

JRET: A Tool for the Reconstruction of Sequence Diagrams from Program Executions

Master's Thesis

Roland Voets

JRET: A Tool for the Reconstruction of Sequence Diagrams from Program Executions

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Roland Voets
born in Krimpen aan den IJssel, the Netherlands



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

JRET: A Tool for the Reconstruction of Sequence Diagrams from Program Executions

Author: Roland Voets
Student id: 1058932
Email: R.Voets@student.TUdelft.nl

Abstract

As opposed to static analysis, in which source code is inspected in order to increase program understanding, dynamic analysis concerns the actual execution of a program and the collection of runtime data. Several strategies to retrieve dynamic information exist, including source code instrumentation and the use of a customized debugger. Since the execution of a program is traced, one will be provided with detailed information on important aspects such as polymorphism and late binding. This detailed information, however, comes at a price. A major drawback of dynamic analysis is the vast amount of data produced. Visualization tools need to deal with this problem by, for example, applying certain abstractions in order for the information to become human-readable. In this research, we developed such a visualization tool that visualizes the execution of programs through sequence diagrams: JRET. We describe the strategy used, show how it attempts to tackle the aforementioned problem, and illustrate its contribution to program comprehension through a case study.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Drs. S.G.M. Cornelissen, Faculty EEMCS, TU Delft

Preface

First of all, I would like to thank my supervisor, Bas Cornelissen, for all his assistance and guidance throughout this research. Another thanks goes out to Andy Zaidman for his help with the CHECKSTYLE framework. Furthermore, I would like to thank Arie van Deursen and Leon Moonen for answering to my call as I was searching for potential graduation projects. Final thanks go out to Adrian Colyer, lead on the AspectJ project, and Rick Giles, one of the CHECKSTYLE developers, for taking the time to respectively answer my AspectJ and CHECKSTYLE related questions by e-mail.

Roland Voets
Krimpen aan den IJssel, the Netherlands
August 20, 2008

Contents

Preface	iii
Contents	v
List of Figures	vii
List of Tables	ix
Listings	xi
1 Introduction	1
2 Related Work	5
2.1 Approaches that reconstruct sequence diagrams	5
2.2 Approaches that reconstruct alternative diagrams	26
2.3 Surveys	26
2.4 Conclusion	26
3 Collecting Runtime Data from Java Software	29
3.1 Source code instrumentation	30
3.2 Bytecode instrumentation	31
3.3 Instrumentation of the runtime environment	31
3.4 Conclusion	32
4 JRET: The Java Reverse-Engineering Tool	33
4.1 Overview of the structure	33
4.2 Creating traces	36
4.3 Applying abstractions	41
4.4 Visualizing the data	48
4.5 JRET's research package	50

5	Case Study: Checkstyle	53
5.1	Introduction to Checkstyle	53
5.2	Checkstyle metrics	54
5.3	Understanding exports in Checkstyle	61
5.4	Understanding checks in Checkstyle	67
5.5	Discussion	81
6	Conclusions and Future Research	89
	Bibliography	93
A	Glossary	99

List of Figures

2.1	ISVis architecture.	6
2.2	A typical Scenario View in ISVis comprising the TMFD (center) and the information mural (right).	7
2.3	Shimba architecture.	9
2.4	SCED sequence diagram notation concepts.	11
2.5	An Execution view in Jinsight.	13
2.6	Result of zooming into the last phase of the Execution view.	14
2.7	JAVAVIS architecture.	15
2.8	Java Platform Debugger Interface (JPDA).	16
2.9	Sample JAVAVIS sequence diagram.	17
2.10	Scenario diagram metamodel.	19
2.11	Trace metamodel.	20
2.12	Sample sequence diagram.	21
2.13	Overview of the SDR framework.	22
2.14	Trace metamodel.	23
2.15	Scenario diagram metamodel.	24
2.16	Sample sequence diagram.	25
3.1	Symbolic steps from source code to sequence diagram model.	29
3.2	Schematical overview of normal compilation (left) versus aspect weaving (right).	30
4.1	The JRET GUI.	34
4.2	The JRET structure.	35
4.3	The trace metamodel.	38
4.4	The sequence diagram metamodel.	39
4.5	An example of detection and replacement of nested sequences in [23].	44
4.6	The result of previous example visualized using JRET.	45
4.7	A sample sequence diagram in SEQUENCE.	49
4.8	A sample sequence diagram in SDEDIT.	50
4.9	Pattern recognition in SDEDIT.	50

5.1	Settings when running JRET on CHECKSTYLE.	55
5.2	A sequence diagram in SEDIT that does not need abstractions.	58
5.3	The sequence diagram in Figure 5.2 when using SEQUENCE.	59
5.4	XML result of an audit using Hello.java.	62
5.5	Sequence diagram of the testAddError test case.	65
5.6	HTML result of an audit using Hello.java.	66
5.7	Visualization of the UncommentedMainCheckTest.testDefaults test case.	69
5.8	process(File) internal message without control flow.	70
5.9	process(File) internal message with its control flow.	70
5.10	Calls in the control flow of the process(FileArray) method call performed by the Checker object on the NewLineAtEndOfFileCheck object.	72
5.11	The walk(DetailAST, FileContents) method call including its message number.	73
5.12	Settings for studying the walk(DetailAST, FileContents) method call in more detail.	74
5.13	The walk(DetailAST, FileContents) method call in more detail.	75
5.14	The endsWithNewline(RandomAccessFile) method call in more detail.	77

List of Tables

5.1	Key statistics for the CHECKSTYLE case study.	56
5.2	The number of calls after the consecutive application of automatic abstractions to a selection of test cases.	57
5.3	The effectiveness of various abstraction techniques when separately applied to the original trace.	60
5.4	The number of calls after the consecutive application of automatic abstractions to the XMLLoggerTest test cases.	63
5.5	The number of calls after the consecutive application of automatic abstractions to five random test cases.	69

Listings

3.1	An example tracer in AspectJ.	31
5.1	Custom pointcuts to trace CHECKSTYLE.	55
5.2	Simple Java file Hello.java.	61
5.3	Part of a metrics file (stack depth limited to 7).	71
5.4	Part of a metrics file (stack depth limited to 6).	71
5.5	Methods of the PackageDeclarationCheck class.	78
5.6	The process method of the NewlineAtEndOfFileCheck class.	79

Chapter 1

Introduction

Over the past two decades, particularly since the introduction of the object-oriented programming paradigm in mainstream software application development in the early 1990s, software systems have grown in size and complexity. As a consequence, maintaining such systems has become an increasingly time-consuming and expensive process. It is for this reason that the understanding of the structure and behavior of software systems has become more and more important. In order to fully comprehend an object-oriented application, proper documentation should be available. Nevertheless, it is not unusual that no documentation exists and if it does, it is often either of poor quality or out of date. An adequate solution in this regard is the use of reverse-engineering techniques to reconstruct design artifacts. Two variants can be distinguished: reverse-engineering based on static analysis and dynamic analysis. Whereas modern CASE tools usually support the former, i.e., statically reconstructing design artifacts (e.g., Rational Rose [55] and Together [68]), the latter is much less frequently found.

Static analysis concerns analyzing a system's static structure or capturing a system's behavior by means of inspecting the code. A comparison of approaches for the reconstruction of static structures based on static analysis is presented in [38]. Various approaches to statically capture a system's behavior are given in [58, 37, 59]. Static analysis can be applied to both procedural and object-oriented software systems. However, when it comes to OO systems, statically reverse-engineering and understanding the behavior is more difficult compared to their procedural counterparts. This is because it can be troublesome or impossible to ascertain the runtime type of an object reference by using source code alone due to OO features such as inheritance, polymorphism and dynamic binding. As a consequence, determining beforehand which method is going to be executed can be a serious problem.

Dynamic analysis, on the other hand, concerns the actual execution of a system and the collection of runtime data. As such, runtime objects can be detected, thus making it possible to expose occurrences of polymorphism and late binding in contrast to static analysis. The produced execution traces contain very detailed information on how a system operates. This detailed information, however, comes with a serious downside. When considering object-oriented systems, a typical execution involves many different objects communicating

with each other. The vast number of interactions yields traces that quickly tend to become relatively complex for the reconstruction of comprehensible design artifacts. For example, when generating interaction diagrams (e.g., UML sequence diagrams [44] or UML collaboration diagrams [43]) from these traces, some form of abstraction has to be applied in order for the diagrams to become human-readable. Another drawback of dynamic analysis is that its result depends on how a system is executed. It is therefore necessary to define appropriate execution scenarios.

UML sequence diagrams are a potentially useful means to visualize a system's behavior [50, 60]. They are easy to read because the chronological order is intuitive. However, as already stated, dynamically reverse-engineered sequence diagrams tend to become too large if no abstractions are applied. Sequence diagrams are usually derived from use cases, but they can be derived from scenarios as well. In the latter case, they are often referred to as scenario diagrams. Consequently, a scenario diagram is a somewhat simplified version of a sequence diagram in that it depicts a specific scenario, rather than a generalization of multiple scenarios like in a use case.

In this thesis, we elaborate on a tool that is able to reconstruct sequence diagrams using dynamic analysis. We have developed it during this research and named it JRET, which stands for Java Reverse-Engineering Tool. To overcome the size problem, we implemented several abstraction techniques which are explained in chapter 4. As far as the execution of the system-under-analysis is concerned, we use JUNIT test cases to represent the different input scenarios. This choice is based on the following:

- JUNIT tests are a useful starting point for program comprehension [13]. The (unit) tests effectively decompose a system's functionalities. Moreover, input scenarios come for free when a test suite is available.
- Test cases are advocated as a form of documentation in Agile Development methods such as eXtreme Programming (XP) [47, 15, 18]. Practically the same goes for Test-Driven Development (TDD) which is related to XP [5].

