Exjdb - Experimental Java Debugger

Daniel Tschan

23rd December 2002

Abstract

Debugging has always been a difficult task for a Java programmer. This specially applies to closed environments with built-in virtual machines like applets, servlets or Lotus Domino Agents. This paper sheds light on an alternate debugging approach, the source code debugger.

Usually, Java debuggers work by communicating with a debugging virtual machine (a virtual machine extended with special debugging features and an interface to communicate with a debugger). Browsers, web servers and other applications that include a Java virtual machine usually work with a version without debugging support or with debugging disabled. So most debuggers fail to debug applets, servlets and the like running in their normal environment. In many cases it is possible to debug the code in a modified environment but this often requires expensive configurations which cause the developer to debug his application by adding print statements. The modified environment may have a different behavior than the original one, too. Some development tools also use proprietary approaches for debugging, which means that they can only debug processes that run on their own, specially modified virtual machine.

This lead to the idea to modify the debugged program itself, instead of modifying the virtual machine. The program is instrumented with special debug code and a communication interface. The modified application runs in its normal environment during the debugging session. This document investigates this approach on the basis of a prototype.

Contents

1. Introduction 4

2. Software 4

3. Analysis 5
- 3.1. Use Cases 5
- 3.2. Architecture 6

4. Design 7
- 4.1. Design by Contract 7
- 4.2. Package structure 8
- 4.3. Debuggee Instrumenter 9
- 4.4. Back end 9
- 4.5. Front end 10
- 4.6. Protocol 11

5. Tools 14
- 5.1. JFlex 14
- 5.2. CUP 14
- 5.3. ASTG 14
- 5.4. GNU Autotools 17

6. Implementation 17
- 6.1. Utility classes 18
- 6.2. GUI components 18
- 6.3. Debuggee instrumenter 18
  - 6.3.1. JavaUnparser class 19
  - 6.3.2. DebugeeInstrumenter class 20
- 6.4. Back End 23
  - 6.4.1. BackEnd class 23
  - 6.4.2. BackEndProxy class 24
- 6.5. Front End 25
  - 6.5.1. ExjdbDocument class 25
  - 6.5.2. ExjdbHighLight class 25
  - 6.5.3. FrontEnd class 26
  - 6.5.4. FrontEndController class 26
  - 6.5.5. FrontEndProxy class 27
  - 6.5.6. FrontEndView class 27
- 6.6. Immediate Instrumenter 28
  - 6.6.1. ImmediateInstrumenter class 28
- 6.7. The Protocol 29
  - 6.7.1. Channel class 29
  - 6.7.2. Transport class 31

6.7.3. MarshalOutputStream class 31
6.7.4. MarshalInputStream class 31
6.7.5. RemoteObject class 32
6.7.6. RemoteStub class 32
6.7.7. UnicastRemoteObject class 33
6.7.8. LocateRegistry class 33
6.8. Jikes modifications 33
6.8.1. Building a shared library 34
6.8.2. Communicating with exjdb 34

7. Compilation **35**
7.1. Patching and compiling Jikes 35
7.2. Compiling exjdb 36
7.3. Compiling the documentation 37

8. Problems and Limitations **38**
8.1. SRMI synchronization 38
8.2. Jikes segfaults 38
8.3. Named pipes 39

9. Conclusion **39**

A. Grammar for ASTG Specification Files **40**

B. Java 2 Grammar **45**

1. Introduction

Each chapter in this document describes an aspect of the project. If possible the chapters are sorted by their natural order. The analysis phase of an it project is usually considered to be before the design phase, so the analysis chapter comes before the design chapter. However, this does not mean the analysis was fully completed before starting the design. The development process was an iterative one, meaning that first, there was a bit of analysis, then a bit design, then some more analysis, some more design, the first implementations, some analysis again and so on. The documentation itself was also part of this process. From time to time the documentation was updated with the knowledge gained from the iterations. By this means some parts of the implementation chapter were written before the design chapter was finished. For more informations about this process please see the book “Objektorientiere Softwareentwicklung: Analyse und Design mit der Unified Modelling Language” [6, 67pp].

2. Software

The following software has been used for the development of the prototype:

- The GNU http://www.gnu.org/ operating system
- The Linux http://www.kernel.org/ kernel
- Cygwin http://cygwin.com/provides ports of many GNU development tools (e.g. make) for Windows
- autoj http://autoj.sourceforge.net/, automated build system for Java
- Jikes http://www.research.ibm.com/jikes/, a very fast Java compiler for various platforms
- JFlex http://www.jflex.de/, a fast scanner generator for Java
- CUP http://www.cs.princeton.edu/~appel/modern/java/CUP/, a parser generator for Java
- ASTG http://astg.sourceforge.net/, an abstract syntax tree generator
- jEdit http://jedit.sourceforge.net/, extensible editor for programmers
- GNU Autotools

The documentation was created using the following applications:

- LyX http://www.lyx.org/, a WYSIWYM (What You See Is What You Mean) document processor
- Dia http://www.lysator.liu.se/~alla/dia/dia.html, a drawing program

- pdfTeX http://www.ctan.org/tex-archive/systems/pdftex/, a version of TeX that can create PDF
- BibTeX http://www.ctan.org/tex-archive/biblio/bibtex/, makes bibliographies for TeX and LaTeX

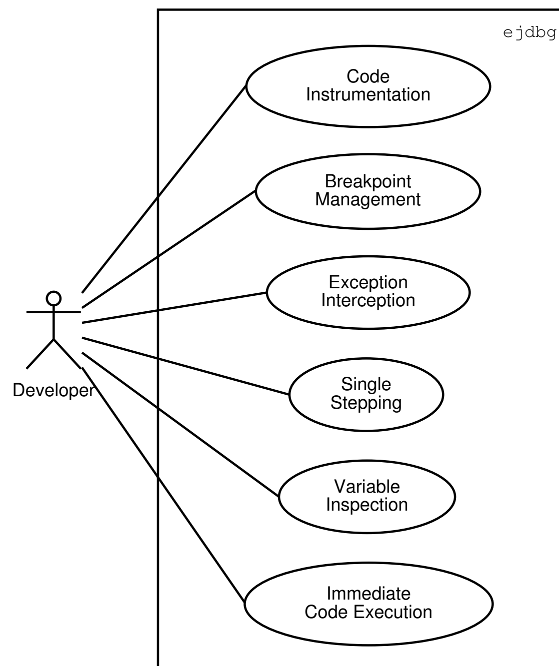
3. Analysis

The analysis and the design phase use the Unified Modelling Language (UML) for most descriptions and illustrations. UML is also used for the class diagrams in the implementation phase. For the complete UML specification please see [5]. In this chapter the different use cases and actors are identified and described. Then the architecture of the prototype is derived from them.

3.1. Use Cases

Figure 1 shows the different use cases of the system.

Figure 1: Use Case Diagram



Code instrumentation: Before a program can be debugged with exjdb it must be instrumented. The developer selects the main class of the project he wants to debug and the debugger then instruments this class and all classes

it depends on. This can be done using the command line or a GUI, which will be described later.

Breakpoint management: The developer can instruct the debugger to stop the application at specific locations by setting breakpoints. Only one breakpoint can be set per line of source code. A breakpoint can have an associated boolean condition, this is then called a watchpoint. Whenever a watchpoint is hit its associated condition is evaluated and if the associated condition evaluates to true the application is stopped, otherwise the application continues to run.

Exception interception: The debugger allows the developer to specify which exceptions should be intercepted. Whenever one of the specified exceptions is thrown the debugger stops the application on the line with the corresponding throw statement.

Single stepping: If an application is stopped the developer can instruct the debugger to execute a single statement and then stop again. This is called single stepping. Two different single step operations are available: step over and step into. Step over is threating method invocations as single instructions while step into jumps into the code of the method.

Variable inspection: If an application is stopped the developer must be able to see the values of all variables in scope.

Immediate code execution: If an application is stopped the developer can enter arbitrary Java statements and execute them directly in the debuggee. The output of the standard output stream and the standard error output stream are redirected to the debugger. This allows the developer to use System.out.print and System.err.print statemens to display values of variables or other information in the debugger.

3.2. Architecture

The use case "code instrumentation" is a special use case. It takes place before the debuggee is running and the implementation of this use case is a prerequisite for the other use cases. The use case "code instrumentation" therefore gets its own subsystem called "debuggee instrumenter". The other uses cases are all part of a debugging session which needs two parts to work. Firstly, code in the debuggee that collects information and secondly, a subsystem to control the debugging session. The first subsystem is called "front end" while the later one is called "back end".

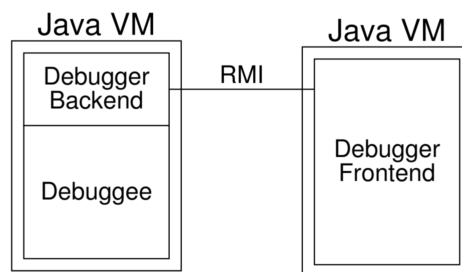
Debuggee instrumenter: The debuggee instrumenter is adding code to the debuggee (the program being debugged) which make calls to the debugger back end. Through this calls the back end receives data like the current statement of the threads running through the instrumented code, values of local variables and the like.

Debugger back end: In order to keep the additional code added by the instrumenter as small as possible, the debugger back end has to do most of the work. It collects the information it receives from the instrumentation code and communicates with the debugger front end.

Debugger front end: The front end serves as the interface between the user and the debugger back end. The user can toggle breakpoints, single step through the code, execute code fragments directly and more. The front end can communicate with multiple back ends simultaneously. This is particularly useful when debugging distributed applications.

Figure 2 shows the relations between the subsystems.

Figure 2: Subsystems



4. Design

This chapter describes the detailed design of the subsystems identified in the previous chapter and the communication between them.

4.1. Design by Contract

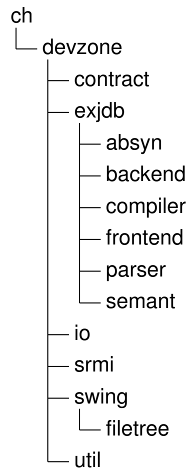
Design by contract helps to find bugs, reducing debug time and improving reliability and stability by explicitly checking contracts. Software elements always fulfill a certain contract, explicit or not. Since Java doesn't support design by contract natively, a package is introduced to provide this support. The contracts are checked with the utility class Contract. Whenever a contract is violated, an exception corresponding to the type of the contract is thrown. Four contract types are supported:

Precondition 'require': Specifies the states in which a method may be invoked

Postcondition 'ensure': Specifies the states in which a method should return

Class Invariant 'invariant': Specifies the states which a valid for objects of a class

Figure 3: Package Structure



Check Statement 'check': The check statement is used to state an assertion at any part of the program code.

Since the contract exceptions are runtime exceptions, they don't need to be declared in the throws clause.

The package also contains a subpackage doclet which contains a JavaDoc doclet. It works and has the same parameters as the standard doclet (which is used if javadoc is run without -doclet). The doclet extracts the contracts from the source code and adds them to the JavaDoc HTML documentation.

4.2. Package structure

In conformance with the Java Language Specification [3] all packages of the project are subpackages of ch.devzone (devzone.ch is one of my domains). Figure 3 shows the package structure of the project.

