, many instances of the stop condition can be created.

\(^3\)from the `cz.cuni.mff.kuba.breakout.core.breakpoint` package
Both the output gate and stop condition are configured by Map where both keys and values are Strings. This is because these configurations need to be saved into Marker\(^4\) that accepts only values typed as String, Integer or Boolean. This marker holds the whole breakpoint configuration and it is persisted when the workbench is closed so that it could be restored when the IDE is started again.

The OutputGate interface has several important methods and some convenient methods for displaying or getting configurations. The important methods are:

- **configure** is called after creating new instance. This method takes a configuration for output gate itself, a stop condition and its configuration. A stop condition configuration is given because of possible later instantiating.

- **init** takes EventRequestManager and is responsible for creating breakpoint requests during a debugee launching.

- **handleBreakpointEvent** handles breakpoint event. It typically extracts data that was written into the output medium, it passes them to the stop condition and based on its response decides about suspending the debugee. For more explanation see Figure 2.4 and section 2.3.3 Output breakpoint life cycle.

- **isJREPatchNeeded** decides whether a JRE patch is needed for the given JRE version. If so, the core plugin checks during a launch if there is -Xbootclasspath/p option among VM options. If there is not, a warning is displayed to the user.

Beside this interface, the AbstractOutputGate class is available – it provides implementation of some convenient methods and therefore simplifies implementation. Particular implementations are a part of the gate plugin.

### 3.2.3 StopCondition interface

The stop condition is an observer of a byte stream in general sense and it accepts bytes written into the output one by one. The stop condition does not know anything about the output gate. Once created, it receives bytes by the write(byte)

\(^4\)from the org.eclipse.core.internal.resources package
method. Optionally, the `open` method can be called at the beginning – some stop conditions can evaluate the beginning as the location to stop there.

The stop condition has to keep the state of observed stream (relevant part of it) that is needed to evaluate the condition. It is not necessary to keep the whole output. This is especially important for excessive sized outputs. By the same reason, the stop condition should not need to walk through the entire output to evaluate the condition since it would lead to quadratic time complexity.

Also in the case of the stop condition there is the `AbstractStopCondition` class that can be used. Particular implementations are a part of the `condition` plugin.

### 3.2.4 Extension points

There are two extension points defined in the `core` plugin – the first for output gates and the second for stop conditions. Their exact definition can be found in related EXSD schemas. For guidance how to use them refer to chapter 3.7 Extensibility.

### 3.2.5 Configuration composites

Configuration composites are parts of the Eclipse GUI introduced by the `core` plugin used to gather configuration for an output gate or a stop condition from the user. The `ConfigComposite`\(^5\) interface has to be implemented to define a new configuration composite.

First, the `fillComposite` method is called given an empty SWT Composite and it is supposed to fill it with components (like labels or text fields). Later, when the user fills information into these components, `getConfiguration` returns the configuration as `Map` where both keys and values are `Strings`.

These configuration composites are used in the `Add Java Output Breakpoint` dialog and in the `Output Breakpoint Property Page` (in the second case in read-only form).

\(^5\)from the `cz.cuni.mff.kuba.breakout.ui.configcomposite` package
3.3 Plugin cz.cuni.mff.kuba.breakout.gate

The cz.cuni.mff.kuba.breakout.gate plugin (shortly referred as the gate plugin) provides implementation of different output gates. For this purpose, it uses the cz.cuni.mff.kuba.breakout.gate extension point introduced in the core plugin.

The work of an output gate is based on observing classes and methods responsible for processing the output. Therefore, main focus in this section will be put on describing the responsible members.

For each output gate there will be a table containing observed methods, distinguishing whether this method is a part of Java API or not (J2SE 6.0 and J2SE 7.0 are considered). Observing of the methods from API is preferred since these methods do not differ between JRE implementations.

An overview about provided output gates can be seen on Figure 3.3. Note that each class implements the OutputGate interface introduced in the core plugin. Also AbstractOutputGate is a common ancestor for all output gate classes but FileGate. The FileGate gate is a special case because it aggregates FileOutputStreamGate and FileChannelGate in the same way as if they were used one by one.

![Output gates hierarchy](image-url)

Figure 3.3: Output gates hierarchy

All output gates based on observing the java.io.OutputStream class and its descendants inherit from OutputStreamGate. Though its descendants differ
in the way of identifying the responsible instance, they share all three write methods from the OutputStream class.

### 3.3.1 FileOutputStreamGate

The FileOutputStreamGate class observes the java.io.FileOutputStream class that is responsible for writing into files. An instance of FileOutputStream is created using a constructor with the File parameter. The File class stores the path to file in its private field path. Another way to identify the corresponding File instance is to observe all constructors of the File class and remember the unique IDs of matching instance and then to compare it with the FileOutputStream constructor parameter.