The process of dynamic reconstruction of sequence diagrams can be divided into the following stages:

1. Creating traces
2. Applying abstractions
3. Visualizing the data

We dissect and describe our tool in terms of these three stages. Next, we illustrate its contribution to program comprehension through a case study. This case study includes a quantitative as well as a qualitative experiment using a widely known open source program: CHECKSTYLE [11]. The quantitative experiment involves gathering data on the need for abstractions, as well as a measure of the effectiveness of various abstraction techniques. In

the qualitative experiment, on the other hand, we demonstrate how JRET facilitates software maintenance. Specifically, we illustrate how JRET simplifies the implementation of two new features in the CHECKSTYLE framework.

Research questions

With this research we seek to answer the following research questions:

- How urgent is the need for abstractions?
- Which abstraction techniques are effective?
- Are sequence diagrams generated by JRET useful for software maintenance?

The answers to these questions are presented in chapter 6.

Structure of the thesis

This thesis is organized as follows. The next chapter describes related work in terms of existing approaches to dynamically reconstruct (sequence) diagrams. Chapter 3 outlines techniques for the collection of runtime data from Java software. Chapter 4 dissects and describes JRET. A case study is presented in chapter 5 and, finally, we conclude our work and give directions for future research in chapter 6.

Chapter 2

Related Work

There is a great deal of ongoing research regarding the visualization of program behavior. Various approaches have been developed to capture a system's behavior through static analysis [58, 37, 59, 42]. As far as dynamic analysis is concerned, different existing approaches are described in the following sections. Section 2.1 outlines ten existing approaches to reconstruct sequence diagrams from a running program. Rather than elaborating them all in more detail, we made a selection of six approaches in order to prevent this section from becoming too bulky. We picked the following: ISVis, Shimba, Jinsight, JAVAVIS, the approach by Briand et al., and the SDR framework. These were chosen based on the following criteria:

- They were all originally designed to operate on the popular object-oriented programming languages Java or C++ [27, 10].
- The techniques have been published.
- The authors clearly state what methods to collect runtime data have been used, and what abstraction techniques have been implemented.

In section 2.2, we briefly summarize three existing approaches that reconstruct alternative diagrams using dynamic analysis. In section 2.3, references to two surveys are given which evaluate multiple visualization tools. Finally, the chapter is concluded in section 2.4.

Furthermore, although beyond the scope of this research, some approaches are particularly interested in concurrent behavior, i.e., threads [45, 41, 19]. Yet other papers specifically deal with distributed systems [65, 8].

2.1 Approaches that reconstruct sequence diagrams

2.1.1 ISVis

The Interaction Scenario Visualizer (ISVis), designed and implemented by Jerding et al., combines static and dynamic analysis [30]. Using ISVis it is possible to browse and analyze

event traces derived from program executions. In order to accomplish this, ISVis provides multiple views which are capable of displaying vast amounts of interaction data and for making abstractions over them. Among these views is the so-called *scenario view* comprising the *information mural* [31] and the *temporal message-flow diagram (TMFD)* [12], the latter being quite similar to a sequence diagram. In order to produce traces source code instrumentation is used. ISVis was originally designed to operate on programs written in C++ (i.e., object-oriented software), however, it has shown to be able to handle applications written in C (i.e., procedural software) as well. Figure 2.1 depicts the ISVis architecture.

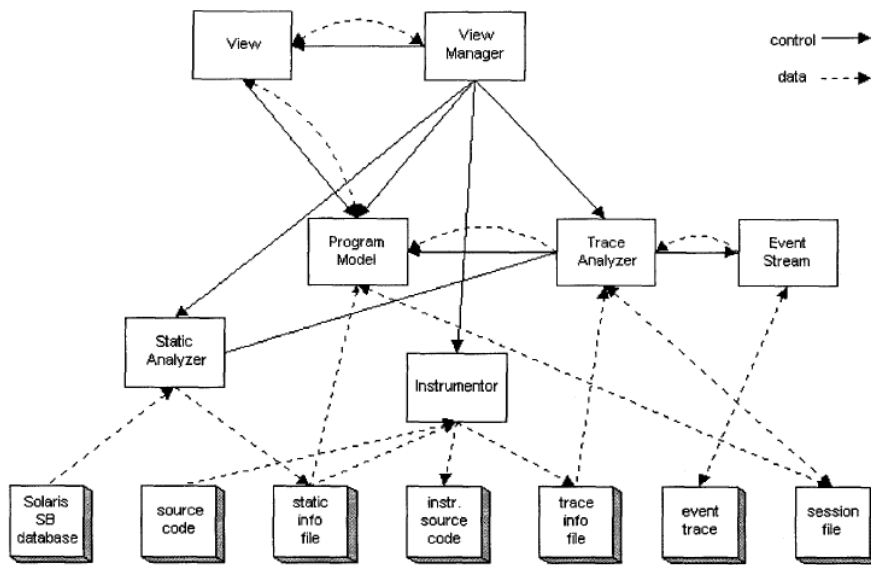


Figure 2.1: ISVis architecture.

As can be seen in the figure, the *Static Analyzer* reads the Source Browser database files produced by Solaris compilers and generates a *static information file*. This static information file, together with the source code and information supplied by the analyst about which actors should be instrumented (specified in the *trace information file*), is passed to the *Instrumentor* in order to generate instrumented source code. The user then has to compile the instrumented source code (i.e., externally to the ISVis tool) and execute the system using relevant test data. Consequently, event traces are generated. Afterwards, the *Trace Analyzer* in ISVis converts these event traces into so-called *interaction scenarios* which are stored in the *Program Model*.

In ISVis, an *actor* is a syntactically identifiable program unit (e.g., a function, an object, or a data item) which can consist of other actors itself. In the latter case it is called a *composite actor*. An *event* is a discernable unit of program execution (e.g., the invocation of a method, function return, object creation/deletion, data reference, or a specific action specified by the analyst). An *interaction* is a relation between an event and one or more actors.

Likewise, an *interaction scenario* is a sequence of interactions. As these interaction scenarios are created by the Trace Analyzer, the actors involved are also added to the Program Model. The user is then able to use the views of the Program Model to perform the analysis. A Program Model can be saved to disk in terms of a session file for later use.

Tracing

ISVis uses source code instrumentation by a Perl script in order to produce runtime information [12, 70]. In [32] Jerding et al. present a data structure, the *directed acyclic graph (DAG)*, for the internal representation of traces. This data structure allows ISVis to scale up to very large event traces. It is a compact form of the call tree (which represents a trace of method calls) in that it omits duplicate subtrees.

Views in ISVis

In ISVis two views can be distinguished: the *main view* and the *scenario view*. The main view lists the actors (including user-defined components, files, classes, and functions), interactions as well as interaction scenarios in the Program Model. Using this view the user is able to assign colors to actors or interactions. Moreover, it contains a shell for textual information i/o. The main view can be used to open a scenario view for each interaction scenario in the model. Such a scenario view is depicted in Figure 2.2.

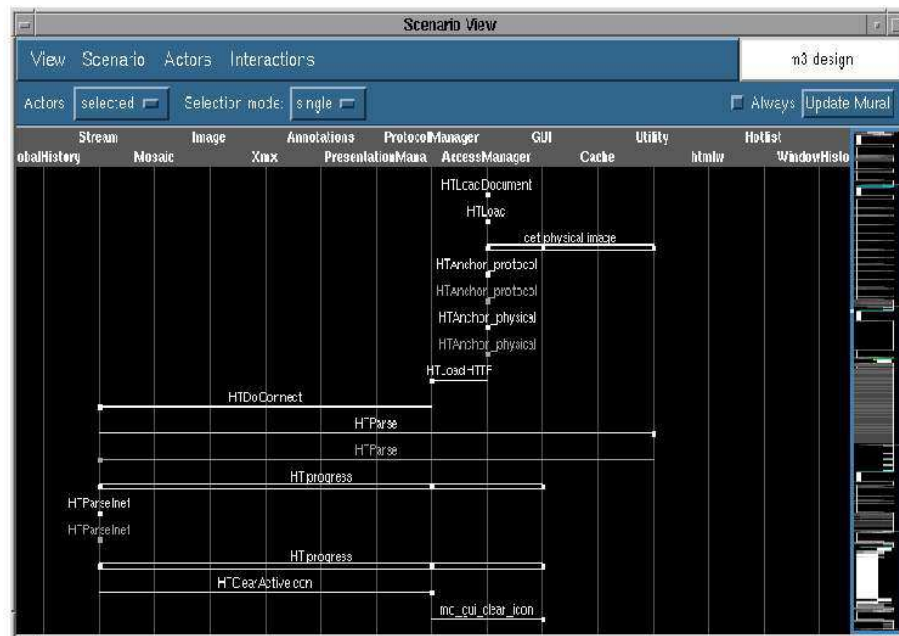


Figure 2.2: A typical Scenario View in ISVis comprising the TMFD (center) and the information mural (right).

As can be seen, a scenario view consists of two diagrams: the information mural (on the right) and the temporal message flow diagram (center) [31, 12]. The latter is quite similar to a sequence diagram and is sometimes referred to as an interaction diagram, message sequence chart (MSC), or event-trace diagram. The columns in such a TMFD represent the actors and the horizontal lines represent the interactions from source to destination actor [12].