The packages are organized by function. Classes outside exjdb package are not directly related to exjdb and can be used in other projects.

contract: Design by contract support

exjdb: Exjdb main classes

exjdb.absyn: Abstract syntax tree classes, generated by ASTG

exjdb.backend: Exjdb back end

exjdb.compiler: Instrumenting compilers

exjdb.frontend: Exjdb front end

exjdb.parser: Parser classes, generated by JFlex and CUP

exjdb.semant: Semantic classes for the abstract syntax trees, partially generated by ASTG

io: Input output classes

srmi: SRMI classes, the protocol that connects the front end and the back end

swing: Additional swing controls

swing.filetree: Filesystem browsing control

4.3. Debuggee Instrumenter

The process to instrument the source code involves several steps. Firstly, the source code needs to be read from disk. For further processing it is useful to have an abstract syntax tree to represent the program. A trio of scanner, parser, and abstract syntax tree generator create the syntax tree. Because the instrumented code needs to be compiled by a Java compiler the tree must be converted back to Java code after the instrumentation. This can be achieved by recursively traversing the tree and printing the corresponding Java code for each node. Because this is the inverse process of parsing, this is called unparsing. The module handling the unparsing is therefore called an unparser. Instrumentation can take place at two different places. Either between the parsing and the unparsing step or during the unparsing step. The first method is more complex because the instrumentation code must be expressed with the help of tree nodes which must then be inserted into the tree at the correct place. The second method however, is far simpler, printing additional Java code when the nodes are processed. Whenever a throw statement is encountered for example, some code is added that allows the debugger to intercept the exception. I decided to use the second method because of its simplicity. The design of the instrumenter is partly given by the tools that are used to create the scanner, parser and abstract syntax tree.

4.4. Back end

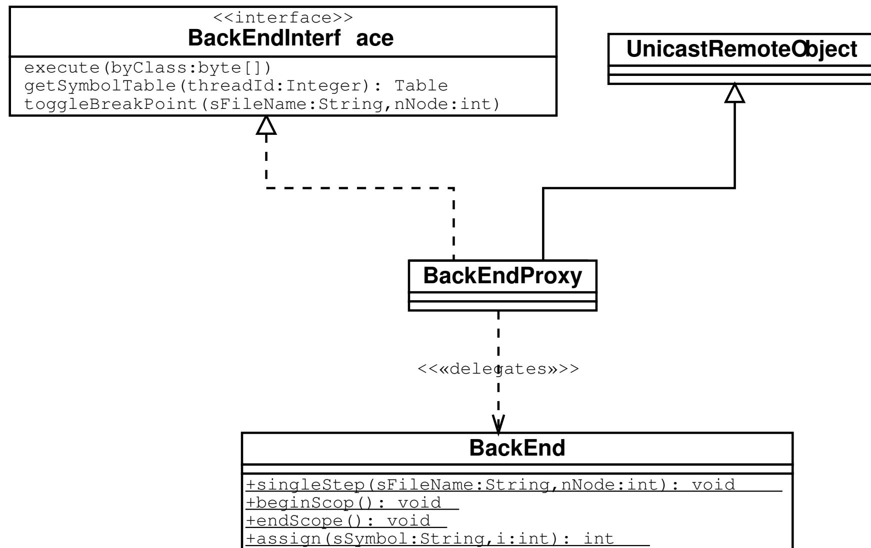
Since the instrumented classes do not know anything about the back end before instrumentation and the debugger does not know where the execution of the debuggee starts, there is no single place to create the back end. A singleton is needed to build the interface. In Java there are multiple possibilities to implement the singleton pattern. The simplest one is to make all attributes and methods in a class static. This way the instrumentation code will be as simple as this:

```
ch.devzone.exjdb.backend.BackEnd.singleStep( __FILE__, 0 );
```

Because this singleton can't implement the RMI interface a proxy class is needed to accomplish this. The proxy is just an ordinary class that implements the interface and delegates all method calls to the BackEnd class. All instrumented classes contain method calls like the above and are therefore referencing the BackEnd class, meaning that the BackEnd class is loaded just before the first instrumented class. The back end then immediately tries to establish a connection

with the debugger front end. Upon successful connection the front end transfers some data the back end needs for its operation, e.g. breakpoints, watchpoints. Whenever the back end hits a breakpoint or a watchpoint that resolves to true it informs the front end and waits for instructions from it (continue, single step, run to cursor, toggle breakpoint, ...).

Figure 4: Back end design



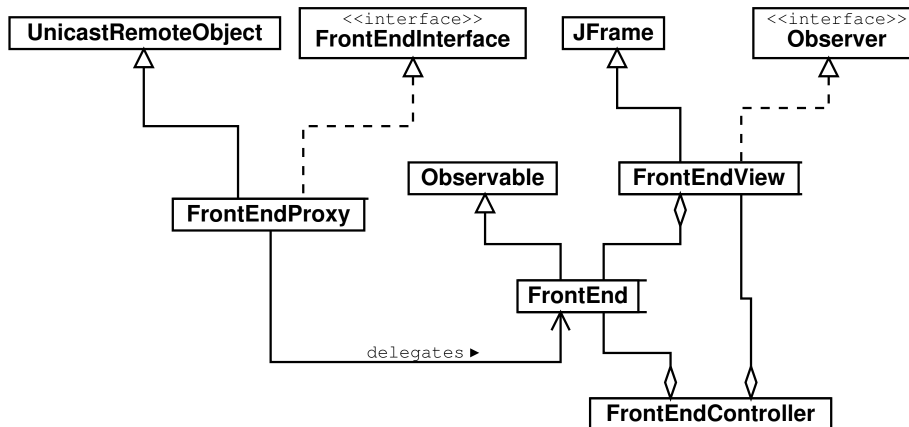
4.5. Front end

Through the front end the user controls the debugging session. To make this as simple as possible it provides a graphical user interface (GUI). The GUI classes are designed with the model view controller (MVC) paradigm. Communication between the model and the view is implemented using the observer pattern. The model is the observable and the view the observer. Figure 5 shows an overview of the front end design.

model: The model is implemented by the class FrontEnd. Since FrontEnd extends Observable it can't extend UnicastRemoveServer, too (Java does not support multiple inheritance). So like in the backend a proxy class is needed. The proxy extends UnicastRemoveServer, implements the remote interface and delegates all method calls to FrontEnd.

view: The view is implemented by the class FrontEndView. FrontEndView inherits from JFrame and implements the Observer interface. Upon construction the view attaches itself to the model, then the frame content is built using the user interface components. Through the Observer.update method the view receives changes in the model and updates the visual representation accordingly.

Figure 5: Front end design



controller: The controller is implemented by the class FrontEndController. FrontEndController first creates an instance of FrontEnd, then an instance of FrontEndView. Finally the controller registers its event handlers with the view. The user interface components call the event handlers whenever the user interacts with them. The event handlers in turn modify the model as necessary using its methods.

Figure 6 shows a screenshot of the GUI.

4.6. Protocol

Java provides a high level protocol to build distributed applications in an object oriented way, RMI. This protocol could be used to realize the communication between the debugger and the debuggee. The problem posed by RMI is that it creates one TCP connection per RMI server. This would severely limit the usability of the debugger:

- Restrictions through security managers
- Problems with firewalls
- Difficult or impossible to tunnel, e.g. with ssh

Exjdb uses its own RMI implementation called SRMI (Simple Remote Method Invocation) instead. SRMI uses a single TCP connection to avoid the problems listed above. It builds on sockets, object streams and reflection. As far as SRMI is implemented, it is interface compatible to RMI. SRMI is designed using the layers shown in table 1.

On the server side the object to be made available remotely is wrapped by a RemoteServer. On the client side it is represented by a RemoteStub. Whenever a method on the stub is called, the method call traverses the layers on the client side, is sent to the server on the lowest layer and is passed to the top layer on the server side. Each layer has a clear task:

Figure 6: Exjdb GUI

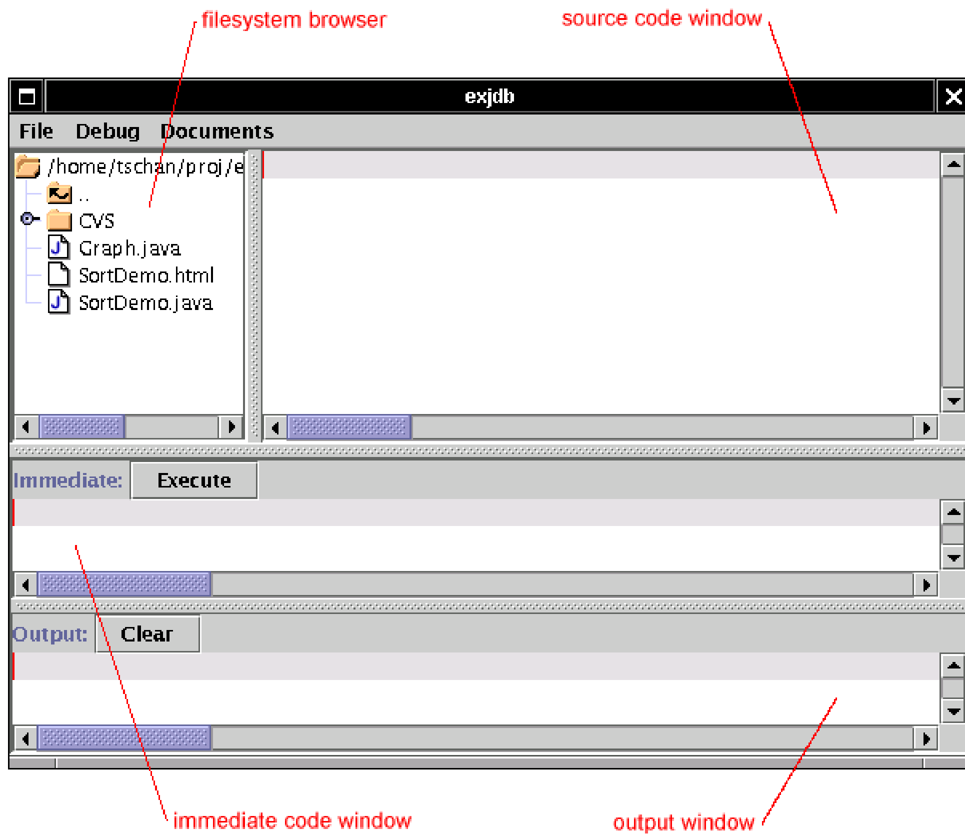


Table 1: SRMI Layers

RemoteStub/RemoteServer
Channel
Transport
MarshalInputStream/MarshalOutputStream
ObjectInputStream/ObjectOutputStream
Socket

RemoteStub/RemoteServer: Convert the method call to/from a serializable from

Channel: Provide a bidirectional communication channel

Transport: Multiplex/Demultiplex channels

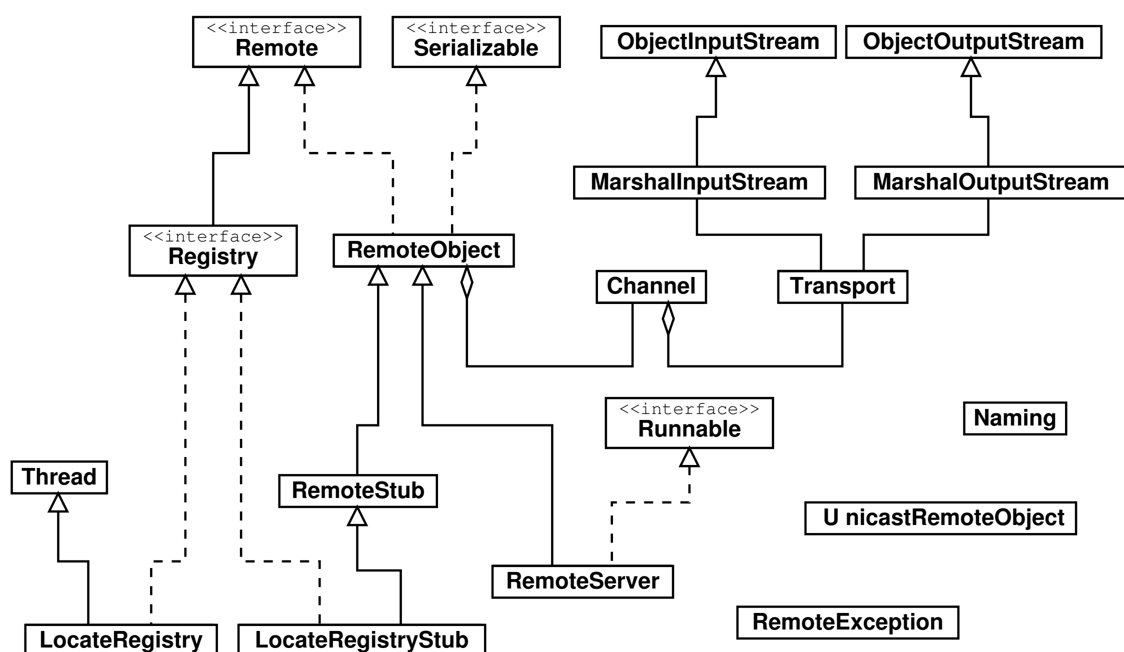
MarshalInputStream/MarshalOutputStream: Create RemoteServer and RemoteStub for remote objects traversing the streams

ObjectInputStream/ObjectOutputStream: Deserialize/serialize objects

Socket: Transport byte streams to other side

Other classes are needed to register and lookup remote objects. Figure 7 shows a class diagram of all SRMI classes.