Although the java.io.FileOutputStream class has five public constructors (see Figure 3.4), it is sufficient to observe only the (File, boolean) constructor. Based on implementation review we know that the first three constructors delegate the creation on the (File, boolean) constructor and the last one cannot be used for identifying a file by name since it uses file descriptor (a number).

![Figure 3.4: Used parts of java.io.OutputStream class](image)

It is important to know that when using a method entry breakpoint to observe a method it is necessary to observe the class where the method is implemented, not where it is declared (in an interface or abstract method). Similarly, when overriding, the overriding method should be observed. Therefore, although all write methods are declared (and two of them are even implemented there) in the
OutputStream class, they are observed in the FileOutputStream class because this class has its own implementation.

<table>
<thead>
<tr>
<th>Observed class</th>
<th>Member signature</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io.FileOutputStream</td>
<td>&lt;init&gt;(File file, boolean append)</td>
<td>Yes</td>
</tr>
<tr>
<td>java.io.File</td>
<td>String path</td>
<td>No</td>
</tr>
<tr>
<td>java.io.FileOutputStream</td>
<td>write(int b)</td>
<td>Yes</td>
</tr>
<tr>
<td>java.io.FileOutputStream</td>
<td>write(byte b[])</td>
<td>Yes</td>
</tr>
<tr>
<td>java.io/FileOutputStream</td>
<td>write(byte b[], int off, int len)</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.1: List of observed classes and inspected fields by FileOutputStreamGate

In Table 3.1, there is an overview about the observed classes, methods and fields. The only member that is not a part of API is the path field but it could be replaced by the observing constructors as discussed above.

### 3.3.2 SystemOutputGate

The SystemOutputGate class enables observing the standard and error outputs. In the Java language, these outputs are accessible from the java.lang.System class, namely from its static fields out and err. Both these fields refer to an instance of the java.io.PrintStream class.

Figure 3.5: Extracting FileOutputStream for System outputs

When the out and err fields are inspected at runtime (in debug mode), they have the structure shown in Figure 3.5. The PrintStream class has a protected field out that holds a reference to the inner stream. In this case, the inner stream is a BufferedOutputStream. Again, BufferedOutputStream has a protected
field `out` that holds a reference to the inner stream. Now the inner stream is of
the `FileOutputStream` type which is the instance we want to observe.

When accessed, this stream is already open and ready to accept output data. Therefore, when an output breakpoint initialises, an instance of underlying `FileOutputStream` is obtained and method entry breakpoints to observe this instance are installed.

<table>
<thead>
<tr>
<th>Observed class</th>
<th>Member signature</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.System</code></td>
<td><code>java.io.PrintStream out</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>java.lang.System</code></td>
<td><code>java.io.PrintStream err</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>java.io.FilterOutputStream</code></td>
<td><code>java.io.OutputStream out</code></td>
<td>No</td>
</tr>
<tr>
<td><code>java.io.FileOutputStream</code></td>
<td><code>write(int b)</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>java.io.FileOutputStream</code></td>
<td><code>write(byte b[])</code></td>
<td>Yes</td>
</tr>
<tr>
<td><code>java.io.FileOutputStream</code></td>
<td><code>write(byte b[], int off, int len)</code></td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.2: List of observed classes and inspected fields by SystemOutputGates

In Table 3.2, there is an overview about the observed classes, methods and fields. The only members that is not a part of API are the protected `out` fields that hold inner streams.

Note that `SystemOutputGate` is an abstract class that holds all the logic. Its descendants `SystemOutOutputGate` and `SystemErrOutputGate` only specify which field in the `System` class should be used.

### 3.3.3 SocketOutputStreamGate

When two nodes (computers or applications) are communicating via the TCP protocol, each node is identified by a socket – a couple of an IP address and a port number. From the application viewpoint this application’s address and port number are local, the counterpart’s address and port number are the remote ones.

The `java.net.SocketOutputStream` class is responsible for the network communication using sockets. In its constructor, a socket instance is given that contains both locale and remote port numbers.

Figure 3.6 shows that the situation differs between Java 6 and Java 7. While in Java 6 the constructor takes `PlainSocketImpl`, `AbstractPlainSocketImpl` is taken in Java 7. In both cases these classes inherit from `SocketImpl` that holds the information about port numbers – in protected fields `port` and `localport`.

34
Table 3.3: List of observed classes and inspected fields by SocketOutputStream-Gate

In Table 3.3 is an overview about observed classes, methods and fields. Only the protected fields `port` and `localport` are not a part of API.