The information mural, on the other hand, depicts a global overview of the scenario. It can be used to navigate through the interactions in the scenario and has proven to be able to effectively visualize overviews of scenarios containing hundreds of thousands of interactions [31]. In order to accomplish this, it uses some sort of anti-aliasing techniques. When some data in the information mural is selected, the particular data will be visible in the TMFD and, hence, can be studied in more detail.

Using the information mural the analyst is able to observe various phases in the interaction scenario including repetitive visual patterns (i.e., interaction patterns in the target program). Colors assigned to interactions in the main view are visible in the information mural as well. Likewise, the same goes for interactions that have been selected in the main view. In fact, both views have a subject-view relationship with the Program Model such that any selection or modification done in one view is immediately reflected in the other.

Abstractions

The scenario view provides means to apply abstractions to the visualized data. Actors in the scenario can be grouped by containing file, class, or component actors. For example, the user can decide to hide the classes that belong to the same subsystem and only show the interactions between this subsystem and the other components of the trace. Consequently, the resulting diagrams will contain less actors in total. Moreover, the user is able to define new scenarios which are added to the Program Model by selecting one or more (i.e., a sequence of) interactions. All occurrences of this particular sequence of interactions in the original scenario are then replaced by a reference to the newly defined scenario. These *subscenarios* are visualized through rectangles which contain all actors involved in the particular scenario.

Furthermore, ISVis provides means to detect *interaction patterns* using an algorithm similar to regular expression matching. Given an interaction pattern, the user can search in the scenario for an *exact match* (actors and interactions match exactly), an *interleaved match* (all interactions in the pattern occur exactly, but others may be interleaved), a *contained exact match* (actors in the scenario contain the actors in the pattern, and the interactions occur in exact order), and a *contained interleaved match*. A nice feature in this regard is the possibility to use wildcards to specify actors, thus allowing the user to formulate more general search queries. Additionally, instead of specifying a search query yourself, it is possible to let ISVis look for repeated sequences of interactions that occur in the scenario.

2.1.2 Shimba

Shimba is a reverse-engineering environment which combines static and dynamic analysis [66]. The static information is obtained by extracting data from Java bytecode using a tool called JExtractor. The extracted information is visualized as directed graphs and analyzed with the Rigi reverse-engineering tool [48]. The runtime information is generated by running the target system under a customized SDK debugger, namely JDebugger. Finally, the generated information is viewed as an UML sequence diagram using the SCED tool [39]. Using SCED, it is possible to automatically extract statechart diagrams from the obtained sequence diagrams. Figure 2.3 depicts the Shimba architecture.

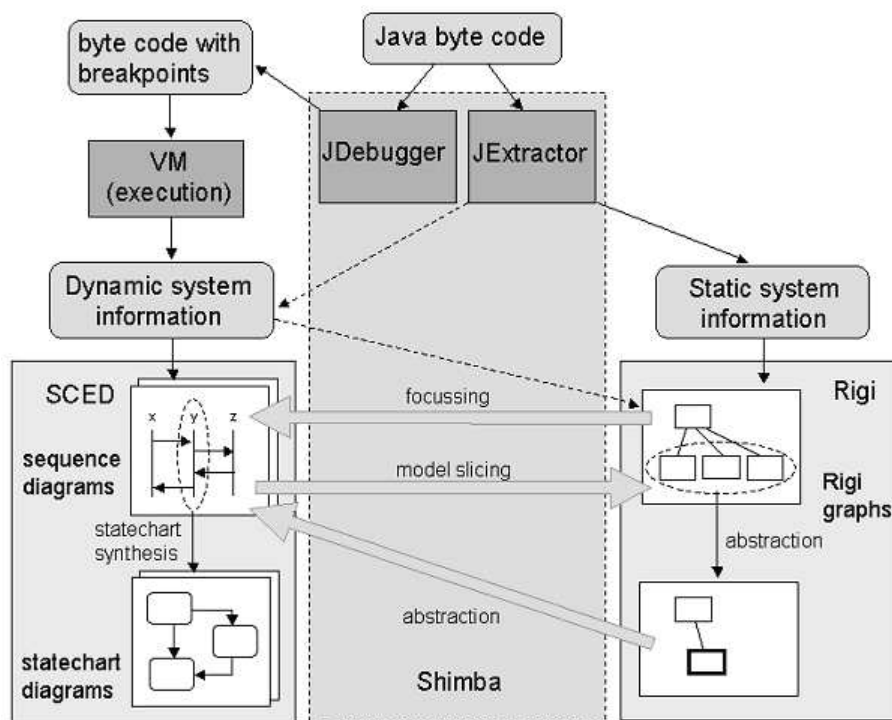


Figure 2.3: Shimba architecture.

As can be seen in the figure, Shimba uses Java bytecode as a starting point for the reverse-engineering tasks. According to Systä et al. this has three advantages:

- The program's source code does not have to be available.
- As Shimba combines static and dynamic analysis, it is guaranteed that both techniques make use of the same (version of the) code.
- Using Java bytecode is straightforward and efficient as it is machine independent (i.e., all it needs is a JVM).

A shortcoming of using bytecode rather than source code is the loss of potentially important information, such as comments and conditions of branching statements.

Static reverse-engineering using Rigi

The semi-automatic reverse-engineering approach of Rigi consists of two phases: the identification of software artifacts and their relations, and the extraction of design information and system abstractions. In order to accomplish this Rigi uses several parsers. The visualization and analysis of the extracted information are made possible by the Rigi visualization engine Rigiedit.

The following artifacts are extracted from bytecode: classes, interfaces, methods, constructors, variables, and static initialization blocks. If applicable, information about return types, visibility and other access modifiers related to these artifacts is extracted as well. Furthermore, the following relationships among these artifacts are extracted: containment (i.e., a class contains a method), call (i.e., a method calls another method), access (i.e., a method accesses a variable), and assignment (i.e., a method assigns a value to a variable).

The static structure of the target software is visualized through (directed) dependency graphs of which the nodes represent the extracted artifacts, and the arcs represent the relationships.

Dynamic reverse-engineering using SCED

Using the static information generated by Rigi, i.e., by selecting certain classes and/or methods, runtime information can be generated. In order to accomplish this, breakpoints are set using the JDebugger tool. Likewise, control statements (if, for, while, do-while, case) can be detected by setting appropriate breakpoints. When breakpointed code is hit, runtime information in terms of an event trace is generated. As a result the sequence diagrams generated by SCED are visually updated. The JDebugger tool maintains a stack containing previously activated classes. This stack is used to retrieve the sender of a message (i.e., caller of a method) in case a certain breakpoint is hit. Furthermore, Shimba provides means to trace exceptions. If selected by the user, as soon as an exception is thrown, the result will be made visible in the sequence diagrams.

Control flow statements, e.g., if-conditions, are added to sequence diagrams through so-called *state boxes* and *assertion boxes*. As Shimba relies on Java bytecode instead of source code, the actual conditions cannot be retrieved. Consequently, the only information available in the aforementioned boxes is the line number of the condition in the program's source code as well as, for example, whether or not a certain condition is satisfied. When needed, one has to lookup the Boolean expressions in the source code using the given line numbers. Worth mentioning is that the availability of these line references requires the source code to be compiled using a debugging switch.

Let us have a look at the structure of sequence diagrams generated by SCED (Figure 2.4).

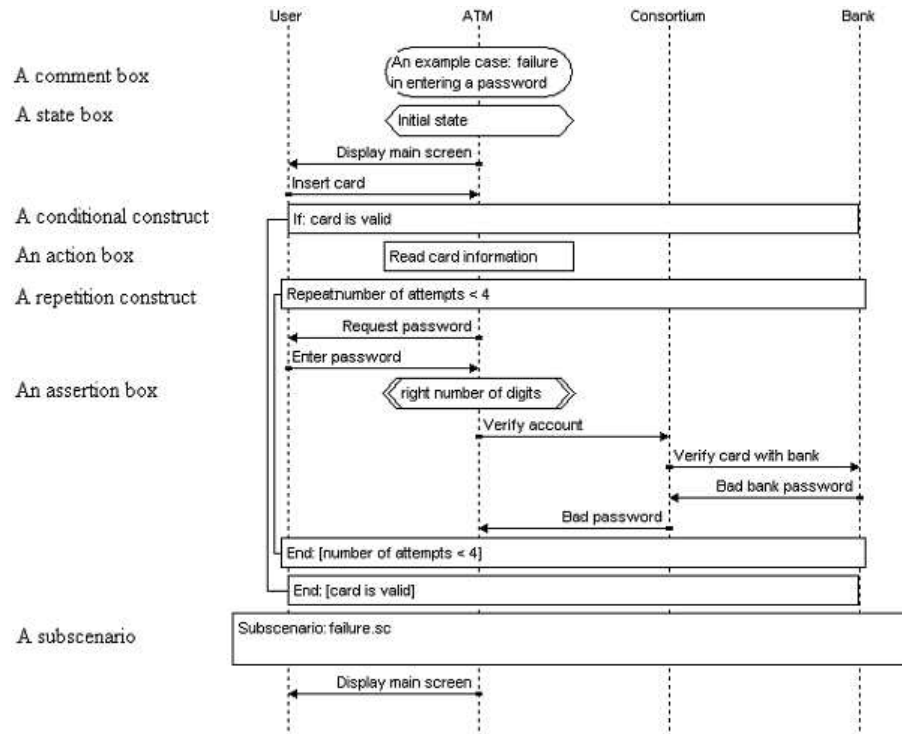


Figure 2.4: SCED sequence diagram notation concepts.