Figure 7: SRMI class diagram



The LocateRegistry class is managing remote objects. The server is registering at least one object which the clients can then look up using a unique URL. As soon as the client has one remote object he can get others through it, so the registry is normally only used for bootstrapping. The class Naming further simplifies the lookup of objects by hiding the creation of the registry. A remote object can be looked up with a simple Naming.lookup call.

Unlike RMI, SRMI does not use skeletons. Instead the RemoteServer class is using reflection to call the desired method. The stubs are still needed, however. There is no generator, so they have to be written by hand, but this is straight forward.

5. Tools

This chapter describes the tools used to develop the prototype. In order to understand certain parts of the implementation it's necessary to have some basic knowledge of these tools.

5.1. JFlex

JFlex is a fast scanner generator for Java, like flex is for C, designed to work together with the CUP parser generator. The syntax of the scanner specification is very similar to the one of flex. The output is single .java file which implements the specified scanner. The scanner is completely independent of JFlex. JFlex is used to build the scanner which reads the Java source files during the instrumentation phase. A specification of a unicode preprocessor and a Java 1.2 scanner is already part of the JFlex distribution. After inserting a package statement into the two JFlex specifications they can be used without further modification.

5.2. CUP

CUP is a LALR(1) parser generator for Java similar to Bison and Yacc. CUP takes a parser specification containing a grammar and creates two .java files. One of them declares all terminals appearing in the grammar, the other one implements the specified parser. The parser depends on the java_cup.runtime package which contains a generic LALR parser driver. JFlex also comes with a CUP parser specification for Java 1.2.

5.3. ASTG

Some time ago I wrote an abstract syntax tree generator (ASTG) which works together with the CUP parser generator. ASTG reads a specification from which it creates a CUP specification with semantic actions, classes representing the abstract syntax tree and a visitor that can traverse this tree. Many ideas of ASTG come from the book "Modern Compiler Implementation in Java" [1].

The ASTG specification is an enhanced version of the CUP specification. It contains a grammar in Backus-Naur Form (BNF) and declarations needed for code generation. In addition to everything required by CUP the following declarations must be present in an ASTG specification:

- package of the abstract syntax tree classes
- package of the semantic classes (visitor)
- name of the root class
- name of the list root class (if list classes are to be used)
- name of the visitor class
- types of the classes

- where to create classes

The packages are declared right at the beginning of the file using apackage (abstract syntax tree classes), spackage (semantic classes) and ppackage (parser classes, appears in the CUP specification file as package declaration):

```
apackage ch.devzone.exjdb.absyn;
spackage ch.devzone.exjdb.semant;
ppackage ch.devzone.exjdb.parser;
```

The types and names of the classes are declared just before the start specification using the different class directives:

```
root class Node;
list root class NodeList;
visitor class NodeVisitor;
list class ImportDeclarations
flag class Modifiers;
```

The places where node classes to be created are specified by adding the class name in parenthesis behind the right hand side of a production:

```
if_then_statement ::=
  IF LPAREN expression RPAREN statement  (IfThenStatement);
```

The same class can appear in multiple right hand sides. For the complete grammar of the ASTG specification see appendix A.

The specification contains two sorts of type declarations. Some terminals, usually identifiers or literals, carry additional information from the scanner. These terminals must be declared with the type of the associated information (this is a CUP requirement). Then there are the productions with the class names of the nodes to be created. ASTG uses this data to derive the types of all right hand sides and all non-terminals using the following rules:

1. If a right hand side has a node class declared, the type of the right hand side is this class
2. If a right hand side consists of only one symbol with a type, the type of the right hand side is this type
3. If all right hand sides of a production have the same type, the type of the left hand side symbol is this type
4. If not all right hand sides of a production have the same type but all have class or interface types, a new interface is introduced and the type of the left hand side symbol is this interface

Table 2: Example productions for ASTG list class

import_declarations →	
import_declaration	(ImportDeclarations) |
import_declarations import_declaration	(ImportDeclarations)
import_declarations →	
ε	(ImportDeclarations) |
import_declarations import_declaration	(ImportDeclarations)

Rule 4 is the most complex one an needs some further explanation. The name of a interface is derived from the name of the left hand side symbol. All types of the right hand side are modified to extend or implement the new interface. If a production is not covered by one of this rules or if there is a non-terminal left without a type, this is an error and no abstract syntax tree can be generated. Terminals which don't have a type stay typeless.

The root class forms the root of the abstract syntax tree, as its name suggests. It implements the root interface whose name is derived from the root class. All node classes are subclasses of the root class and all interfaces are subinterfaces of the root interface. The root class contains attributes to store line and column of the source code corresponding to a node as well as an attribute to store which typeless terminals the production who created the node consists of. The following example shows a case where these terminal flags come in handy:

```
formal_parameter ::=
  type variable_declarator_id        (FormalParameter) |
  FINAL type variable_declarator_id  (FormalParameter);
```

The FormalParameter class defines symbolic constants for all eligible terminals, in this case FINAL is the only one. Because the presence of the FINAL terminal is stored in the flags attribute the two productions can share the same class thus simplifying the code. ASTG knows 3 types of node classes:

Regular: Regular classes have one attribute for each right hand side symbol that has a type. Their declaration is optional.

List: List classes store theirs children as a list and provide the usual list operations. They can be used for productions of the forms shown in table 2.

Flag: Flag classes only store the or linked terminal flags of their children. This is useful for productions like shown in table 3.

The node classes and the visitor class implement a visitor pattern. For a description of the visitor design pattern please see [2]. NodeVisitor is not an abstract class but already defines the

Table 3: Example productions for ASTG flag class

modifiers →	
modifier	(Modifiers) |
modifiers modifier	(Modifiers)
modifier →	
PUBLIC	(Modifier) |
PROTECTED	(Modifier) |
PRIVATE	(Modifier) |
STATIC	(Modifier) |
ABSTRACT	(Modifier) |
FINAL	(Modifier) |
NATIVE	(Modifier) |
SYNCHRONIZED	(Modifier) |
TRANSIENT	(Modifier) |
VOLATILE	(Modifier) |
STRICTFP	(Modifier)

logic to recursively traverse the abstract syntax tree. This allows the user to add his own semantic code by subclassing NodeVisitor and override only the overloads of visit he is interested in. The user never has to touch a machine generated file, which could be overwritten by the generator on its next run.

5.4. GNU Autotools

The GNU autotools are a collection of tools, namely autoconf, automake and libtool, which help to make software more portable. Autoconf creates configuration scripts which can automatically adapt source code packages to most UNIX like systems. Automake is a tool for automatically generating makefile templates for autoconf. Libtool is a script that allows to build and use shared libraries in a platform independent way.

The autotools are needed to compile the Jikes compiler (which is written in C++) as shared library. Since version 1.12 Jikes uses the Autoconf and Automake for compilation. Some small modifications and the introduction of Libtool allow Jikes to be compiled as shared library on nearly every platform. The use of this shared library will become visible later.

6. Implementation

This section describes the implementation details of the Exjdb prototype.

6.1. Utility classes

All classes related to Design by Contract can be found in the package ch.devzone.contract. The Contract class provides two method overloads for each contract type. If the contract condition is false, an exception is thrown, otherwise nothing happens. The optional Message appears in the contract violation exceptions and in the JavaDoc.

The package ch.devzone.util contains several utility classes used by exjdb:

Boxing: Explicit boxing and unboxing of Java primitive types

BreakPointTable: Manages a breakpoint table

ByteArrayClassLoader: Loads a class from a byte array

CombSort: Simple sort algorithm with $O(n \cdot log(n))$ complexity

Format: Provides C like sprintf method

Queue: Thread safe fifo queue

SymbolTable: Symbol table with scoping mechanisms

The ch.devzone.io package contains classes to run a named pipe over an existing srmi connection. They are used to redirect debuggee output to the debugger.

For a detailed description of these classes please see the JavaDoc.

6.2. GUI components

Exjdb uses some non standard GUI components.

StrutLayout: A simple but powerful layout manger. StrutLayout homepage: http://members.ozemail.com.au/~mpp/strutlayout/doc/overview.html

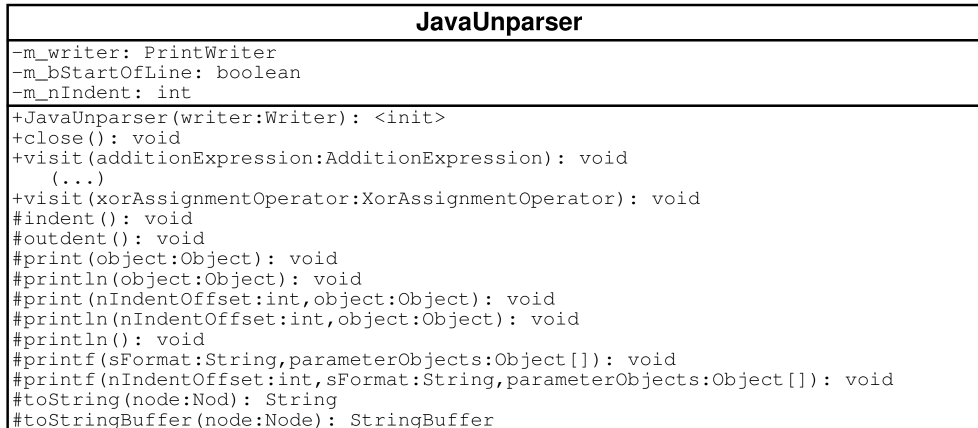
jEdit Syntax Package: Textarea control with syntax highlighting capabilities. jEdit Syntax Package homepage: http://syntax.jedit.org/

JFileTree: Implements a graphical file tree useful for navigating through file systems. The filesystem data is read on demand as soon as the corresponding node is expanded by the user. For more information please see the JavaDoc.

6.3. Debuggee instrumenter

JFlex, CUP and ASTG are used to create a parser which reads Java source code and builds an abstract syntax tree. ASTG also provides a suitable node visitor. Appendix B contains the grammar fed to ASTG. The debuggee instrumenter subclasses the node instrumenter to implement semantic actions which instrument the Java source code. The parser as well as the debuggee instrumenter are based on the Java Language Specification [3].

Figure 8: JavaUnparser class diagram



6.3.1. JavaUnparser class

JavaUnparser is a subclass of NodeVisitor which prints the Java source code corresponding to the tree to a writer. It contains print and println methods which take care of the correct indentation of the Java source code. There is also a toString method that converts a node and its subtrees to a string. JavaUnparser overrides the overloads of visit in NodeVisitor that need to print something and implements them with the help of the helper methods. Example:

```
public void action( IfThenElseStatement ifThenElseStatement )
{
  if ( ifThenElseStatement != null )
  {
    print( "if (" );
    action( ifThenElseStatement.m_e );
    println( ")" );
    indent();
    action( ifThenElseStatement.m_snsi );
    outdent();
    println( "else" );
    indent();
    action( ifThenElseStatement.m_s );
    outdent();
  }
}
```

Since the overloads that are not overridden recursively traverse the tree, it is now possible to print the Java source of the whole tree by calling action with the root of the tree as argument.