![Diagram showing the inheritance and methods of SocketOutputStream](image)

Figure 3.6: Obtaining port number for SocketOutputStream

### 3.3.4 FileChannelGate

Since Java 1.4 there is, beside output streams from the `java.io.*` packages, a new way to write into a file – it is represented by the classes from the `java.nio.*` packages which introduce the terms *channels* and *buffers*.

There is the `java.nio.channels.FileChannel` class for writing to a file. Note that this class is abstract (while e.g. FileOutputStream was not abstract and therefore it was possible to observe its methods). Thus, this abstract class has to be extended by a proprietary implementation, in the case of Oracle Java it is the `sun.nio.ch.FileChannelImpl` class.
<table>
<thead>
<tr>
<th>Observed class</th>
<th>Member signature</th>
<th>API</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.io.FileDescriptor</td>
<td>&lt;init&gt;()</td>
<td>Yes</td>
</tr>
<tr>
<td>java.io.FileOutputStream</td>
<td>&lt;init&gt;(File, boolean append)</td>
<td>Yes</td>
</tr>
<tr>
<td>java.io.File</td>
<td>String path</td>
<td>No</td>
</tr>
<tr>
<td>sun.nio.ch.FileChannelImpl</td>
<td>&lt;init&gt;(FileDescriptor, boolean, boolean, Object, boolean)</td>
<td>No</td>
</tr>
<tr>
<td>sun.nio.ch.FileChannelImpl</td>
<td>write(ByteBuffer src)</td>
<td>No</td>
</tr>
<tr>
<td>sun.nio.ch.FileChannelImpl</td>
<td>write(ByteBuffer[], int offset, int length)</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer[R]</td>
<td>byte[] hb</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer[R]</td>
<td>int position</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer[R]</td>
<td>int limit</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.HeapByteBuffer[R]</td>
<td>int offset</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.DirectByteBuffer[R]</td>
<td>sun.misc.Unsafe unsafe</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.DirectByteBuffer[R]</td>
<td>int position</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.DirectByteBuffer[R]</td>
<td>int limit</td>
<td>No</td>
</tr>
<tr>
<td>java.nio.DirectByteBuffer[R]</td>
<td>long address</td>
<td>No</td>
</tr>
<tr>
<td>sun.misc.Unsafe</td>
<td>getByte(long address)</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.4: List of observed classes and inspected fields by FileChannelGate

Generally speaking, **FileChannel** is the most difficult output method to observe which is also evidenced by a quite long Table 3.4. The table is separated into four parts – let us discuss them one by one:

1. **Identifying the responsible instance.** The `FileChannelImpl` constructor contains only a reference to `FileDescriptor` that does not contain information about file path. Therefore, we need to identify the correct instance of `FileDescriptor` in the moment of its creation. The `FileDescriptor` instance is created in the `FileOutputStream` constructor where is accessible the corresponding `File`. Thus, when the program stops in the `FileDescriptor` constructor, one level above in the stack trace is the `FileOutputStream` constructor with `File` instance as a local variable (method argument). The `File` class contains the path in its private field `path`. The `FileDescriptor` object reference is remembered and later compared with the `FileChannelImpl` constructor argument.

2. **Observing write methods.** This part is quite simple since there are only two `write` methods, one taking a single `ByteBuffer` and the other one that takes an array of `ByteBuffers` and bounds defining which buffers are actually used. First it has to be decided what type of buffer was used to call the `write` method.
3. **Heap (non-direct allocated) byte buffers.** Heap byte buffers are created by the static `ByteBuffer.allocate` method on the heap. Heap buffers are represented by the non-public `HeapByteBuffer` and `HeapByteBufferR` classes (the second is a read-only variant of the previous). In both cases there is a private byte array `hb` and private fields `position`, `limit` and `offset` that determine which part of buffer will be written into a channel – bytes indexed from `(offset + position)` to `(offset + limit - 1)`.

4. **Direct (directly allocated) byte buffers.** Direct byte buffers are created by the static `ByteBuffer.allocateDirect` method. The contents of direct buffers may reside outside of the normal garbage-collected heap. Direct buffers are represented by the non-public classes `DirectByteBuffer` and `DirectByteBufferR` (which is a read-only variant of the previous). In both cases there are private fields `position`, `limit` and `address` that determine part of memory that will be written into a channel – bytes indexed from `(address + position)` to `(address + limit - 1)`. This this only case buffer value cannot be inspected as a suspended program state but it has to be retrieved by calling the `Unsafe.getByte` method.