As can be seen a SCED sequence diagram can contain *comment boxes*, *state boxes*, *action boxes*, *assertion boxes*, *conditional constructs*, *repetition constructs*, and *subscenarios*. Action boxes represent messages sent by an object to itself. State boxes and assertion boxes are linked to each other, i.e., the former represent the execution of some Boolean expression whereas the latter indicate which clauses of the Boolean expression are actually evaluated as true. Repetition constructs and subscenarios are used to make the diagrams more readable. They are used to indicate so-called *behavioral patterns*, as stated by Systä et al.. A subscenario contains a reference to another sequence diagram, thus abstracting the actual sequence diagram from message sequences in the referenced diagram. Shimba uses Boyer-Moore string matching algorithms to identify these behavioral patterns [7].

Combining static and dynamic information

Shimba provides means to use static information to generate dynamic information and vice versa. It is possible to select components in the Rigi graphs in order to generate sequence diagrams concerning only the selected items. Likewise, one can select a set of sequence diagrams in order to generate the related static view. The latter is called *model slicing*, i.e., given a set of sequence diagrams the current model is sliced to obtain a model representing a specific situation. Moreover, in Rigi the static view can be raised to a higher level

by selecting nodes and collapsing them into higher level components (e.g., subsystems or packages). Consequently, using this high-level Rigi graph the generated SCED sequence diagram will be at a higher level of abstraction as well.

2.1.3 Jinsight

Jinsight is a tool which can be used to visualize the runtime behavior of Java programs [54]. According to the authors, De Pauw et al., it is helpful for performance analysis, debugging, and any task in which you need to better understand what your Java program is really doing. In order to trace a program's execution Jinsight makes use of a specially instrumented Java Virtual Machine (JVM) or a profiling agent and a standard JVM, depending on the version of the Java platform. A user is presented with several different views in order to simplify the exploration of the execution trace. These views are linked such that it is possible to navigate from one view to another.

The following views are available: the *Histogram view*, the *Reference Pattern view*, the *Execution view*, and the *Call Tree view*. The Execution view is quite similar to a sequence diagram in that it depicts objects and their interactions while the vertical axis represents the time (from top to bottom). Moreover, it is possible to zoom into areas of interest, thus creating the possibility to examine specific parts in more detail. Let us shortly summarize the functionality of the available views:

- The Histogram view. This view can be used by the user to detect performance bottlenecks. It shows object references, instantiation, and garbage collection.
- The Reference Pattern view. This view visualizes the pattern of references of objects by arcs and nodes, respectively. To reduce the amount of visual overhead, instances are grouped by type and reference. It is useful for detecting memory leaks, for instance.
- The Execution view. This view represents the program's execution sequence. Like already mentioned, this visualization is quite similar to sequence diagrams. Using this view the user is able to analyze concurrent behavior, thread interactions, and possible deadlocks.
- The Call Tree view. This view presents the user with a summary about, e.g., the total number of calls per method and their contribution to the total execution time.

Jinsight is a research prototype that was originally designed and implemented at IBM's T.J. Watson Research Center in 1998. Since 2005 a number of its techniques, including the execution pattern notation, are used in a plugin that is part of Eclipse's Test & Performance Tools Platform (TPTP) Project [17].

Instrumentation and tracing

As already mentioned, Jinsight depends on a specially instrumented Virtual Machine, rather than instrumented source- or bytecode. Moreover, depending on the Java platform version,



Figure 2.5: An Execution view in Jinsight.

a standard JVM is used with a profiler. The user is able to specify the amount of time a target program must run in order to produce a trace. Since traces can be very large, Jinsight supports so-called *task-oriented tracing*. This allows users to trace details of a program task selectively while, at the same time, retaining important contextual and sequencing information.

Abstraction

In order to abstract from irrelevant data Jinsight provides *information exploration techniques*. Practically, this means that a user can exploit his knowledge to structure the visualizations by focusing on specific areas of interest. For instance, the user can select certain unusual looking invocations of a method, and navigate to other views to investigate the behavior in more detail, thereby abstracting from irrelevant information. Moreover, users are also able to define their own analysis units, so-called *execution slices*, to group related activities together. Likewise, activities that are outside the scope of study can be excluded. There are different ways to define execution slices, for example, a simple point-and-click

in a certain view, or a full query capability based on static and dynamic attributes of trace data.

Visualization

As stated before, Jinsight produces four views of which the execution view is quite comparable to a large sequence diagram. Figure 2.5 depicts such an execution view.

The vertical axis represents the time (from top to bottom). The colored stripes correspond to the executions of methods on objects and are, therefore, similar to the object lifelines in sequence diagrams. The execution stack increases from the left to the right. The different colors of the stripes represent the different classes. A set of stripes, a so-called *lane*, is displayed for each thread in the program. In Figure 2.5 only one thread is visualized. The behavior of each thread is characterized by the following features:

- The length of each stripe (corresponding to the time spent in each method)
- The number of stripes (representing the stack depth)
- The different colors (corresponding to the classes)

In Figure 2.5 four phases can be distinguished. Figure 2.6 shows the result when zooming into the last phase.

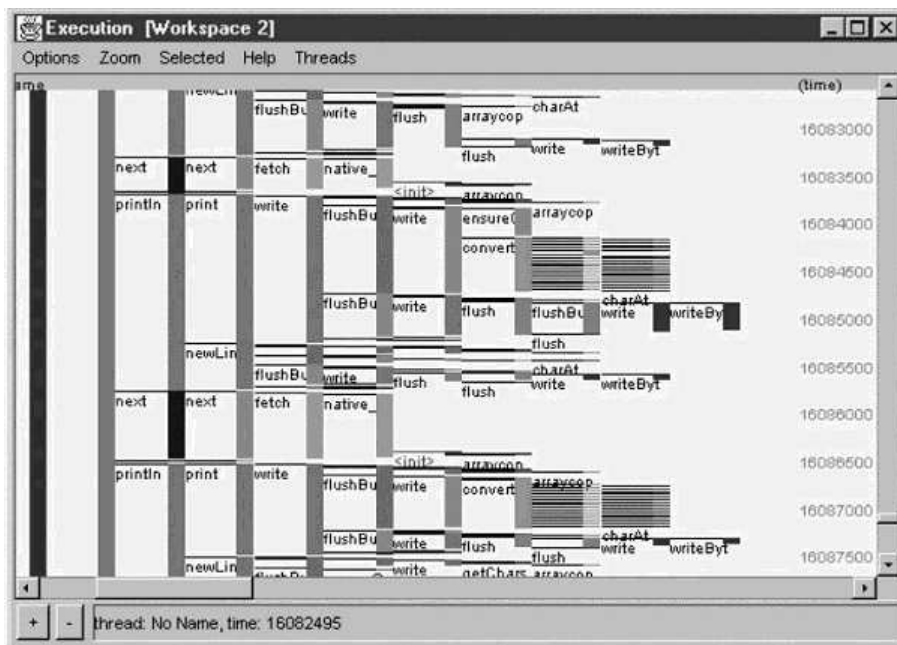


Figure 2.6: Result of zooming into the last phase of the Execution view.

As can be seen, Figure 2.6 does not differ much from a sequence diagram. A difference between them is the variable thickness of the edges (i.e., messages in sequence diagrams) that represent the method executions. The thicker such an edge is, the longer the time spent executing the specific method. Consequently, in Figure 2.6, `println` takes more time to finish than `next`. Another difference is the fact that when going deeper into the call stack, the edges which represent the messages pretty much stay at the same height, whereas in sequence diagrams these messages are placed somewhat to the bottom.

2.1.4 JAVAVIS

JAVAVIS is a reverse-engineering environment focussed towards increasing the understanding of object-oriented programming concepts and hence has educational purposes [49]. In order to trace a program's execution JAVAVIS makes use of the Java Debug Interface (JDI) [25]. The execution is visualized through object diagrams and a sequence diagram. The visualization of the event trace is made possible using the Vivaldi kernel, which is a Java class library for programming 2D animations with smooth transactions. Figure 2.7 illustrates the JAVAVIS architecture.

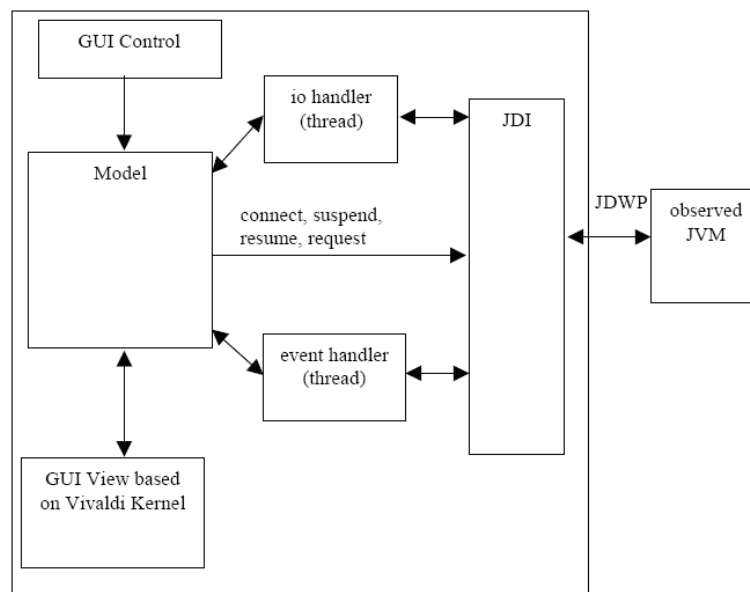


Figure 2.7: JAVAVIS architecture.