6.3.2. DebugeeInstrumenter class

The DebuggeeInstrumenter class is doing the actual instrumentation of the source code. It subclasses the JavaUnparser class in order to add the instrumentation code during output generation. The task of the instrumented code is to collect information and to communicate with the debugger back end. The class Instrumenter overrides certain methods of the class Unparser in order to add the debugger code. The following subsections describe the instrumentation steps in detail.

File name

The debugger always needs to know which code comes from which source file. Therefore he adds a constant called $FILE to each class. The value of the constant is the absolute path of the file that defines the class. Example:

```
public class SortDemo extends Applet
{
  private static final String $FILE =
    "/home/tschan/proj/exjdb/examples/sortdemo/SortDemo.java";
```

Single Steppingbackend

The instrumenter adds a piece of code in front of every statement which informs the debugger back end about the current location (source code line) of the current thread. The debugger back end in turn checks its state (single step mode, break point table, ...) and takes the necessary actions. Example:

```
ch.devzone.exjdb.backend.BackEnd.singleStep( $FILE, 376 );
swap( nFirst[ 0 ], nLast[ 0 ] );
ch.devzone.exjdb.backend.BackEnd.singleStep( $FILE, 377 );
m_graph.repaint( nFirst[ 0 ], nLast[ 0 ] );
```

Local variables

The problem with local variables is that they are stored on the stack which can't be read and modified directly with Java. The instrumenter solves this problem by rewriting the debuggee code so that all local variable are stored on the heap using single element arrays. The addresses of the arrays are stored by the back end which gives the debugger direct read/write access to all local variables in the instrumented code. Example:

```
int i = 0;
```

becomes:

```
int[] i = ( int[] ) ch.devzone.exjdb.backend.BackEnd.declare(
  "i", "int", new int[] { 0 } );
```

The declare method stores name, type and address of the array and returns the address back to the instrumentation code. All occurrences of the local variable are rewritten to use the array:

```
i++;
```

becomes

```
i[0]++;
```

Formal Parameters

At the beginning of a method the formal parameters are copied to the heap and thereafter threated like local variables:

```
private void swap(int $i,int $j)
{
  // frame tracking code, explained in the next section

  ch.devzone.exjdb.backend.BackEnd.declare( "this", "SortDemo", this );
  int[] i = ( int[] ) ch.devzone.exjdb.backend.BackEnd.declare(
    "i", "int", new int[] { $i } );
  int[] j = ( int[] ) ch.devzone.exjdb.backend.BackEnd.declare(
    "j", "int", new int[] { $j } );
```

The formal parameters are renamed to prevent naming collisions.

Frames and Scopes

In order to access the correct variables the debugger must be informed about stack frames and variable scopes. At the beginning and the end of each method code is inserted to track the frames:

```
ch.devzone.exjdb.backend.BackEnd.beginFrame();
try
{
  // original code
}
finally
{
  ch.devzone.exjdb.backend.BackEnd.endFrame();
}
```

The finally guarantees that the endFrame method is executed in any case. Similarly at the start and the end of a code block code to track the scope is added:

```
ch.devzone.exjdb.backend.BackEnd.beginScope();
try
{
  // original code
}
```

```
finally
{
  ch.devzone.exjdb.backend.BackEnd.endScope();
}
```

Accessors

Fields and methods may be unreachable by the debugger because they are private or protected. The instrumenter adds accessor methods to overcome this limitation. Each method gets an accessor for invocation and each field one for reading and one for writing (unless it is final). Numeric non final fields have 4 additional accessors for the preincrement, predcrement, postincrement and postdecrement operations. These are needed for the immediate code execution feature. Example:

```
private void swap(int $i,int $j)
{
  // code omitted
}
public void access0$swap(int i,int j)
{
  swap( i, j );
}



private int m_nWidth;

public int access1$m_nWidth()
{
  return m_nWidth;
}

public int access2$m_nWidth( int $value )
{
  return m_nWidth = $value;
}

public int access3$m_nWidth()
{
  return m_nWidth++;
}

public int access4$m_nWidth()
{
  return m_nWidth--;
```

```
}

public int access5$m_nWidth()
{
  return ++m_nWidth;
}

public int access6$m_nWidth()
{
  return --m_nWidth;
}
```

This pointer

To call the accessors the debugger needs the class and for non static members the "this" pointer. The instrumenter adds code to add the "this" pointer into the symbol table at all places where a new frame begins. Example:

```
ch.devzone.exjdb.backend.BackEnd.beginFrame();
try
{
  ch.devzone.exjdb.backend.BackEnd.declare( "this", "SortDemo", this );
```

If this code is inside a static method the value of the "this" pointer is set to null. This means that the debugger cannot access non static members when stopped inside a static method. But the method doesn't have access to them either, so this is ok.

6.4. Back End

6.4.1. BackEnd class

The BackEnd class collects and stores data about the debuggee. There is at most one BackEnd class in a Java virtual machine. Figure 9 shows a class diagram of BackEnd. Each back end has an id it receives when it attaches itself to the front end. The back end uses the id to identify itself to the front end. The back end can reside in one of four different states:

Suspended: The debuggee is suspended. This is the initial state and allows the developer to set breakpoints or do other preparations.

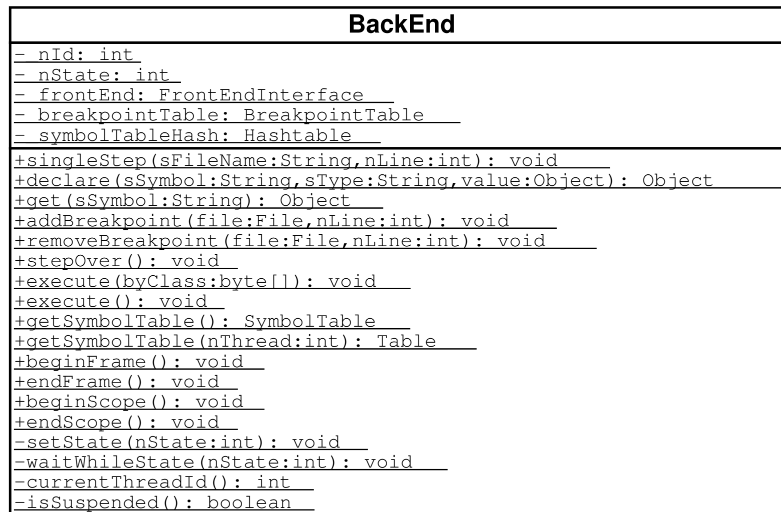
Running: All threads in the debuggee are running.

Breaked: A thread has hit a breakpoint and stopped.

Execute: The byte code in _byClass is ready to be executed in the currently stopped thread.

Most methods are used by the instrumentation code to inform the back end about the current state of the back end. These methods identify the thread reporting the changes by using

Figure 9: BackEnd class diagram



Thread.currentThread. The end of a thread is detected by the endFrame method when the last frame ends. All data regarding this thread are then discarded.

BackEnd also manages a breakpoint table. It is consulted in the singleStep method called by the instrumentation code. If a breakpoint is hit the execution of the current thread is stopped using Object.wait, the state switched to breaked and the front end is informed.

The execute(byte[]) method is called by the front end when the user wants to execute immediate code. The byte code is stored in _byClass and the state switched to execute. The stopped debuggee thread notifies the state change and executes the immediate code. The Java memory model guarantees that a single thread is unaffected of optimization side effects and always sees life values of all its variables and objects. For other threads looking at the same variables and objects these assumptions don't need to be true. Therefore its important that the immediate code is executed in the very thread the developer stopped. Otherwise the results would be unpredictable. For a good description of the Java memory model please see the book "Concurrent Programming in Java" [4].

6.4.2. BackEndProxy class

BackEndProxy plays the role of a SRMI server for the back end. The BackEnd class can't do this itself because there are no instances of it. Figure 10 shows a class diagram of BackEndProxy. The proxy delegates all remote method invocations to their static counterparts in BackEnd.

Figure 10: BackEndProxy class diagram

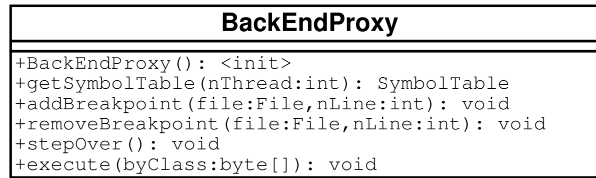
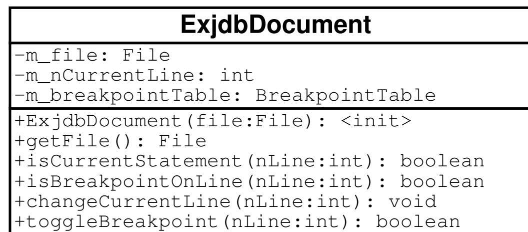


Figure 11: ExjdbDocument class diagram



6.5. Front End

6.5.1. ExjdbDocument class

ExjdbDocument represents a document of exjdb. A document consists of a source file, and information associated with this file. Currently this are the breakpoint table and the current line.

6.5.2. ExjdbHighLight class

ExjdbHighlight is responsible to visually represent breakpoints and the current line in the source text area. JEdit textarea highlighters are organized in a chain. ExjdbHighlight just draws the breakpoint and current line representation of the given line and calls the next highlighter in the chain. Figure 12 show a class diagram of ExjdbHighLight.

Figure 12: ExjdbHighLight class diagram

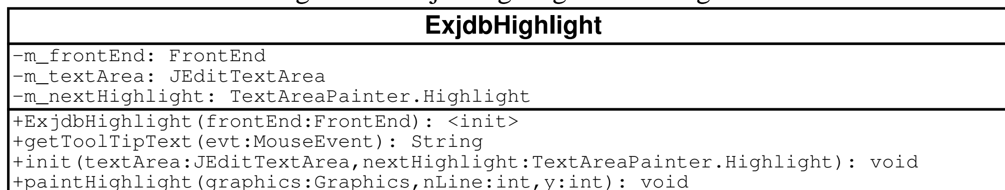
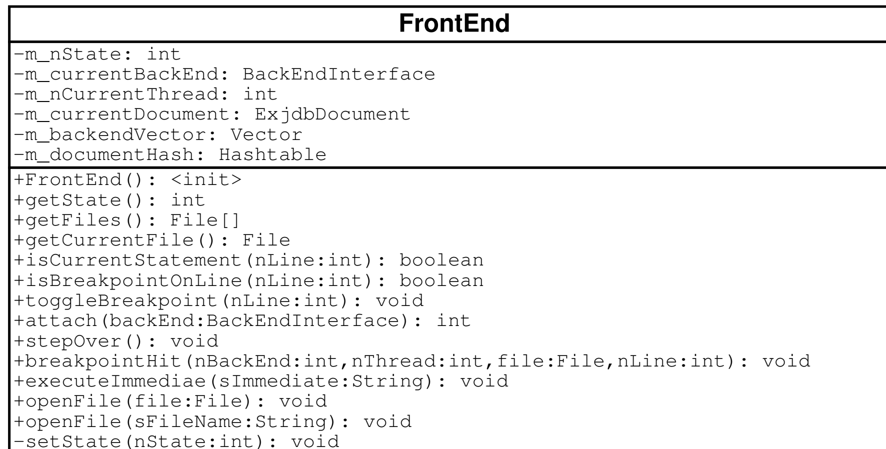


Figure 13: FrontEnd class diagram



6.5.3. FrontEnd class

The FrontEnd class implements the model and is the core of the front end. It can manage multiple back ends and multiple documents. The details of the document handling are implemented in the earlier mentioned ExjdbDocument class. FrontEnd knows three states:

Stopped: No back ends are connected, this is the initial state

Running: At least one back end is connected, all back ends are running

Breaked: A breakpoint has been hit, the corresponding thread is waiting

Communication between the model and the views is implemented using the observer pattern. This is why FrontEnd extends Observable. Whenever the state of the model is modified, all attached observers (views) are notified through the Observable.notifyObservers method.