### 3.4 Plugin cz.cuni.mff.kuba.breakout.condition

The `cz.cuni.mff.kuba.breakout.condition` plugin (shortly referred as `condition` plugin) provides implementation of different stop conditions. For this purpose, it uses the `cz.cuni.mff.kuba.breakout.condition` extension point introduced in the `core` plugin.

Each stop condition implements the `StopCondition` interface. For a description refer to section 3.2.3 StopCondition interface. Work of a stop condition is based on receiving bytes one by one as they are written into the output. After each written byte the stop condition decides whether the condition is fulfilled or not.

The `condition` plugin provides following stop conditions (note that processing of a single byte write takes a constant time for each enlisted stop condition except the last one – the `RegexStopCondition`):

- **AbsolutePathStopCondition** stops when a byte is written at given absolute position. This stop condition is not encoding aware.
• **AbsoluteRangeStopCondition** stops whenever a byte is written between given absolute bounds. The bounds are inclusive. This stop condition is not encoding aware.

• **LineAndCharStopCondition** stops when a byte is written at given line and char (position on the line). It supports CR, LF, CR-LF and LF-CR line endings. This stop condition is not encoding aware.

• **LineAndCharRangeStopCondition** stops when a byte is written between given line and char bounds (starting line and char, ending line and char). It supports CR, LF, CR-LF and LF-CR line endings. This stop condition is not encoding aware.

• **SubstringStopCondition** stops when the entire output is suffixed by given string. This stop condition is encoding aware (with limitations described below).

• **RegexStopCondition** stops when the entire output matches given regular expression. This stop condition is encoding aware. Note that the regular expression is matching the whole output therefore when looking for `needle` substring it has to be expressed by `.needle` regular expression.

---

Stop conditions has quite simple inheritance hierarchy – see Figure 3.7. All concrete stop conditions inherit directly from **AbstractStopCondition** except the case of **RegexStopCondition**. In this case it was needed to buffer whole output to evaluate a regular expression. The **BufferingStopCondition** provides buffering in a byte array that doubles its size when the capacity is reached. After each written byte the whole buffer is passed to the abstract method **matches** where the descendant class decides whether its condition was fulfilled. The
**RegexStopCondition** first converts this byte array into a string using the correct encoding. Since the whole output is used, the limitations concerning encoding described later do not apply for this stop condition. Then, using the pre-compiled `java.util.regex.Pattern` is this string compared with the regular expression.

This procedure leads to quadratic time complexity. Theoretically, a regular expression could be detected using a deterministic finite automata. In that case, the processing of a single byte would take a constant time (just a change of the automata state according to the state transition table).

### 3.4.1 Encoding

Since the stop condition receives bytes it is byte-oriented. However it enables to identify a charset (an encoding) that can be used to interpret the byte stream as a character stream. It depends on the particular stop condition implementation whether it will be encoding aware or not.

Although all four positional stop conditions are declared not to be encoding aware, they work correctly with all single-byte encodings (including Windows-1250 or ISO-8859-2, also known as Latin 2). In the case of multi-byte encodings they would indicate fulfilling of the condition earlier that it actually happened from user viewpoint (since the user is usually character oriented).

The **SubstringStopCondition** receives from the user a Unicode encoded substring that is converted into a byte array using the defined encoding. This $n$-byte long array is then compared with last $n$ bytes from output. Therein lies the limitation. Two problems can occur:

1. It is theoretically possible to create a encoding that would not be byte aligned. For example, each character would be encoded by 12 or 20 bits. In this case some substring occurrences would not be matched. However, there is not known such encoding.

2. Another problem are false positives. Let us imagine an encoding as shown in Table 3.5. When looking for a substring “Z”, it will be converted to a byte array \([B1, C2]\). Then, when a text containing “XY” will be written into output, the \([B1, C2]\) array will be matched though the “Z” letter was not present. The false positives problem could be avoided by decoding the output byte stream identifying the bounds of characters. Another way
to avoid this is to remember the entire output, after each new byte to convert it to Unicode string and then to check against the substring by the `String.endsWith` method. Note that the second way is both memory and time consuming.

<table>
<thead>
<tr>
<th>Character(s)</th>
<th>Are encoded as</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>A0 B1</td>
</tr>
<tr>
<td>Y</td>
<td>C2 D3</td>
</tr>
<tr>
<td>Z</td>
<td>B1 C2</td>
</tr>
<tr>
<td>XY</td>
<td>A0 B1 C2 D3</td>
</tr>
</tbody>
</table>

Table 3.5: Imaginary encoding to demonstrate false positives

### 3.5 Plugin `cz.cuni.mff.kuba.breakout.extra`

The `cz.cuni.mff.kuba.breakout.extra` plugin (shortly referred as extra plugin) provides a user-friendly extension of text editor that allows to add an output breakpoint using the context menu in editor.