The system contains a control component which detects user events by listening to actions regarding Java swing components (e.g., selection of a menu entry or clicking a button). The model component contains all data generated by the JDI. Likewise, the JDI is used by the model component for sending commands to the observed program. When the model data is updated, the view component is updated accordingly. The system design, hence, follows the MVC (Model-View-Control) design pattern. The i/o handler and event handler are put

in separate threads since i/o operations as well as removing events from the event stack are blocking operations. These threads inform the model component as soon as an event has occurred.

The Java Debug Interface

As already stated, JAVAVIS uses the Java Debug Interface to trace a program's execution. It is part of the Java Platform Debugger Architecture (JPDA) [26]. Figure 2.8 depicts the JPDA.

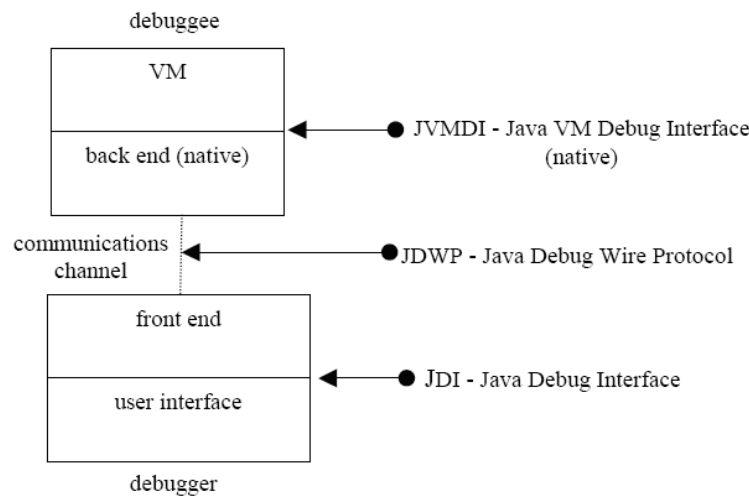


Figure 2.8: Java Platform Debugger Interface (JPDA).

As can be seen the JPDA consists of three components:

- Java Virtual Machine Debug Interface (JVMDI)
- Java Debug Wire Protocol (JDWP)
- Java Debug Interface (JDI)

The JVMDI is a native interface that each Java Virtual Machine has to provide in order to be debugged. The JDWP is the protocol used for communication between a debugger and the Java Virtual Machine which it debugs. The JDWP allows the JVMDI to be used from remote computers. The JDI is an interface which uses JDWP to access a JVMDI (which can be located at a remote computer). The specific transport mechanism used by JDI can be chosen based on the type of connection, e.g., through shared memory or a TCP connection.

There are three ways to establish a connection between the debugger (here: JAVAVIS) and the debuggee (i.e., target program):

- The debugger launches the debuggee using the JDI, thus automatically establishing a connection between them. This option is the one chosen by JAVAVIS.
- The debugger is chosen to be the client which connects to a currently active JVM.
- The debugger is chosen to be the server and waits until some JVM opens a connection to it. This option is only possible if the JVM has been started with a special option in order to open a connection to a “waiting” debugger.

The JDI provides means to read the standard output and write to the standard input of the debuggee. Furthermore, the debugger is able to register its interest in certain events. The debuggee puts an event description into an event queue as soon as a registered event occurs. Consequently, the debugger can remove such an event description from the event queue. The debugger is able to suspend and resume the debuggee. When the debuggee has been suspended, the debugger can read all kinds of state information.

JAVAVIS aims at students who want to become familiar with object-oriented programming concepts. When running the target program, diagrams are generated and updated step by step. The user is presented with different views and changes in the diagrams are smoothly visualized. The sequence diagrams are updated as soon as, for example, new objects are created or methods are executed. The only abstraction method used by JAVAVIS is filtering out system classes in packages like `java.lang` and `java.util`. In Figure 2.9 a sample JAVAVIS sequence diagram is shown.

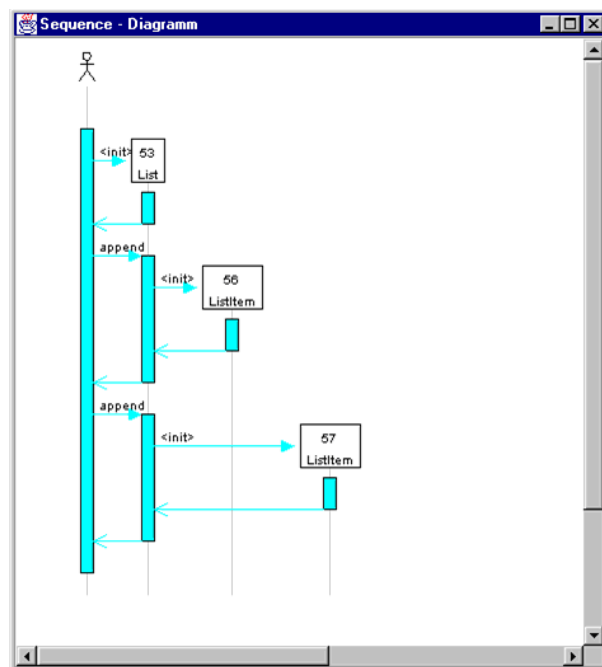


Figure 2.9: Sample JAVAVIS sequence diagram.

2.1.5 Approach by Briand et al.

Briand et al. have designed and implemented a prototype to reconstruct sequence diagrams using source code instrumentation [9]. Their approach consists of instrumenting a program and running it, thus producing traces. These traces are then analyzed and visualized through UML sequence diagrams. In order to accomplish this two metamodels have been designed. On the one hand, we have a metamodel of which the traces are instances: the *trace metamodel*. On the other, we have a metamodel representing the sequence diagrams: the *scenario diagram metamodel*. Since traces are generated by a specific execution of the target program, i.e., a scenario, the eventual diagrams are called *scenario diagrams*. Mapping rules have been defined, using the Object Constraint Language (OCL) [71], in order to map a metamodel of traces to a metamodel of scenario diagrams.

The prototype has been implemented using different programming languages. The instrumentation part has been implemented in Perl whereas the transformation part (traces $\implies$ scenario diagrams) has been implemented in Java. The target language is C++, although, according to Briand et al., the prototype can be easily changed to support other languages, since the executed statements monitored by the instrumentation are universal (e.g., method's entry and exit, control flow structures). Moreover, they merely focus on instrumentation and transformation, thereby not addressing the actual visualization of the scenario diagrams. Their intent is to interface with existing UML CASE tools and import the generated diagrams using a data interchange format such as XMI [51].

Scenario diagram metamodel

In order to know what information to trace, first a scenario diagram metamodel has been created, which is then used to design a trace metamodel. The scenario diagram metamodel is an adaption of the UML metamodel, which is a class diagram that describes the structure of sequence diagrams. This model has been adapted to ease the generation of sequence diagrams from traces. Figure 2.10 depicts the scenario diagram metamodel.

As can be seen, messages (abstract class *Message*) have a *callerObject* and a *calleeObject*, which are the source and target, respectively. Both objects correspond to a *ContextSD*, which can be an *InstanceSD* or a *ClassSD*. The former represents runtime objects whereas the latter represents entire classes (e.g., in case the abstraction is at the class level rather than at the object level). Messages can be categorized in different categories, that is, method calls (class *MethodMessage*), return messages (class *ReturnMessage*), or the iteration of one or more messages (class *IterationMessage*). A message can contain parameters (class *ParameterSD*) and can be sent under certain conditions (class *ConditionClauseSD*). Class *ConditionClauseSD* has two attributes, namely *clauseKind* to indicate the type of condition (e.g., if or while) and *clauseStatement* to indicate the actual condition. Class *MethodMessage* contains an ordered list of *ConditionClauseSD* instances (aggregation in figure 2.10) indicating the logical conjunction of conditions, thereby representing the complete condition under which a message is sent.

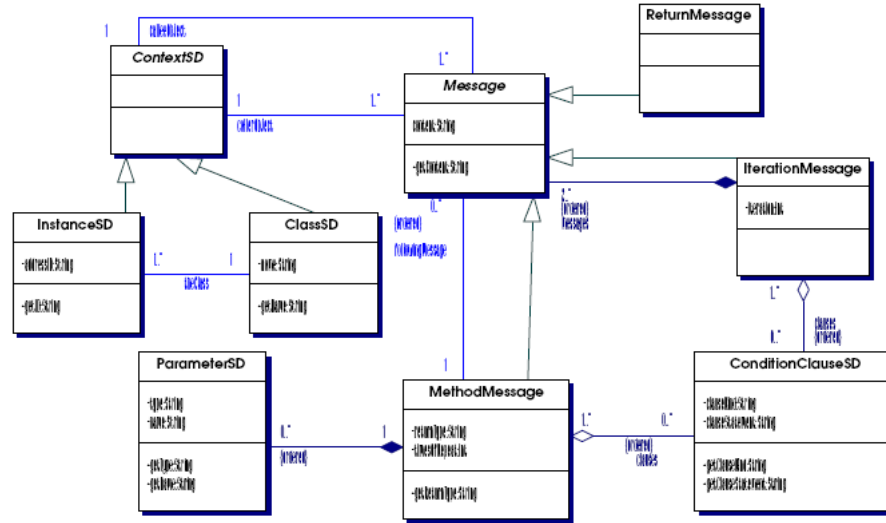


Figure 2.10: Scenario diagram metamodel.