FrontEnd also contains the logic to instrument and compile the immediate code. It puts a class and a method around the immediate code to get a complete compilation unit which is then passed to the immediate instrumenter mentioned in the next section. Afterward the instrumented code is compiled with Jikes. Finally the byte code is sent to the back end for execution in the debuggee.

6.5.4. FrontEndController class

FrontEndController is the first class of the front end that is instantiated. The controller first creates the model and the view. Then it connects the view to the model. Finally it registers its event handlers at the view which in turn registers them at the corresponding controls. The event handlers are implemented using function objects which are stored in private attributes of the class. Here's an example:

Figure 14: FrontEndController class diagram

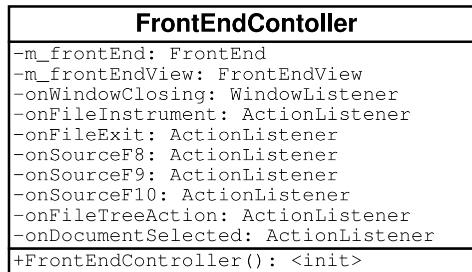
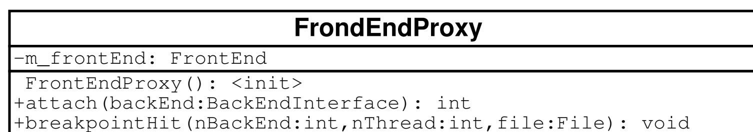


Figure 15: FrontEndProxy class diagram




```
private final ActionListener onExecute = new ActionListener()
{
  public void actionPerformed( ActionEvent actionEvent )
  {
    _frontEnd.executeImmediate( _frontEndView.getImmediate() );
  }
};
```

Here is some example code to register the event handler:

```
m_frontEndView.addSourceKeyBinding( "F8", onExecute );
```

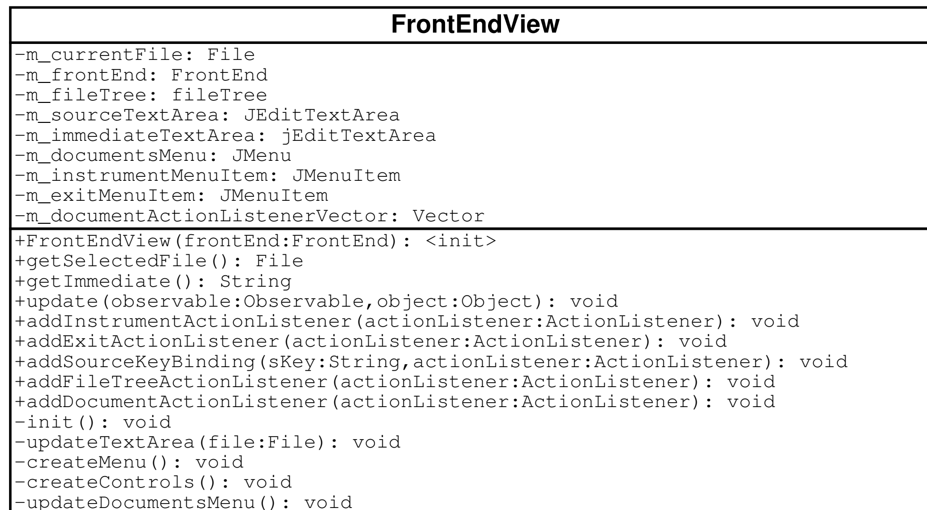
6.5.5. FrontEndProxy class

The FrontEnd class needs to be available remotely, so that the back end can communicate with it. But because FrontEnd already inherits from Observable it can't inherit from UnicastRemoteServer too. That's where FrontEndProxy comes in. It inherits from UnicastRemoteServer and implements all methods of FrontEndInterface by delegating them to the FrontEnd class. FrontEndProxy is also responsible for creating the remote registry if it's not already present and for registering the front end in the registry.

6.5.6. FrontEndView class

FrontEndView implements a view for the FrondEnd model using swing components. There's a member variable for each component which are used for communication with them. The

Figure 16: FrontEndView class diagram



view first registers itself at the model as an observer using the FrontEnd.addObserver method (inherited from Observable). It then creates and initializes the graphical components. Most of the public method are used by the controller to register it's event handlers and to communicate with the view.

6.6. Immediate Instrumenter

The code entered in the immediate window should behave as if it is part of the debuggee. It should read and write the variables and objects of the debuggee. The immediate instrumenter makes the necessary modifications to the immediate code to ensure this behavior. The modified immediate code uses the data collected by the back end and the extensions in the debuggee added by the debuggee instrumenter.

6.6.1. ImmediateInstrumenter class

ImmediateInstrumenter uses the same parser, abstract syntax tree, node visitor and unparser as the debuggee instrumenter and therefore extends JavaUnparser.

This Pointer

If the debuggee has been stopped in a non static method the following code sequence is inserted at the beginning to ensure the immediate code has access to the "this" pointer. Example:

```
Simple $this = ( Simple ) ch.devzone.exjdb.backend.BackEnd.get( "this" );
```

BackEnd.get accesses the current frame and scope of the thread which called the method. Since the immediate code will be executed in the same debuggee thread that was stopped BackEnd.get

will return the "this" pointer of this very thread.

Local Variables

After the eventual "this" pointer code statements are inserted to allow access to all local variables visible in the current debuggee scope. Example:

```
int[] i = ( int[] ) ch.devzone.exjdb.backend.BackEnd.get( "i" );
```

All occurrences of local variables are then modified the same way as in the debuggee by appending [0]. The immediate code can declare its own local variables. Local variables in the immediate code hide local variables with the same names in the debuggee code.

Fields

Accesses of fields of instrumented classes are modified to use the accessors added by the debuggee instrumenter. Example:

```
_n = 2;
System.err.println( _n );
```

becomes

```
$this.access2$_n( 2 );
System.err.println( $this.access1$_n() );
```

Method Invocations

If a method of an instrumented class is called the code is modified to use the accessor added by the debuggee instrumenter. This allows the immediate code to call all methods in the instrumented code no matter what their visibility is. Example:

```
swap( 0, 50 );
```

becomes

```
$this.access0$swap( 0, 50 );
```

The immediate code could declare its own methods but this is not implemented in the prototype.

6.7. The Protocol

This chapter discusses the details of the SRMI protocol.

6.7.1. Channel class

A channel represents a connection between a remote object and its stub. Each channel is part of a transport and has an id which uniquely identifies it within the transport. Reading and writing methods for different types are provided. The actual work is delegated to the transport.

Figure 17: SRMI collaboration diagram

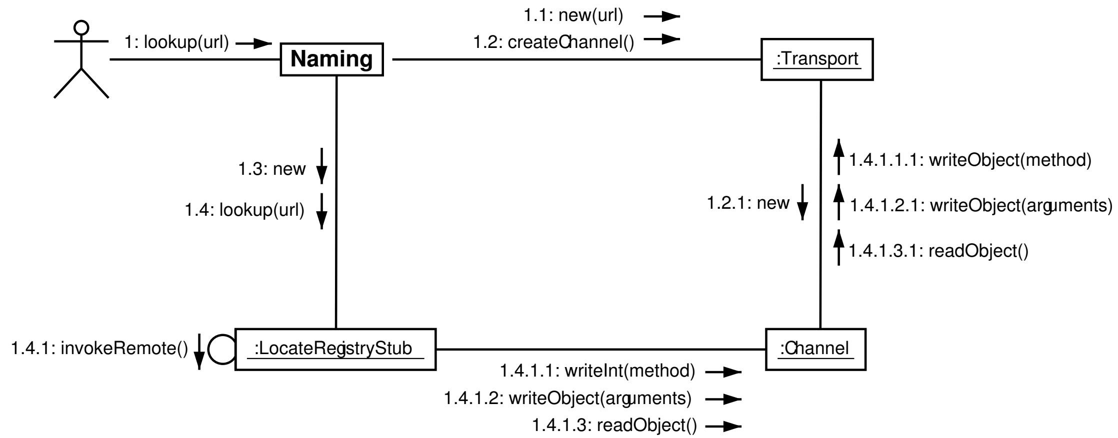


Figure 18: Channel class diagram

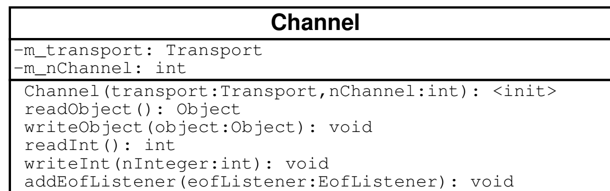


Figure 19: Transport class diagram

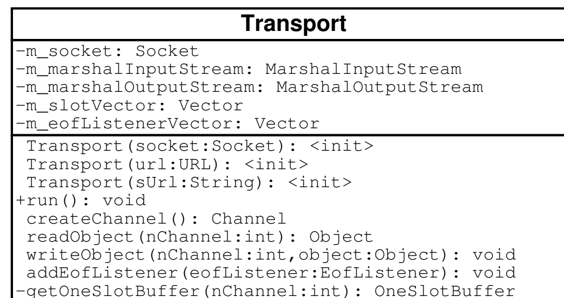


Figure 20: MarshalOutputStream class diagram

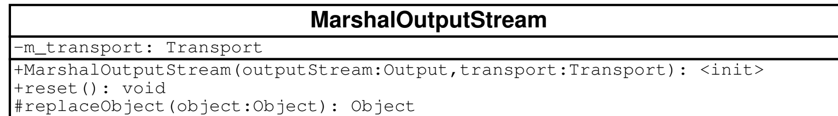


Figure 21: MarshalInputStream class diagram

6.7.2. Transport class

The transport class is managing all communication channels between a client and a server. Communication and multithreading details are hidden from the channels. A TCP stream socket is used to communicate with the other end of the transport. Two object streams (input and output) are used to abstract the socket connection. Whenever a channel wants to write an object, it calls a method in the transport passing its channel id and the object. The transport then simply writes the channel id followed by the object into the object output stream. Since the two write operations must be atomic the write method is synchronized. Each transport is using its own thread to read objects from the object input stream and to dispatch them to the channels. Since the channels are running in different threads, queues (one per channel) are used to pass the received objects from the transport to the channel. The queues are visible to the transport only.

6.7.3. MarshalOutputStream class

MarshalOutputStream is scanning all outgoing objects. If an object is identified as an SRMI receiver (subclass of UnicastRemoteObject) it is being replaced with its stub. Figure 20 shows a class diagram of MarshalOutputStream. Only trusted classes are allowed to replace objects in object streams. A class is trusted if it has been loaded by the system class loader. Let us assume an object of class A instantiates an object of class B. If class B is not loaded yet, the class loader that loaded class A is used to load class B, which is not necessarily the system class loader. Therefore the Transport class explicitly loads the MarshalOutputStream class with the system class loader.

6.7.4. MarshalInputStream class

MarshalInputStream is responsible to create the channel for remote objects after deserialization. Figure 21 shows a class diagram of MarshalInputStream. The createChannel method is called called by RemoteStub.readObject during deserialization.