Namely, it adds an instance of `FileGate` set to observe the actual opened file. The user can choose from stop conditions provided by the `gate` plugin. In the case of range stop conditions, user can define the range using text selection.

Note that the `extra` plugin depends on all others (`core`, `gate` and `condition`) plugins. For more detail see the source code or related JavaDoc documentation.

### 3.6 Known bugs

In fact, there is only one know bug. When the debugee stops at a output breakpoint, once or more times, at the end of the `main` method the target JVM freezes (the debugee does not terminates). Albeit that does not prevent this solution to work, it can be a bit annoying.

When properly watched, a method exit event on the `main` method occurs. Later, the `Thread.exit` method is called by the system and it seems also to finish correctly. After that the target JVM is still alive. When suspended via Eclipse GUI, four threads remain there as follows:
Moreover, when the `System.exit(0)` call is added at the end of the `main` method, target JVM finishes so that the described freezing does not occur. Also, when “Suspend VM” suspend policy on output breakpoint is set, this does not happen.

It obviously relates to the way output breakpoints are integrated into Eclipse IDE. Unfortunately, the author was not able to find the cause and fix it.

### 3.7 Extensibility

The entire solution can be easily extended by new output gates or stop conditions using extension points designed for that purpose. In the following two sections step-by-step tutorial how to do it will be described.

#### 3.7.1 Creating an output gate

In the beginning one has to know which classes are responsible for the particular output. Also, it must be ensured that none of observed methods are `native`. If so, these methods have to be reimplemented not to be `native`, if possible. Otherwise this output cannot be observed. Then, the following steps are to be taken:

- To create a class that implements the `OutputGate` interface. It is recommended to extend the abstract class `AbstractOutputGate` for simplification. For description of the interface refer to section 3.2.2 OutputGate interface.

- To create (or reuse) a class that implements the `ConfigComposite` interface. It is recommended to extend the abstract class `AbstractConfigComposite` for simplification. For more detail about configuration composites refer to section 3.2.5 Configuration composites.
3.7.2 Creating a stop condition

To create a stop condition is a bit easier than to create an output gate since stop conditions do not depend on any classes but they only receive a stream of bytes and return boolean answer after each of them. To create a stop condition the following steps are to be taken:

- To create a class that implements the StopCondition interface. It is recommended to extend the abstract class AbstractStopCondition for simplification. For description of the interface refer to section 3.2.3 StopCondition interface.

- To create (or reuse) a class that implements the ConfigComposite interface. It is recommended to extend the abstract class AbstractConfigComposite for simplification. For more detail about configuration composites refer to section 3.2.5 Configuration composites.

- To register this stop condition in plugin.xml like this:
  <extension point="cz.cuni.mff.kuba.breakout.stopcondition">
    <stopCondition
      class="com.example.ExampleStopCondition"
      configComposite="com.example.ExampleConfigComposite"
      name="Example stop condition "/>
  </extension>

It is sufficient to add a plugin containing the newly created stop condition or output gate to the plugins folder of your Eclipse installation (next to the four plugins of this solution) and restart Eclipse. The new stop condition or the new output gate will be now present in the Add Java Output Breakpoint dialog.
4. Evaluation

This chapter sums up this thesis, reviews conclusions and achieved results. It also recalls limitations of this solution and propose possible future work in this field. There is also included a report about performance testing in this chapter.

Goals achievement

Generally speaking, all the objectives (as listed in the beginning of this thesis) were achieved. During the realisation phase, a proof-of-concept project was assembled to ensure that the selected approach has a chance to work. Once proved, it was further realised. During the time there arose problem with native methods observation and also the so far unresolved bug described in section 3.6 Known bugs. In addition to these there were no more obstacles during the realisation.

As a disadvantage of this solution one can take its dependency of the target virtual machine JRE and quite high performance requirements (described later).

Limitations

There are a few limitations that are described in detail in chapter 2.5 Limitations:

- The debugee JRE is limited to the Oracle Java 6 or 7. In the case of the Java 6 there is needed a JRE patch. The patch is described in section 2.3.2 JRE modification and its usage is described in Appendix B: User manual.

- The provided solution plugins are written primarily for Eclipse 3.7 Indigo. It is possible that it will work also in other versions but with no warranty.

- An obvious limitation is also for the outputs that are not supported by an output gate. For example, when there is a (native) library that writes into a file, without a special output gate observing this library, the output into these files cannot be observed.