Furthermore, the repetition of a single message is specified by the `timesOfRepeat` attribute in `MethodMessage`, whereas the repetition of two or more messages is specified by the `IterationMessage` class. This is necessary since both situations are modelled differently in UML sequence diagrams. Finally, the association between `Message` and `MethodMessage` is needed as a message can trigger other messages.

Trace metamodel

Using a Perl script the source code is instrumented by adding specific statements in order to retrieve the required information at runtime. Such a statement produces a single text line in the trace file. The following “events” are reported on:

- Method entry and exit. The method signature, the class of the target object (i.e., the object executing the method) and its memory address are retrieved.
- Conditions. For each condition statement the kind of statement (e.g., if) and the exact condition are retrieved.
- Loops. For each loop statement the kind of loop (e.g., while), the corresponding exact condition and the end of the loop are retrieved.

Using this information it is possible to retrieve the source of a method call. After all, since a trace file contains a chronological ordering of events, the source of a method call simply is the previous call in the trace file. Moreover, as conditions can be nested, the complete condition under which a call is performed can be detected by performing a logical conjunction of all conditions that appear before the specific method call. Figure 2.11 illustrates the trace metamodel.

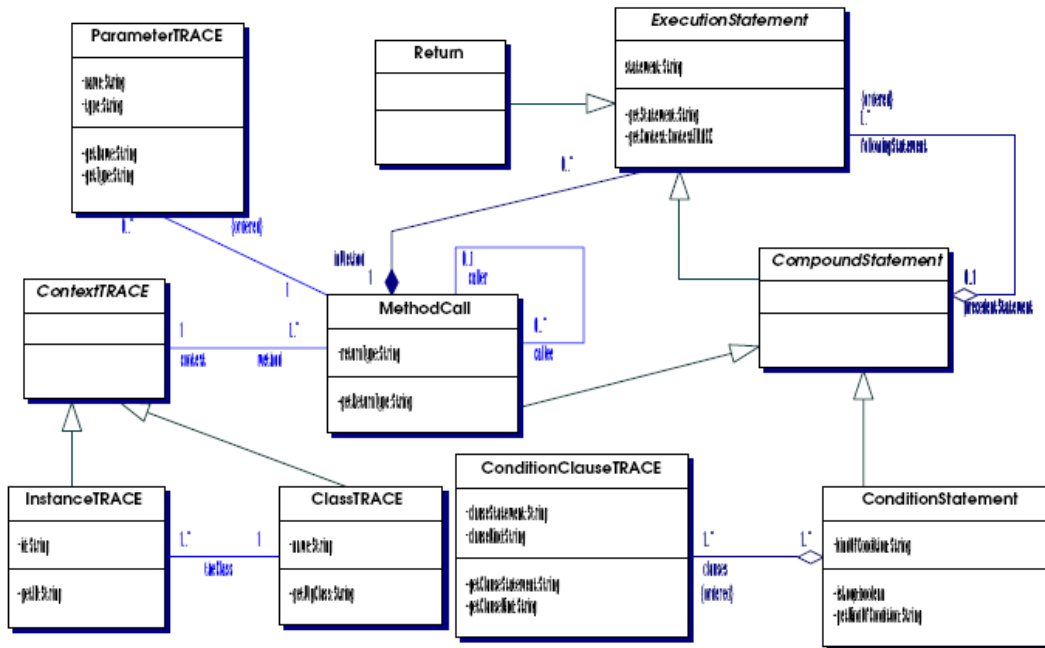


Figure 2.11: Trace metamodel.

The trace metamodel and the scenario diagram metamodel look quite similar. However, there are some key differences. For example, class `MethodCall` only has direct access to its `calleeObject` (target) but not to its `callerObject` (source). The latter has to be retrieved by querying the previous method call. Class `MethodMessage`, on the other hand, has direct access to both its source and target objects (instances of `ContextSD`). Consequently, the mapping between both metamodels is not straightforward. In order to be able to create an instance of the trace metamodel, certain pieces of information have to be computed from the trace file, e.g., the complete condition under which a method call is performed, repetition of method calls (loops) and the identification of a return message for a call.

Consistency rules

Three consistency rules have been derived, expressed in the OCL, which associate an instance of the trace metamodel with an instance of the scenario diagram metamodel. Briand et al. emphasize that the derived OCL rules only express constraints between the two metamodels. Thus, they are not algorithms, but they provide a specification and insights into how to implement such algorithms. These consistency rules identify instances of classes `MethodMessage`, `ReturnMessage`, and `IterationMessage` (scenario diagram metamodel) from instances of classes `MethodCall`, `Return`, and `ConditionStatement` (trace metamodel), respectively. In Figure 2.12 a sample sequence diagram, reconstructed by the prototype of Briand et al., is shown.

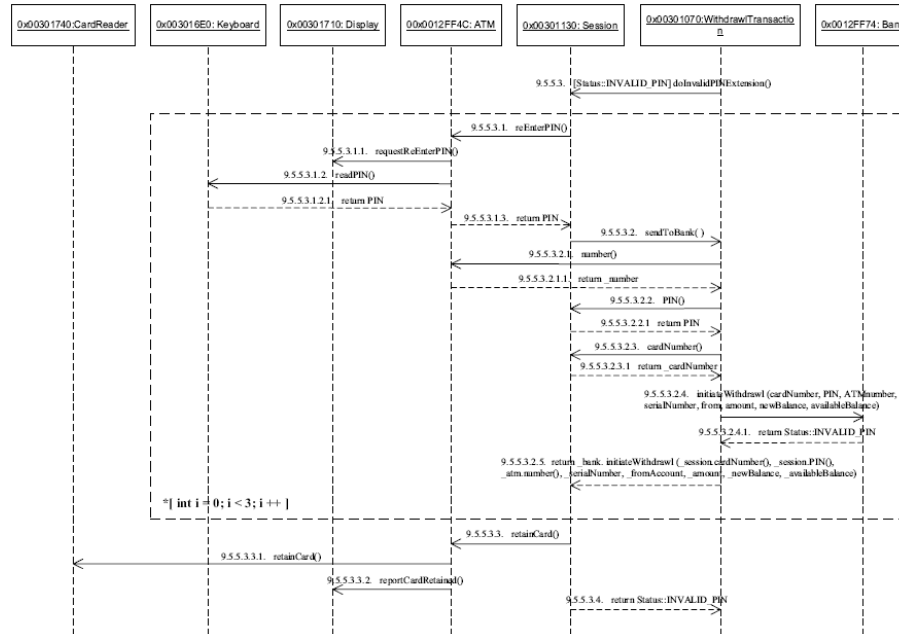


Figure 2.12: Sample sequence diagram.

2.1.6 SDR Framework

The Scenario Diagram Reconstruction (SDR) framework, designed and implemented by Cornelissen et al., makes use of JUnit test cases as a starting point for the reconstruction of sequence diagrams [13, 6]. Like in [9], the source code of the target program is instrumented in order to produce traces. Likewise, the eventual generated diagrams are referred to as *scenario diagrams*, since they depend on a particular execution (i.e., scenario) of the target program. In contrast to Briand et al., Cornelissen et al. provide information on how these input scenarios are derived, namely by using JUnit test cases [35, 4]. Another similarity between SDR and [9] is the design of two metamodels: a *scenario diagram metamodel* and a *trace metamodel*. In addition, SDR provides means to apply abstractions to instances of the scenario diagram metamodel in order to reduce the vast amount of data generated during program execution. Figure 2.13 depicts an overview of the SDR framework.

As can be seen the framework can be divided into several components. It contains an instrumentation component, an abstraction component, and a visualization component. As a consequence, the framework is extensible in that it makes a clear separation between the various stages in the reconstruction process. SDR has been implemented using multiple programming languages including Java and Perl.

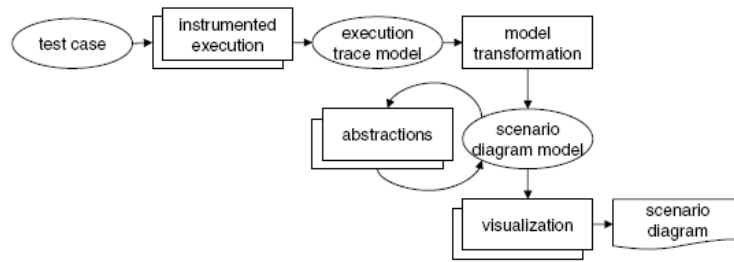


Figure 2.13: Overview of the SDR framework.

Instrumentation and tracing

The SDR framework uses Aspect Oriented Programming (AOP), and more particularly AspectJ, in order to instrument the target program including its JUnit test suite [2]. Instrumentation using aspects has appeared to be very useful, since (1) the system-under-analysis does not require any changes to its implementation in order to trace its execution, and (2) aspects allow us to specify which statements to trace very accurately. For example, AspectJ provides means to report on method calls, method entry / exit, and constructor calls. Moreover, AspectJ allows us to retrieve information on the unique objects involved in certain method calls, the runtime arguments as well as the actual return values, if they exist. Furthermore, using AspectJ it is easy to trace methods by using wildcards and specifying only part of the name of the specific method to be instrumented. As a result, the various stages in a test case (i.e., *fixture*, *test execution*, *result validation*, and *teardown*) can be easily distinguished by exploiting the naming conventions or annotations for these stages within JUnit.