Figure 22: RemoteObject class diagram

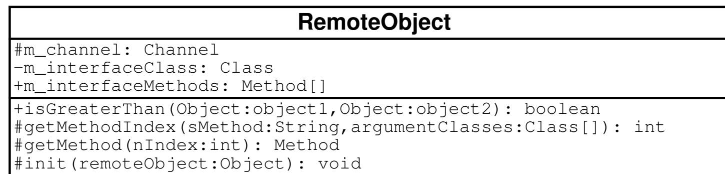
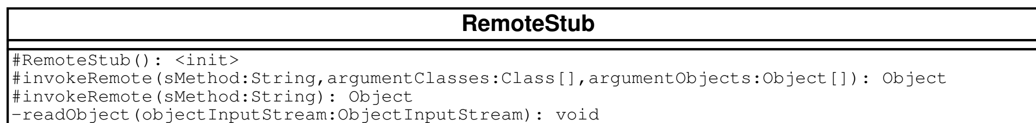


Figure 23: RemoteStub class diagram



6.7.5. RemoteObject class

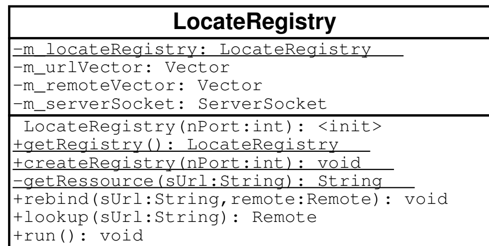
RemoteObject provides some basic functionality needed by *RemoteStub* and *RemoteServer*. Since *Method* objects are not serializable they must be converted to something that is serializable. Figure 22 shows a class diagram of RemoteObject. All exported methods are part of a remote interface that is implemented by both, the stub and the remote object. The *init* method now looks for the first interface of the stub and the remote object respectively which subclasses the interface *Remote*. It then stores the class object of this interface in *_interfaceClass* and all method objects of it in the array *_interfaceMethods*. Since the order of the methods in the array depend on the Java virtual machine, init sorts the methods by name and parameter types. Now each remote method is uniquely identified by the index in the array. The method methodToId allows a subclass to get the identifier of a method while idToMethod maps the identifier to a method.

6.7.6. RemoteStub class

RemoteStub represents a remote object on the remote side. Figure 23 shows a class diagram of RemoteStub. The *remoteInvoke* method writes the method identifier and the argument into the channel and then reads the result from it. The three operations on the channel must be atomic, otherwise arguments and results of different method calls could be mixed up. Each stub exclusively uses one channel, causing multiple concurrent method calls on the same stub being serialized. If necessary this could be improved by using multiple channels per stub and rotating them round robin for example. Another possibility would be to introduce dynamically allocated subchannels. The result can either be the return value of the method, or an exception it threw. If it's an exception it is wrapped into a RemoteException which is then thrown.

The *readObject* method is called upon deserialization of a stub. It reconstructs the information contained in the transient attributes by calling the inherited *init* method and creates a new

Figure 24: LocateRegistry class diagram



channel for the stub. Here is an important detail which is not visible at first. When something is written to a channel, the channel calls the *writeObject* method on the transport passing its identifier. The identifier is transported to the other end without translation, meaning that the channel identifiers on the client and the server must be synchronized. How is this ensured? The channel identifier is created by the createChannel method of the Transport class. It uses a simple incrementing counter. The channel on the server is created when a remote server objects passes through a MarshalOutputStream. On the client it is created upon deserialization as described before. Since *Transport* uses serializing object streams for transporting the data, concurrent channel creations can't interfere, the counters are always synchronized. Whenever a channel is created on the server side, a corresponding channel is created on the client side, with the same id.

6.7.7. UnicastRemoteObject class

In SRMI UnicastRemoteObject is an empty class. It is present to provide compatibility with RMI and is used by MarshalOutputStream to recognize remote servers that should be replaced with their stubs.

6.7.8. LocateRegistry class

LocateRegistry is the central SRMI registry. Figure 24 shows a class diagram of LocateRegistry. A server is registering one or more remote objects in the registry using *rebind*. A client then looks up a remote object in the registry using *lookup* in order to establish the first SRMI connection. For each object looked up the client receives a stub that implements a remote interface. The stub forwards all method calls over the established connection to the remote object. All further connections can then be created by either using the registry again or by a method of a remote object which returns another remote object.

6.8. Jikes modifications

Exjdb contains a debugger to create instrumented class files and to compile the immediate code. I took the Jikes compiler from IBM because its open source, runs on most platforms and is fast.

The modifications of Jikes are executed in two steps. First the build system is modified to build a shared library instead of a binary and then the code is modified to communicate with exjdb. On the Java side the class ch.devzone.compiler.Jikes is responsible for the communication with the Jikes compiler. It defines a callback method called by the compiler to retrieve the source code. The callback method in turn calls the the instrumenter to create the modified source code and return it to the compiler.

6.8.1. Building a shared library

Since version 1.12 Jikes is using the GNU build system, specifically autoconf, automake and libtool (see GNU Autoconf, Automake and Libtool [7]). This make it easy to compile it on various platforms and also simplifies building a shared library of Jikes instead of a binary.

First I modified configure.in by inserting the following lines after AC_PROG_CXX()

```
AC_LIBTOOL_WIN32_DLL()
AM_PROG_LIBTOOL()
AC_CHECK_LIB(stdc++,main)
```

The first line tells configure that it should build a DLL on Windows. Ommiting this line would cause the modified configure to fail on Windows. The second lines adds libtool support to the build system. The AC_CHECK_LIB macro searches for the standard c++ library. It is needed to successfully link a shared library that contains c++ code. Next I modified src/Makefile.am by replacing:

```
bin_PROGRAMS = jikes
```

with:

```
lib_LTLIBRARIES = libjikes.la
```

This tells automake to build a shared library named libjikes.so on Unix and jikes.dll on Windows. All other occurrences of the binary's name need to be changed, too. There is only one:

```
jikes_SOURCES
```

became:

```
libjikes_la_SOURCES
```

6.8.2. Communicating with exjdb

This modification is a bit more complex. I modified the build system again to auto detect the JNI header using the AC_PROG_JAVAH macro from the GNU autoconf macro archive http://www.gnu.org/software/ac-archive/. The macro didn't work correctly on Windows, which was easily to fix though. I modified the following line:

```
ac_machdep=`echo $build_os | sed 's,[-0-9].*,,'`
```

like this:

```
ac_machdep=`echo $build_os | sed 's,[-0-9].*,,' | sed 's,cygwin,win32,'`
```

Meanwhile, this change has been merged back into the official macro. To integrate the macro into the build system the following line has to added to the file acinclude.m4:

```
builtin(include,src/m4/ac_prog_javah.m4)
```

And this one to configure.in just after the AC_PROG_CXX directive:

```
AC_PROG_JAVAH()
```

7. Compilation

This section describes how to compile the various parts of exjdb.

7.1. Patching and compiling Jikes

To patch and compile Jikes you need the following tools:

- GNU patch 2.5.4
- GNU autoconf 2.53
- GNU automake 1.6.3
- GNU libtool 1.4.2
- gcc 2.95.3

Newer versions may also work, but have not been tested. gcc 2.96 is known not to work see 8.2.

First you need a cvs snapshot of Jikes 1.17. It can be checked out by issuing the following commands:

```
cvs -d :pserver:anoncvs@oss.software.ibm.com:/usr/cvs/jikes login
cvs -z3 -d :pserver:anoncvs@oss.software.ibm.com:/usr/cvs/jikes \
  co -rv1-17 jikes
```

When prompted for a password enter *anoncvs*. Switch to the jikes directory and apply the patches jikes-1.17-lib.patch and jikes-1.17-jni.patch:

```
patch -p1 <../exjdb/patch/jikes-1.17-lib.patch
patch -p1 <../exjdb/patch/jikes-1.17-jni.patch
```

Then execute these commands to update the build system:

```
libtoolize --force
sh autogen.sh
```

You can now build the Jikes shared library with the usual:

```
./configure
make
make install
```

7.2. Compiling exjdb

To compile exjdb you need:

- A UNIX like operating system or Windows with Cygwin
- bash 2.05 or later
- make 3.79.1 or later
- JDK 1.1 or later
- JFlex 1.3.5 or later
- CUP 0.10k or later
- ASTG
- autoj cvs snapshot 20021204 or later (for compilation of cvs snapshots only)

exjdb uses autoj build system for compilation. The steps required to build exjdb depend on the kind of source you compile from.

- cvs snapshot:

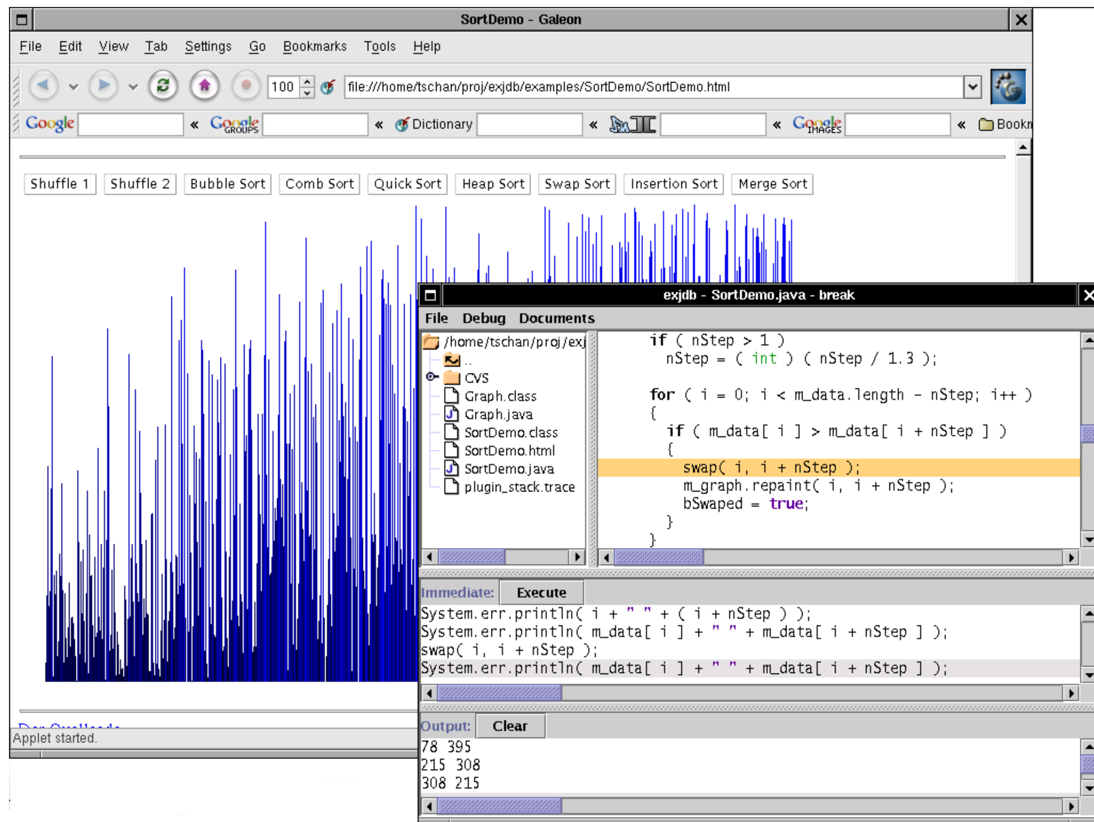
  ```
  ./bootstrap
  ./configure
  make lib
  make
  ```

- distribution tarball:

  ```
  ./configure
  make
  ```

The bootstrap script invokes autoj which creates a configure script, classpath.in, Makefile.in and run.in from configure.aj and Makefile.aj. A distribution tarball already contains these files. configure adapts the files generated by autoj to your machine. You can influence some of configure's decisions with command line parameters. For a description of available parameters type:

Figure 25: Exjdb debugging an applet running in the Galeon web browser




```
./configure --help
```

The make lib command downloads all jar files required by exjdb. Distribution tarballs already contain these files so make lib is not necessary in this case. Finally a simple make compiles exjdb. For a detailed description of autoj please visit http://autoj.sourceforge.net/.