Related work

Author is not aware of any related work addressing this issue.
4.1 Performance

A simple performance test was performed to figure out how much does the observation needed for output breakpoints slow down the program run. For this purpose, a simple Java class was composed. The interesting part of it follows:

```java
FileOutputStream out = new FileOutputStream(new File("c:\test.txt"));
long start = System.currentTimeMillis();
for (int i = 0; i < 10000; i++) {
    out.write(i % 50 + 50);
}
long finish = System.currentTimeMillis();
int duration = (int) (finish - start);
out.close();
```

Simply said, it writes ten thousand times a byte into c:\test.txt file. Time that elapsed is measured only for writes, opening and closing is not included. There are five cases to measure:

- A launch without any breakpoint – it will be a reference measurement.
- A launch with line breakpoint on the line `out.write(i % 50 + 50)` with hit count set to a number greater than ten thousand so that it would not stop. Note that hit count is evaluated by the deuggee JVM.
- A launch with conditional line breakpoint on the line `out.write(i % 50 + 50)` with a condition "false" that is obviously never true. Note that conditions are evaluated by the debugger.
- A launch with an output breakpoint containing a FileOutputStreamGate set to observe a non-existing file.
- A launch with an output breakpoint containing a FileOutputStreamGate set to observe c:\test.txt file and a AbsolutePosition stop condition set to a number greater that ten thousand so that it would not stop.

For each case there were ten measurements performed. The times in Table 4.1 are in milliseconds. Beside them there are average times for each case and
the standard deviation calculated as \( \sqrt{\frac{1}{N} \left( \sum_{i=1}^{N} t_i^2 \right) - \overline{t}^2} \), where \( N = 10 \). In the last column is a relative average time relative to the case with no breakpoint.

<table>
<thead>
<tr>
<th>Installed breakpoint</th>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>( t_3 )</th>
<th>( t_4 )</th>
<th>( t_5 )</th>
<th>( t_{10} )</th>
<th>Avg.</th>
<th>St.dev.</th>
<th>Rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>No breakpoint</td>
<td>110</td>
<td>110</td>
<td>109</td>
<td>109</td>
<td>109</td>
<td>109.4</td>
<td>0.5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Line breakpoint</td>
<td>204</td>
<td>187</td>
<td>203</td>
<td>188</td>
<td>187</td>
<td>192.2</td>
<td>7.3</td>
<td>1.8</td>
<td></td>
</tr>
<tr>
<td>Line breakpoint with condition</td>
<td>4735</td>
<td>4078</td>
<td>4094</td>
<td>4265</td>
<td>4297</td>
<td>4259.4</td>
<td>173.6</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>Output breakpoint (non-existing file)</td>
<td>7672</td>
<td>7531</td>
<td>7625</td>
<td>7640</td>
<td>7625</td>
<td>7631.1</td>
<td>86.6</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>Output breakpoint (c:\test.txt)</td>
<td>12219</td>
<td>12687</td>
<td>13328</td>
<td>14000</td>
<td>14406</td>
<td>14423.2</td>
<td>1327.3</td>
<td>132</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Performance test result times (in milliseconds)

**Observing the c:/test.txt file**

Using the JProfiler was realized that the method `handleBreakpointEvent` in the `JavaOutputBreakpoint` class takes 35% of total time.

More interesting than that is the fact that the `handleBreakpointEvent` method is called four times during one cycle. It is because there are two method entry requests – the first was created to observe file openings, the second one was created to observe writes. Furthermore, each call of the `write(int)` method causes a call of another `native` method on the same class that actually makes the write. Hence, two requests multiplied by two methods are exactly the four calls of the `handleBreakpointEvent` method.

It is also remarkable that in this case the measured times have an increasing tendency which does not happen in the case when observing non-existing file. This increasing tendency also causes quite big value of the standard deviation in this case. This behaviour indicated a possible hidden problem.

To sum up, using output breakpoints has severe performance impact – according to the measurement above runs the debugee about hundred times slower. In reality, the slowdown depends on the measure of write activity. Therefore, if the debugee spend only five percent of its time with writing into files then the run will be only five times as long.
4.2 Future work

There are several ideas how to possibly improve and extend the entire solution in the future:

- When a debugged program is suspended, last frame of stack trace is focused. However, in the case of output breakpoints the last frame is used to be a JRE method (like FileOutputStream.write). It would be useful to find in the stack trace the first non-JRE method and set the focus on this frame so that the user can see immediately the interesting part of the stack trace.

- When using BufferedOutputStream or buffers from java.nio, the actual write happens later, far from the user code that has caused it. Therefore, the support for early detection could be added. In the case of buffers from java.nio one could observe there buffers in general, without knowledge about their destination since it is not known in the moment of filling the buffer. In the case of BufferedOutputStream one can “dig out” the underlying FileOutputStream similarly as in the case of the standard output.