The generated traces are saved to disk in trace files using a common format describing the events that can take place during the system’s execution. A distinction is made between the beginnings and endings of method calls, static method calls, and constructor calls. Figure 2.14 depicts the trace metamodel.

As can be seen, an execution trace consists of multiple events which can be of different types. For example, a “method start” event consists of a method identifier, the caller and callee object, the actual parameters passed to the method (if they exist), and the method signature. A “method end” event, on the other hand, only consists of a method identifier and the actual return value. The other events are similar to the ones mentioned above, so these are omitted in Figure 2.14 in order for the diagram to remain readable. No information about conditions or repetitions is collected.

Abstraction

Since tracing the execution of a program inevitably leads to large trace files, abstraction techniques need to be applied in order for the reconstructed scenario diagrams to become human-readable. However, as in SDR each test case produces a separate diagram, some-

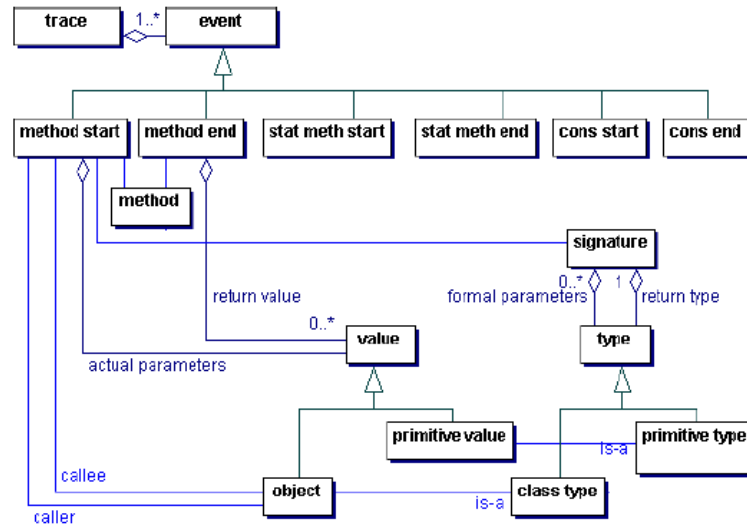


Figure 2.14: Trace metamodel.

times abstractions are not even necessary. This is particular the case when using small unit tests. Cornelissen et al. have defined a catalog of abstractions and filterings in the context of reverse-engineering of sequence diagrams:

- Constructor hiding. Omit all constructors and their control flows. This is particularly useful in the fixture (i.e., initialization phase) of complex JUNIT tests.
- Selective constructor hiding. Omit irrelevant constructors and their control flows, i.e., constructors related to objects which are never used. By doing so, the amount of information is reduced while at the same time essential information is kept.
- Maximum stack depth. Omit all interactions above a certain stack depth threshold, thus filtering out low-level messages.
- Minimum stack depth. Omit all interactions below a certain stack depth threshold, thus filtering out high-level messages. This can be useful to filter out start-up messages.
- Fragment selection by using stack depths. Choose a set of methods of interest and, by selecting appropriate minimum and maximum stack depths, highlight its direct environment.
- Fragment selection by zooming. Zoom in on areas of interest.
- Pattern recognition. Identify recurrent patterns of execution and summarize them (i.e., the involved method calls) by, for example, collapsing techniques.
- Object merging (clustering). Merge lifelines of closely related (or manually selected) objects such that their mutual interactions are hidden.

- Colors. Use color techniques (e.g., graying out or using different colors for different fragments) to distinguish between relevant and irrelevant parts.
- Getters and setters. Omit simple get- and set-methods.
- Textual techniques. Omit return values when they do not exist (i.e., void methods), abbreviate full parameters, and abbreviate return values.

Visualization

Cornelissen et al. do not focus on the visualization of the scenario diagrams. Their main focus is on the instrumentation of JUNIT test cases in order to produce trace files (which are instances of the trace metamodel), converting these instances to instances of the scenario diagram metamodel, and applying appropriate abstraction techniques based on runtime metrics. Figure 2.15 depicts the scenario diagram metamodel.

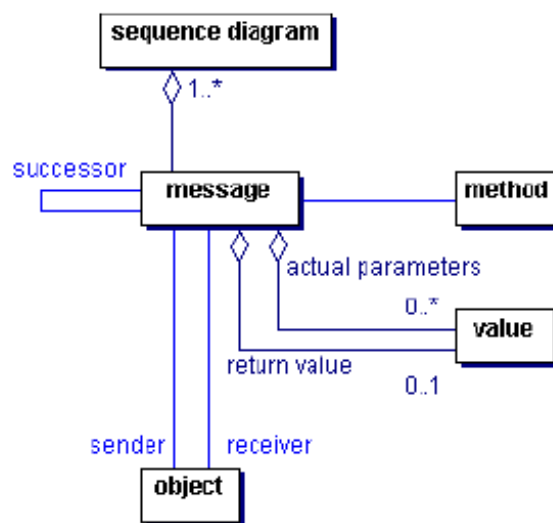


Figure 2.15: Scenario diagram metamodel.

As can be seen a scenario diagram consists of a sequence of messages between objects. Each message contains information about its sender and receiver objects, the corresponding method, the actual parameters, and the return value, if it exists. Furthermore, the metamodel does not contain information about conditions and repetitions. This is because a scenario concerns a specific situation and it is, therefore, not adequate to deal with multiple situations like in a use case, which is a generalized notation. A sample sequence diagram produced using the SDR framework is shown in Figure 2.16.

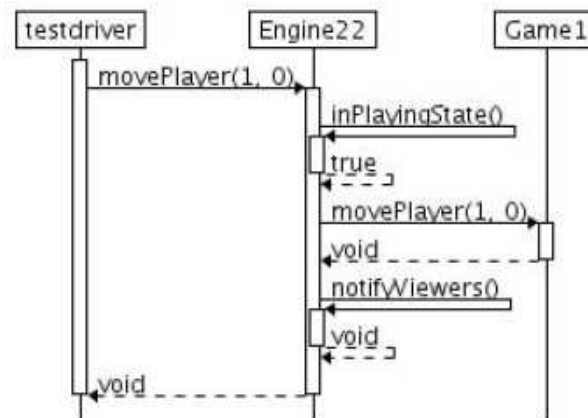


Figure 2.16: Sample sequence diagram.

2.1.7 Scene

In [40] another tool that uses source code instrumentation in order to produce diagrams from a program's execution is presented. It is called Scene (SCENario Environment). It is implemented in Oberon [56] and produces scenario diagrams. The target language is Oberon as well.

2.1.8 Approach by Taniguchi et al.

In [67] a method is proposed to extract compact sequence diagrams from dynamic information of object-oriented programs. The authors, however, do not give information on how the execution of software is traced. They present four compaction rules, including compaction of repetitions and compaction of recursive calls, in order for the traces to be reduced in size, thereby producing compact sequence diagrams. The target language is Java.

2.1.9 Approach by Delamare et al.

In [14] a method is proposed to generate UML 2.0 sequence diagrams from execution traces. The method is based on state vectors that allow detection of loops and alternatives within a single sequence diagram as well as the combination of several sequence diagrams. In order to trace a program's execution a debugging tool called JTracor is used, which relies on the Java Debug Interface (JDI). The target language is Java.

2.1.10 Approach by Merdes et al.

In [46] the authors present a thorough study of technological options when it comes to developing a tool for the reconstruction of UML sequence diagrams from executing Java programs. They implemented such a tool as well. It uses AspectJ to instrument the Java system at load-time (i.e., load-time weaving).

2.2 Approaches that reconstruct alternative diagrams

2.2.1 Approach by Walker et al.

In [69] an approach has been developed for visualizing the operation of an object-oriented system at the architectural level. It is implemented in Smalltalk [64] and instruments the Smalltalk Virtual Machine in order to capture program behavior. Obviously, the target language is Smalltalk as well.

2.2.2 The Collaboration Browser

In [57] a tool, The Collaboration Browser, is presented for recovering object collaborations from execution traces. It is implemented in Smalltalk and uses source code instrumentation. Pattern matching techniques are used to apply abstractions in terms of so-called collaboration patterns. These are similar to the behavioral patterns in Shimba as well as the interaction patterns in ISVis. The target language is Smalltalk.

2.2.3 Approach by De Pauw et al.

In [53] an approach is presented that generates novel views of the behavior of object-oriented systems, and an architecture for creating and animating these views. It uses source code instrumentation and the target language is C++.

2.3 Surveys

A comparative evaluation of dynamic visualization tools is presented in [52]. The tools are evaluated on a number of general software comprehension and specific reengineering tasks. The authors, however, do not focus on how the execution of a program is traced, or which abstraction techniques are used. Moreover, they are not particularly interested in sequence diagrams.

In [22] a survey of eight trace exploration tools and techniques is presented. The goal of the authors is to uncover the underlying concepts behind these tools in an attempt to build a common core of techniques that can be useful for understanding the behavior of object-oriented systems. As such, they present the advantages and limitations of these tools and discuss how they can be improved. Although the authors do not focus on methods to trace a program's execution, they do present several abstraction techniques.

2.4 Conclusion

In this chapter we listed several (surveys involving) existing approaches that reconstruct diagrams using dynamic analysis. As for sequence diagrams in particular, we outlined ten approaches of which six have been elaborated in more detail. During this survey we encountered various abstraction techniques including, for example:

- Filtering out system classes (JAVAVIS).
- Replacing recurrent patterns by repetition constructs (Shimba, ISVis, SDR).
- Replacing separated repeated patterns by subscenario boxes (Shimba, ISVis).
- Omitting asserts (SDR).
- Omitting getters (SDR).
- Constructor hiding (SDR).
- Stack depth limitation (SDR).