After compiling exjdb you can run it by executing the following command in the exjdb root directory:

```
./run Main
```

The run script is also provided by autoj. Figure 25 shows exjdb in action.

7.3. Compiling the documentation

The main documentation has been edited in LyX. It contains various diagrams created with dia. LyX is mainly a what you see is what you mean (WYSIWYM) front end for LaTeX but can also create other formats like DocBook or HTML. To create a single PDF file of the documentation you need the following software:

- LyX 1.2.0 or later
- dia 0.88 or later
- teTeX 1.0.7 or later

First the diagrams need to be converted from dia to eps using dia. The eps files must then be converted to PDF using the epstopdf tool from the teTeX distribution. Finally LyX is used to create a single PDF file containing the text and the diagrams. LyX creates a temporary LaTeX file and used pdfTeX and BibTeX to create the PDF from it. To simplify things the makefile provides a target called *doc* which automates the documentation compilation process. The resulting PDF is fully scalable including the text in the diagrams.

Support to build the JavaDoc documentation is provided by the autoj build system. The javadoc target of the makefile scans all java files of the project to find all packages to document. Generated java files are excluded. If available the contract doclet is used to insert precondition, postconditions and invariants into the JavaDoc.

8. Problems and Limitations

This chapter describes solved and unsolved problems I encountered during this project as well as its limitations.

8.1. SRMI synchronization

At the beginning I had problems with the correct synchronization of the SRMI code. It dead locked regularly and had non deterministic behavior. I wasn't able to locate the source of the problems even after trying multiple different debugging approaches. Debugging synchronization problems is usually difficult and the debugger influences the behavior of incorrectly synchronized code. So I decided to refactor the code and introduce concurrent design patterns [4]. After the refactoring all problems were gone.

8.2. Jikes segfaults

The Jikes shared library was crashing with a segmentation fault on certain conditions. If you entered the following statement in the immediate window Jikes crashed:

```
System.err.println( "test: " + 1 );
```

Later it became apparent that this problem was related to the C and C++ compilers used to compile Jikes. The problem only occurred when Jikes was compiled with gcc 2.96 part of Red Hat Linux 7.x. This is either because of a binary incompatibility between gcc 2.96 and the compiler used to compile the JDK (most likely gcc 2.95.3) or because of problem with Jikes and the more modern gcc 2.96.

Another problem is that Jikes crashes if it calls a Java method and a exception occurs in the Java code. One solution to address this problem is to use Jikes as an executable rather than

a shared library and execute it using java.lang.Process. Communication between Exjdb and Jikes can then be realized with pipes (the output of Jikes would be piped to the corresponding java.lang.Process object) or sockets. Using pipes additionally allows to display Jikes error messages in a Exjdb window.

8.3. Named pipes

The named pipes used to redirect debuggee output to the debugger are terribly slow. I haven't figured out where so much time is lost. The problem could be related to SRMI, which is used to implement the pipes, or incorrect synchronization of the named pipe code.

9. Conclusion

Most Java virtual machines now support JDPA (Java Debugger Platform Architecture) and some good JDPA are available, too. But JDPA still suffers from the problem that it can't debug applications in their production environment. One of the first points in the documentation of a JDPA debugger usually tells the developer to disable hot spot engines, just in time compilers and other optimizing technologies commonly used in production environments. The reason is that the optimizations done by these technologies interfere with JDPA debuggers. Virtually any optimization can be applied as long as the Java Language Specification [3] is respected. The very same specification guarantees that exjdb is unaffected by these optimizations. Exjdb executes all operations in the thread of the debuggee it investigates. The Java Memory Model guarantees that a single thread does not see any effects of an optimization and always sees life values of all its variables.

The immediate execute feature of exjdb is very powerful if fully developed and something most other debuggers don't provide. Common features like watch points, variable inspection, exception catching and even not so common features like reverse single stepping could be implemented using the technologies presented in this document. But Exjdb has its limitations, too. It requires permission to use a custom classloader. Some environments like applets in web browser don't have these permissions by default. Additionally the instrumentation code may have unexpected side effects on the debuggee. I haven't observed any side effects during the development and testing of the prototype but can't guarantee that there are none. A deeper investigation would be needed to tell whether there are side effects and how they influence the debuggee.

Source based debugging isn't the ultimate solution but can complement other debugging solutions since it has different limitations.

A. Grammar for ASTG Specification Files

```
java_astg_spec →
    apackage_spec spackage_spec ppackage_spec import_list code_parts symbol_list
        precedence_list node_class_list start_spec production_list

apackage_spec →
    APACKAGE multipart_id SEMI |
    ε

spackage_spec →
    SPACKAGE multipart_id SEMI |
    ε

ppackage_spec →
    PPACKAGE multipart_id SEMI |
    ε

import_list →
    import_list import_spec |
    ε

import_spec →
    IMPORT import_id SEMI

code_part →
    action_code_part |
    parser_code_part |
    init_code |
    scan_code

code_parts →
    code_parts code_part |
    ε

action_code_part →
    ACTION CODE CODE_STRING opt_semi

parser_code_part →
    PARSER CODE CODE_STRING opt_semi
```

continue ...

... continue

```
init_code →
   INIT WITH CODE_STRING opt_semi

scan_code →
   SCAN WITH CODE_STRING opt_semi

symbol_list →
   symbol_list symbol |
   symbol

symbol →
   TERMINAL type_id declares_term |
   NON TERMINAL type_id declares_non_term |
   NONTERMINAL type_id declares_non_term |
   TERMINAL declares_term |
   NON TERMINAL declares_non_term |
   NONTERMINAL type_id declares_non_term

term_name_list →
   term_name_list COMMA new_term_id |
   new_term_id

non_term_name_list →
   non_term_name_list COMMA new_non_term_id |
   new_non_term_id

declares_term →
   term_name_list SEMI

declares_non_term →
   non_term_name_list SEMI

precedence_list →
   precedence_l |
   ε

precedence_l →
   precedence_l preced |
   preced

preced →
```

continue ...

... continue

```
  PRECEDENCE LEFT terminal_list SEMI |
  PRECEDENCE RIGHT terminal_list SEMI |
  PRECEDENCE NONASSOC terminal_list SEMI

terminal_list →
  terminal_list COMMA terminal_id |
  terminal_id

node_class_list →
  node_class_list node_class |
  ε

node_class →
  CLASS class_id SEMI |
  ROOT CLASS class_id SEMI |
  LIST CLASS class_id SEMI |
  LIST ROOT CLASS class_id SEMI |
  FLAG CLASS class_id SEMI |
  VISITOR CLASS class_id SEMI

start_spec →
  START WITH nt_id SEMI |
  ε

production_list →
  production_list production |
  production

production →
  nt_id COLON_COLON_EQUALS rhs_list SEMI

rhs_list →
  rhs_list BAR rhs |
  rhs

rhs →
  prod_part_list PERCENT_PREC term_id opt_class |
  prod_part_list opt_class

opt_class →
  LPAREN class_id RPAREN |
```

continue ...

... continue

ε

prod_part_list →
  prod_part_list prod_part |
  ε

prod_part →
  symbol_id opt_label |
  CODE_STRING

opt_label →
  COLON label_id |
  ε

multipart_id →
  multipart_id DOT ID |
  ID

import_id →
  multipart_id DOT STAR |
  multipart_id

type_id →
  multipart_id

terminal_id →
  term_id

term_id →
  symbol_id

new_term_id →
  ID

new_non_term_id →
  ID

nt_id →
  ID

symbol_id →

continue ...

... continue

ID

label_id →

ID

class_id →

ID

opt_semi →

SEMI |

ε

B. Java 2 Grammar

```
goal →
   compilation_unit

literal →
   INTEGER_LITERAL |
   FLOATING_POINT_LITERAL |
   BOOLEAN_LITERAL |
   CHARACTER_LITERAL |
   STRING_LITERAL |
   NULL_LITERAL

type →
   primitive_type |
   reference_type

primitive_type →
   numeric_type |
   BOOLEAN

numeric_type →
   integral_type |
   floating_point_type

integral_type →
   BYTE |
   SHORT |
   INT |
   LONG |
   CHAR

floating_point_type →
   FLOAT |
   DOUBLE

reference_type →
   class_or_interface_type |
   array_type

class_or_interface_type →
```

continue ...

... continue

name

class_type →
class_or_interface_type

interface_type →
class_or_interface_type

array_type →
primitive_type dims |
name dims

name →
simple_name |
qualified_name

simple_name →
IDENTIFIER

qualified_name →
name DOT IDENTIFIER

compilation_unit →
package_declaration_opt import_declarations_opt type_declarations_opt

package_declaration_opt →
package_declaration |
ε

import_declarations_opt →
import_declarations |
ε

type_declarations_opt →
type_declarations |
ε

import_declarations →
import_declaration |
import_declarations import_declaration

continue ...

... continue

```
type_declarations →
   type_declaration |
   type_declarations type_declaration

package_declaration →
   PACKAGE name SEMICOLON

import_declaration →
   single_type_import_declaration |
   type_import_on_demand_declaration

single_type_import_declaration →
   IMPORT name SEMICOLON

type_import_on_demand_declaration →
   IMPORT name DOT MULT SEMICOLON

type_declaration →
   class_declaration |
   interface_declaration |
   SEMICOLON

modifiers_opt →
   modifiers |
   ε

modifiers →
   modifier |
   modifiers modifier

modifier →
   PUBLIC |
   PROTECTED |
   PRIVATE |
   STATIC |
   ABSTRACT |
   FINAL |
   NATIVE |
   SYNCHRONIZED |
   TRANSIENT |
   VOLATILE |
```

continue ...

... continue

```
    STRICTFP

class_declaration →
    modifiers_opt CLASS IDENTIFIER super_opt interfaces_opt class_body

super →
    EXTENDS class_type

super_opt →
    super |
    ε

interfaces →
    IMPLEMENTS interface_type_list

interface_type_list →
    interface_type |
    interface_type_list COMMA interface_type

class_body →
    LBRACE class_body_declarations_opt RBRACE

class_body_declarations_opt →
    class_body_declarations |
    ε

class_body_declarations →
    class_body_declaration |
    class_body_declarations class_body_declaration

class_body_declaration →
    class_member_declaration |
    static_initializer |
    constructor_declaration |
    block

class_member_declaration →
    field_declaration |
    method_declaration |
    modifiers_opt CLASS IDENTIFIER super_opt interfaces_opt class_body |
    modifiers_opt CLASS IDENTIFIER super_opt interfaces_opt class_body SEMICOLON |
```

continue ...

... continue

interface_declaration

field_declaration →
modifiers_opt type variable_declarators SEMICOLON

variable_declarators →
variable_declarator |
variable_declarators COMMA variable_declarator

variable_declarator →
variable_declarator_id |
variable_declarator_id EQ variable_initializer

variable_declarator_id →
IDENTIFIER |
variable_declarator_id LBRACK RBRACK

variable_initializer →
expression |
array_initializer

method_declaration →
method_header method_body

method_header →
modifiers_opt type method_declarator throws_opt |
modifiers_opt VOID method_declarator throws_opt

method_declarator →
IDENTIFIER LPAREN formal_parameter_list_opt RPAREN |
method_declarator LBRACK RBRACK

formal_parameter_list_opt →
formal_parameter_list

formal_parameter_list →
formal_parameter |
formal_parameter_list COMMA formal_parameter

formal_parameter →
type variable_declarator_id |

continue ...