- Few more stop conditions could be added – e.g. stop conditions supporting regular expressions or XPath expressions. The regular expressions could be simply evaluated when the entire output would be stored as a String. However, an implementation not doing this would be appreciated. In the case of XPath expressions, only a subset can be supported, since the XML file would not be completed – for example [last()] predicate or following axis is not well defined in the middle of a XML document.

  Note: A stop condition that supports regular expressions using buffering the entire output was finally implemented – see RegexStopCondition.

- Useful task is also to fix the bug described in section 3.6 Known bugs.

- If possible, the performance could be improved – see section 4.1 Performance for detail. Also, other approaches described in section 2.2 Choice of solution that might have better performance could be explored.
Conclusion

In conclusion, we can say that all stated goals was achieved. After exploring possible approaches to output breakpoints, the chosen solution was designed, and realised as a set of Eclipse plugins.

Using these plugins, user can observe files, the standard and error output and outgoing network communication. Several stop conditions are provided – absolute (range) position, line and char (range) position, substring and finally regular expression stop conditions.

Since the solution provides extension points for output gates and stop conditions, anyone can add his own output gate or stop condition.

Finally it should be noted that the topic of output breakpoints for Java was not completely exhausted and provides many opportunities for future work.
Bibliography

Eclipse Debug Project [online].  

Eclipse Java development tools (JDT) [online].  

Eclipse Projects [online].  
http://eclipse.org/projects/listofprojects.php

Platform architecture [online].  

SWT: The Standard Widget Toolkit [online].  
http://eclipse.org/swt/


Java Debug Interface - JavaDoc documentation [online].  
http://docs.oracle.com/javase/6/docs/jdk/api/jpda/jdi/

Java Debug Wire Protocol [online].  
http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jdwp-spec.html

Java Platform Debugger Architecture [online].  
http://java.sun.com/javase/technologies/core/toolsapis/jpda/
Java Platform Debugger Architecture Structure Overview [online].  
http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html

java - the Java application launcher [online].  
http://docs.oracle.com/javase/6/docs/technotes/tools/windows/java.html

JNI Types and Data Structures [online].  
http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/types.html

Java Virtual Machine Tool Interface [online].  
http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/index.html

JPDA Connection and Invocation Details [online].  
http://docs.oracle.com/javase/1.4.2/docs/guide/jpda/conINV.html

Method (Java Debug Interface) [online].  
http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdi/com/sun/jdi/Method.html

http://www.eclipse.org/articles/article-Debugger/how-to.html
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>JPDA architecture and possible usages by a debugger</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Eclipse SDK (taken from [4])</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Basic concept of output breakpoints</td>
<td>17</td>
</tr>
<tr>
<td>2.2</td>
<td>The MethodEntryEvent class</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Initialization of output breakpoints</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Handling breakpoint events</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Plugins dependencies</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>OutputGate and StopCondition interfaces</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>Output gates hierarchy</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Used parts of java.io.OutputStream class</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>Extracting FileOutputStream for System outputs</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>Obtaining port number for SocketOutputStream</td>
<td>35</td>
</tr>
<tr>
<td>3.7</td>
<td>Stop conditions inheritance hierarchy</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Warning message when missing needed JRE patch</td>
<td>53</td>
</tr>
<tr>
<td>5.2</td>
<td>Breakpoints view with different types of output breakpoints</td>
<td>54</td>
</tr>
<tr>
<td>5.3</td>
<td>Adding output breakpoint via button in Breakpoints view</td>
<td>55</td>
</tr>
<tr>
<td>5.4</td>
<td>Adding output breakpoint via context menu in editor</td>
<td>56</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(J)VM</td>
<td>(Java) Virtual Machine</td>
</tr>
<tr>
<td>API</td>
<td>Application programming interface</td>
</tr>
<tr>
<td>EXSD</td>
<td>XSD file describing extension point in an Eclipse plugin.</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated development environment</td>
</tr>
<tr>
<td>JAR</td>
<td>Java ARchive</td>
</tr>
<tr>
<td>JDI</td>
<td>Java Debug Interface</td>
</tr>
<tr>
<td>JDT</td>
<td>Java development tools</td>
</tr>
<tr>
<td>JDWP</td>
<td>Java Debug Wire Protocol</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JPDA</td>
<td>Java Platform Debugging Architecture</td>
</tr>
<tr>
<td>JVM TI</td>
<td>JVM Tools Interface</td>
</tr>
<tr>
<td>PDE</td>
<td>Plug-in Development Environment</td>
</tr>
<tr>
<td>SWT</td>
<td>The Standard Widget Toolkit is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. [5]</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema, one of several XML schema languages.</td>
</tr>
</tbody>
</table>
Appendix A: Content of attached CD

This appendix describes the structure of the attached CD.