... continue

FINAL type variable_declarator_id

throws_opt →
throws

throws →
THROWS class_type_list

class_type_list →
class_type |
class_type_list COMMA class_type

method_body →
block |
SEMICOLON

static_initializer →
STATIC block

constructor_declaration →
modifiers_opt constructor_declarator throws_opt constructor_body

constructor_declarator →
simple_name LPAREN formal_parameter_list_opt RPAREN

constructor_body →
LBRACE explicit_constructor_invocation block_statements RBRACE |
LBRACE explicit_constructor_invocation RBRACE |
LBRACE block_statements RBRACE |
LBRACE RBRACE

explicit_constructor_invocation →
THIS LPAREN argument_list_opt RPAREN SEMICOLON |
SUPER LPAREN argument_list_opt RPAREN SEMICOLON |
primary DOT THIS LPAREN argument_list_opt RPAREN SEMICOLON |
primary DOT SUPER LPAREN argument_list_opt RPAREN SEMICOLON

interface_declaration →
modifiers_opt INTERFACE IDENTIFIER extends_interfaces_opt interface_body

extends_interfaces_opt →

continue ...

... continue

extends_interfaces |
ε

extends_interfaces →
EXTENDS interface_type |
extends_interfaces COMMA interface_type

interface_body →
LBRACE interface_member_declarations_opt RBRACE

interface_member_declarations_opt →
interface_member_declarations

interface_member_declarations →
interface_member_declaration |
interface_member_declarations interface_member_declaration

interface_member_declaration →
constant_declaration |
abstract_method_declaration |
class_declaration |
interface_declaration

constant_declaration →
field_declaration

abstract_method_declaration →
method_header SEMICOLON

array_initializer →
LBRACE variable_initializers COMMA RBRACE |
LBRACE variable_initializers RBRACE |
LBRACE COMMA RBRACE |
LBRACE RBRACE

variable_initializers →
variable_initializer |
variable_initializers COMMA variable_initializer

block →
LBRACE block_statements_opt RBRACE

continue ...

... continue

block_statements_opt →
  block_statements

block_statements →
  block_statement |
  block_statements block_statement

block_statement →
  local_variable_declaration_statement |
  statement |
  class_declaration |
  interface_declaration

local_variable_declaration_statement →
  local_variable_declaration SEMICOLON

local_variable_declaration →
  type variable_declarators |
  FINAL type variable_declarators

statement →
  statement_without_trailing_substatement |
  labeled_statement |
  if_then_statement |
  if_then_else_statement |
  while_statement |
  for_statement

statement_no_short_if →
  statement_without_trailing_substatement |
  labeled_statement_no_short_if |
  if_then_else_statement_no_short_if |
  while_statement_no_short_if |
  for_statement_no_short_if

statement_without_trailing_substatement →
  block |
  empty_statement |
  expression_statement |
  switch_statement |

continue ...

... continue

do_statement |
break_statement |
continue_statement |
return_statement |
synchronized_statement |
throw_statement |
try_statement

empty_statement →
SEMICOLON

labeled_statement →
IDENTIFIER COLON statement

labeled_statement_no_short_if →
IDENTIFIER COLON statement_no_short_if

expression_statement →
statement_expression

statement_expression →
assignment |
preincrement_expression |
predecrement_expression |
postincrement_expression |
postdecrement_expression |
method_invocation |
class_instance_creation_expression

if_then_statement →
IF LPAREN expression RPAREN statement

if_then_else_statement →
IF LPAREN expression RPAREN statement_no_short_if ELSE statement

if_then_else_statement_no_short_if →
IF LPAREN expression RPAREN statement_no_short_if ELSE statement_no_short_if

switch_statement →
SWITCH LPAREN expression RPAREN switch_block

continue ...

... continue

switch_block →
  LBRACE switch_block_statement_groups switch_labels RBRACE |
  LBRACE switch_block_statement_groups RBRACE |
  LBRACE switch_labels RBRACE |
  LBRACE RBRACE

switch_block_statement_groups →
  switch_block_statement_group
  switch_block_statement_groups switch_block_statement_group

switch_block_statement_group →
  switch_labels block_statements

switch_labels →
  switch_label
  switch_labels switch_label

switch_label →
  CASE constant_expression COLON
  DEFAULT COLON

while_statement →
  WHILE LPAREN expression RPAREN statement

while_statement_no_short_if →
  WHILE LPAREN expression RPAREN statement_no_short_if

do_statement →
  DO statement WHILE LPAREN expression RPAREN SEMICOLON

for_statement →
  FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
  for_update_opt RPAREN statement

for_statement_no_short_if →
  FOR LPAREN for_init_opt SEMICOLON expression_opt SEMICOLON
  for_update_opt RPAREN statement_no_short_if

for_init_opt →
  for_init |
  ε

continue ...

... continue

for_init →
statement_expression_list
local_variable_declaration

for_update_opt →
for_update
ε

for_update →
statement_expression_list

statement_expression_list →
statement_expression
statement_expression_list COMMA statement_expression

identifier_opt →
IDENTIFIER
ε

break_statement →
BREAK identifier_opt SEMICOLON

continue_statement →
CONTINUE identifier_opt SEMICOLON

return_statement →
RETURN expression_opt SEMICOLON

throw_statement →
THROW expression SEMICOLON

synchronized_statement →
SYNCHRONIZED LPAREN expression RPAREN block

try_statement →
TRY block catches
TRY block catches_opt finally

catches_opt →
catches

continue ...

... continue

ε

catches →

catch_clause

catches catch_clause

catch_clause →

CATCH LPAREN formal_parameter RPAREN block

finally →

FINALLY block

primary →

primary_no_new_array

array_creation_expression

primary_no_new_array →

literal

THIS

LPAREN expression RPAREN

class_instance_creation_expression

field_access

method_invocation

array_access

primitive_type DOT CLASS

VOID DOT CLASS

array_type DOT CLASS

name DOT CLASS

name DOT THIS

class_instance_creation_expression →

NEW class_type LPAREN argument_list_opt RPAREN

NEW class_type LPAREN argument_list_opt RPAREN class_body

primary DOT NEW IDENTIFIER LPAREN argument_list_opt RPAREN

primary DOT NEW IDENTIFIER LPAREN argument_list_opt RPAREN class_body

argument_list_opt →

argument_list |

ε

argument_list →

continue ...

... continue

expression
argument_list COMMA expression

array_creation_expression →
NEW primitive_type dim_exprs dims_opt
NEW class_or_interface_type dim_exprs dims_opt
NEW primitive_type dims array_initializer
NEW class_or_interface_type dims array_initializer

dim_exprs →
dim_expr
dim_exprs dim_expr

dims_opt →
dims

dims →
LBRACK RBRACK
dims LBRACK RBRACK

field_access →
primary DOT IDENTIFIER
SUPER DOT IDENTIFIER
name DOT SUPER DOT IDENTIFIER

method_invocation →
name LPAREN argument_list_opt RPAREN
primary DOT IDENTIFIER LPAREN argument_list_opt RPAREN
SUPER DOT IDENTIFIER LPAREN argument_list_opt RPAREN
name DOT SUPER DOT IDENTIFIER LPAREN argument_list_opt RPAREN

array_access →
name LBRACK expression RBRACK
primary_no_new_array LBRACK expression RBRACK

postfix_expression →
primary
name
postincrement_expression
postdecrement_expression

continue ...

... continue

postincrement_expression →
  postfix_expression PLUSPLUS

postdecrement_expression →
  postfix_expression MINUSMINUS

unary_expression →
  preincrement_expression
  predecrement_expression
  PLUS unary_expression
  MINUS unary_expression
  unary_expression_not_plus_minus

preincrement_expression →
  PLUSPLUS unary_expression

predecrement_expression →
  MINUSMINUS unary_expression

unary_expression_not_plus_minus →
  postfix_expression
  COMP unary_expression
  NOT unary_expression
  cast_expression

cast_expression →
  LPAREN primitive_type dims_opt RPAREN unary_expression
  LPAREN expression RPAREN unary_expression_not_plus_minus
  LPAREN name dims RPAREN unary_expression_not_plus_minus

multiplicative_expression →
  unary_expression
  multiplicative_expression MULT unary_expression
  multiplicative_expression DIV unary_expression
  multiplicative_expression MOD unary_expression

additive_expression →
  multiplicative_expression
  additive_expression PLUS multiplicative_expression
  additive_expression MINUS multiplicative_expression

continue ...

... continue

```
shift_expression →
    additive_expression
    shift_expression LSHIFT additive_expression
    shift_expression RSHIFT additive_expression
    shift_expression URSHIFT additive_expression

relational_expression →
    shift_expression
    relational_expression LT shift_expression
    relational_expression GT shift_expression
    relational_expression LTEQ shift_expression
    relational_expression GTEQ shift_expression
    relational_expression INSTANCEOF reference_type

equality_expression →
    relational_expression
    equality_expression EQEQ relational_expression
    equality_expression NOTEQ relational_expression

and_expression →
    equality_expression
    and_expression AND equality_expression

exclusive_or_expression →
    and_expression
    exclusive_or_expression XOR and_expression

inclusive_or_expression →
    exclusive_or_expression
    inclusive_or_expression OR exclusive_or_expression

conditional_and_expression →
    inclusive_or_expression
    conditional_and_expression ANDAND inclusive_or_expression

conditional_or_expression →
    conditional_and_expression
    conditional_or_expression OROR conditional_and_expression

conditional_expression →
    conditional_or_expression
```

continue ...

... continue

conditional_or_expression QUESTION expression COLON conditional_expression

assignment_expression →
conditional_expression
assignment

assignment →
left_hand_side assignment_operator assignment_expression

left_hand_side →
name
field_access
array_access

assignment_operator →
EQ |
MULTEQ |
DIVEQ |
MODEQ |
PLUSEQ |
MINUSEQ |
LSHIFTEQ |
RSHIFTEQ |
URSHIFTEQ |
ANDEQ |
XOREQ |
OREQ

expression_opt →
expression |
ε

expression →
assignment_expression

constant_expression →
expression

List of Figures

1. Use Case Diagram . . . 5
2. Subsystems . . . 7
3. Package Structure . . . 8
4. Back end design . . . 10
5. Front end design . . . 11
6. Exjdb GUI . . . 12
7. SRMI class diagram . . . 13
8. JavaUnparser class diagram . . . 19
9. BackEnd class diagram . . . 24
10. BackEndProxy class diagram . . . 25
11. ExjdbDocument class diagram . . . 25
12. ExjdbHighLight class diagram . . . 25
13. FrontEnd class diagram . . . 26
14. FrontEndController class diagram . . . 27
15. FrontEndProxy class diagram . . . 27
16. FrontEndView class diagram . . . 28
17. SRMI collaboration diagram . . . 30
18. Channel class diagram . . . 30
19. Transport class diagram . . . 30
20. MarshalOutputStream class diagram . . . 31
21. MarshalInputStream class diagram . . . 31
22. RemoteObject class diagram . . . 32
23. RemoteStub class diagram . . . 32
24. LocateRegistry class diagram . . . 33
25. Exjdb debugging an applet running in the Galeon web browser . . . 37

List of Tables

1. SRMI Layers 12
2. Example productions for ASTG list class 16
3. Example productions for ASTG flag class 17

References

[1] Appel, A. W. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK, Jan. 1998. http://www.cs.princeton.edu/~appel/modern/java/.

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995. http://hillside.net/patterns/DPBook/DPBook.html.

[3] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison Wesley, 1997. http://java.sun.com/docs/books/jls/.

[4] Lea, D. *Concurrent Programming in Java[tm], Second Edition: Design principles and Patterns*, 2nd ed. The Java Series. Addison Wesley, 1999. http://gee.cs.oswego.edu/dl/cpj/.

[5] Object Modeling Group. *Unified Modelling Language Specification, version 1.3*, Mar. 2000. OMG document formal/00-03-01. http://cgi.omg.org/cgi-bin/doc?formal/00-03-01.ps.gz.

[6] Oestereich, B. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language*, 4 ed. Oldenbourg, Muenchen, 1998. http://www.oose.de/publikationen/buchpublikationen/ooswuml/index.htm.

[7] Vaughan, G. V., Elliston, B., Tromey, T., and Taylor, I. L. *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Carmel, IN, USA, 2000. http://sources.redhat.com/autobook/.