/dist folder contains distribution sources. For installation instructions see Appendix B: User manual.

/dist/plugins folder contains the provided solution as four Eclipse plugins.

/dist/patch folder contains the JRE patch for debuges running on Java 6. For more explanation see Appendix B: User manual.

/doc folder contains documentation.

/doc/javadoc folder contains JavaDoc documentation of provided Eclipse plugins.

/doc/text folder contains text of this thesis in both PDF and \LaTeX sources.

/src/ws folder contains Java source files and other related resources of provided solution with Eclipse metadata.

/src/runtime-ws folder contains a testing Eclipse workspace with simple applications for testing different types of output breakpoints.
Appendix B: User manual

Installation

To install this solution there are two steps needed although the second one can be omitted in the case of using JRE 7 (see below).

First step consists of copying plugins to your Eclipse installation. Supported version is Eclipse 3.7 Indigo – this version can be downloaded from this location: http://www.eclipse.org/downloads/packages/release/indigo/r

Copy all four JARs from /dist/plugins into plugins folder of your Eclipse installation. If running, Eclipse has to be restarted to reflect these new plugins.

If you want to use output breakpoints for debuggee running on JRE 6, you need to copy the patch from /dist/patch somewhere to your filesystem (e.g. /my/space/patch). Later, you have to modify each launcher in Eclipse by adding following VM option:

-Xbootclasspath/p:/my/space/patch

If you forgot to add this option and the patch is needed for any of installed output breakpoint, a warning message will be shown (see Figure 5.1). Note that the patch is only needed when using file output based on the FileOutputStream class.

![Problem Occurred](image)

Figure 5.1: Warning message when missing needed JRE patch

Another option for applying the patch is to replace patched .class files directly in rt.jar of your JRE. In that case, you can ignore the above mentioned warning.

However, the easiest way to use output breakpoints is to use JRE 7 as debuggee runtime environment. It that case, no patch is needed at all.
Usage

After installation you can run your Eclipse. Now you should see the Add Java Output breakpoint button (ether) in Breakpoints view which indicates successful installation (see Figure 5.2).

![Breakpoints view with different types of output breakpoints](image)

Figure 5.2: Breakpoints view with different types of output breakpoints

Using output breakpoints is very similar to using other types of Java breakpoints. Created output breakpoints are shown in the standard Breakpoints view. Their type is indicated by a special icon ether. What is specific for output breakpoints is the way how they are being added.

Adding Output Breakpoints

Output breakpoint consists of two parts – an output gate and a stop condition. There are several kinds of output gates:

- File – given a file name (full path); this can be further divided according to its implementation to:
  - File (OutputStream) – output based on the java.io.OutputStream class
  - File (Channel) – output based on the java.nio.channels.FileChannel class
  - File (both) – both preceding together
- System.out – the standard output
- System.err – the standard error output
- Port – given a port number, local or remote one
As a stop condition, you can choose from:

- Absolute position (or range)
- Line and char position (or range)
- Substring
- Regular expression – note that the regular expression is matching the whole output therefore when looking for *needle* substring it has to be expressed by .*needle regular expression.

To add an output breakpoint, click on the *Add Java Output breakpoint* button ( ) in the Breakpoints view. *Add Java Output Breakpoint* dialog will appear (see Figure 5.3).

![Add Java Output Breakpoint](image)

**Figure 5.3:** Adding output breakpoint via button in Breakpoints view
In this dialog, you choose an output gate that you want to use, filling its configuration. Similarly, you choose a stop condition and its configuration. In some cases you also have to choose Charset – encoding used on output medium.

Another way to add an output breakpoint is using context menu in text editor directly in your Eclipse workspace (see Figure 5.4). This way you can add only FileGate output gate, with several options of stop conditions. Adding output breakpoint via context menu seems to be more user-friendly and avoids misspelling the file name.

![Figure 5.4: Adding output breakpoint via context menu in editor](image)

The context menu in the text editor can be used with or without a selection. Without selection you can create output breakpoint with Absolute position and Line and char stop conditions. As the position it will be taken the char/position before cursor. With selection (single or multiple lines) you can create Absolute position range, Line and char range and Substring stop conditions.

**Other actions with output breakpoints**

Other actions with output breakpoints are identical with other breakpoint types. Namely following actions are done in the same way:

- enabling/disabling breakpoint
- setting suspend policy - Thread or VM
- deleting breakpoint
- showing property page

Note that although you can set Hit count, this value is ignored.