We implemented a number of the encountered abstraction techniques in JRET as explained in chapter 4. When it comes to the collection of runtime data, the described approaches make use of techniques like source code instrumentation and a customized debugger. In order to acquire an overview of the different ways to instrument a program, the following chapter outlines the available possibilities for collecting runtime data from Java software.

Chapter 3

Collecting Runtime Data from Java Software

In this chapter we describe different techniques to retrieve data from a running Java application. These techniques include source code instrumentation (section 3.1), bytecode instrumentation (section 3.2), and instrumentation of the application's runtime environment (i.e., its Virtual Machine, section 3.3). The chapter is concluded in section 3.4.

Figure 3.1 illustrates the symbolic process from source code to sequence diagram.

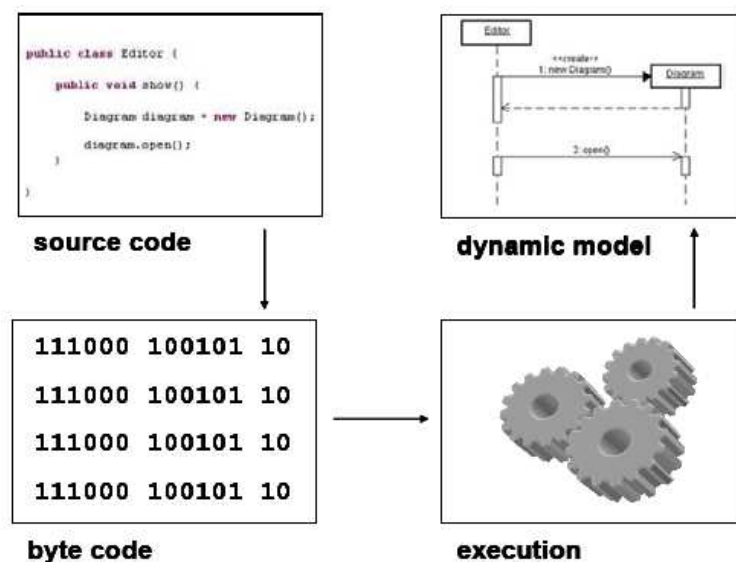


Figure 3.1: Symbolic steps from source code to sequence diagram model.

3.1 Source code instrumentation

Source code instrumentation means inserting code fragments, often referred to as *probes*, into a program's source code. For example, one could insert print statements at the beginning and end of each method, in order to check which runtime objects exist and how they interact. An obvious limitation of source code instrumentation is the fact that one must have access to the source code. Source code instrumentation basically can be divided into two categories:

- Manually instrumenting the source code.
- Instrumenting the source code using aspects.

Obviously, source code can be instrumented manually. However, this method is both troublesome and error-prone. Usually some script is used to insert the probes. An example of a tool providing this functionality is InsectJ [24, 62].

Another approach is using Aspect Oriented Programming (AOP) and, in particular, AspectJ [2, 36, 16, 20] to instrument the source code. Aspect oriented programming provides a way to add functionality to several classes at once, while the functionality is managed in a central place, the *aspect*. These aspects are then “woven” through the source code using a so-called weaver. Thereafter, the result can be compiled as usual. Moreover, AspectJ provides means for weaving aspects through bytecode, which we will discuss later on. Figure 3.2 illustrates the weaving process.

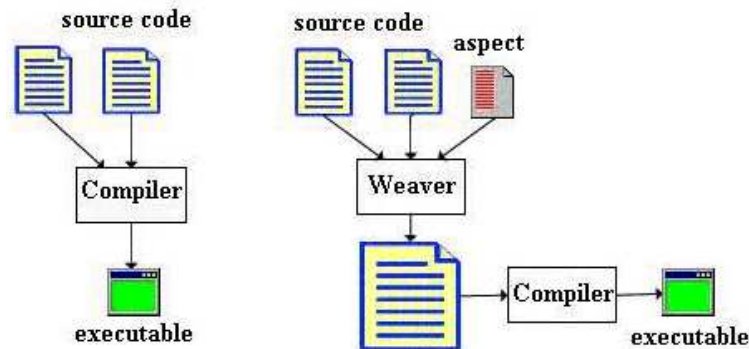


Figure 3.2: Schematical overview of normal compilation (left) versus aspect weaving (right).

Three important terms in AOP are *join points*, *pointcuts*, and *advices*. Join points represent places in a program where additional behavior can be added, for example, before and after execution of methods as well as before and after calling constructors. Pointcuts consist of one or more join points. They are language constructs that pick out a set of join points based on defined criteria. The criteria can be explicit function names or function names

specified by wildcards. The actual code to be executed at the join points is called the advice in AspectJ. It is possible to specify execution of code before, after, and around join points. Consequently, AspectJ provides a flexible way to instrument a Java application. In Listing 3.1 an example tracer written in AspectJ is shown.

```
import sdr.tracers.SimpleTracer;

public abstract aspect CustomTracer extends SimpleTracer {

    protected pointcut theConstructors() :
        call(org.package...new(..));

    protected pointcut theCallers() :
        call(* org.package...*(..));

    protected pointcut startingPoint() :
        execution(* org.package...myStartMethod(..));
}
```

Listing 3.1: An example tracer in AspectJ.

3.2 Bytecode instrumentation

Bytecode instrumentation means inserting code fragments into a Java program's bytecode, i.e., class files. As opposed to source code instrumentation, the source code does not have to be available and is therefore not manipulated. Consequently, one does not have to worry about the existence of various different versions of the source code. Again, AspectJ can be used to weave aspects through the bytecode. In this regard, figure 3.2 should be somewhat adapted in that the source code is first compiled, after which the weaver weaves the aspects through the generated bytecode.

3.3 Instrumentation of the runtime environment

Instrumentation of the runtime environment means instrumenting the environment in which an application runs. As far as Java is concerned, this means instrumentation of the Java Virtual Machine (JVM). This can be accomplished by using virtual machine agents, for instance. The advantage of this method is the fact that neither the source code nor the bytecode is altered. However, the execution of an application can be slowed down considerably. Another option would be implementing a new or modifying an existing JVM. Although this option has the same advantage as the aforementioned instrumentation, i.e., not having to change the application's source code and bytecode, it has a serious limitation, namely the huge effort to implement a new JVM or to even modify an existing JVM.

For the instrumentation of a JVM it is again possible to use AspectJ. However, this time aspects are woven into the byte code at load time, i.e., at the time they are loaded by the class loader of the Java Virtual Machine. Hence, this technique is called *load-time weaving* (LTW). Another option in this regard is the Java Debug Interface (JDI) which is part of the Java Platform Debugger Architecture (JPDA) [25, 26]. The JDI provides means to debug a

JVM from a remote machine. Both LTW by AspectJ and the JDI are techniques based on virtual machine agents which can be applied to existing JVM implementations.

Yet another agent-based solution is the use of the Java Virtual Machine Profiler Interface (JVMPI) which is a two-way function call interface between the JVM and an in-process profiler agent [28]. Like the JDI, it provides means to operate on a JVM that is located on a remote machine. The Eclipse Test & Performance Tools Platform (TPTP), which is a software platform and a set of components to develop testing, profiling and monitoring tools for applications, makes use of the JVMPI [17]. From J2SE 5.0, however, the JVMPI was replaced by the new JVMTI (Java Virtual Machine Tool Interface) [29].

3.4 Conclusion

In this chapter we described three instrumentation techniques and listed their advantages and disadvantages:

1. Source code instrumentation
2. Bytecode instrumentation
3. Instrumentation of the runtime environment

When comparing these methods, we think bytecode instrumentation is the best option to collect runtime data from a Java program. As opposed to source code instrumentation, the source code of the target program does not have to be available. It also prevents multiple versions of the source code to co-exist. Moreover, in contrast to instrumentation of the JVM, it does not slow down the execution of a program considerably. Neither does it require the implementation of a new or the modification of an existing JVM, which would take a huge effort.

Taking this all into account, we implemented bytecode instrumentation in JRET, which is elaborated in the following chapter.

Chapter 4

JRET: The Java Reverse-Engineering Tool

In this chapter we elaborate on our tool that is capable of reconstructing UML sequence diagrams through dynamic analysis. JRET is documented and made publicly available as an open source project through SourceForge [34]. The tool is completely written in Java/AspectJ and offers both command-line possibilities and a simple GUI which is depicted in Figure 4.1. It is able to generate sequence diagrams in PNG format, as well as output that can be visualized as sequence diagrams using two different visualizers:

- Alex Moffat’s `SEQUENCE` [63].
- `SDEDIT`: Quick Sequence Diagram Editor [61].

Each visualizer has its own advantages and disadvantages which are explained in section 4.4. As already mentioned in chapter 1, the process of dynamic reconstruction can be divided into the following stages:

1. Creating traces
2. Applying abstractions
3. Visualizing the data

We dissect our tool in terms of these three stages in the following sections. First, in section 4.1, we present an overview of its structure. Section 4.2 explains the first phase in which traces are generated. In section 4.3, we describe the abstraction techniques implemented, followed by section 4.4 in which we present information on the visualization phase. Finally, in section 4.5, we describe the features of JRET’s research package.

4.1 Overview of the structure

Figure 4.2 provides an overview of the structure of our tool. As can be seen, and as previously described in chapter 1, JRET makes use of test cases as a starting point for the

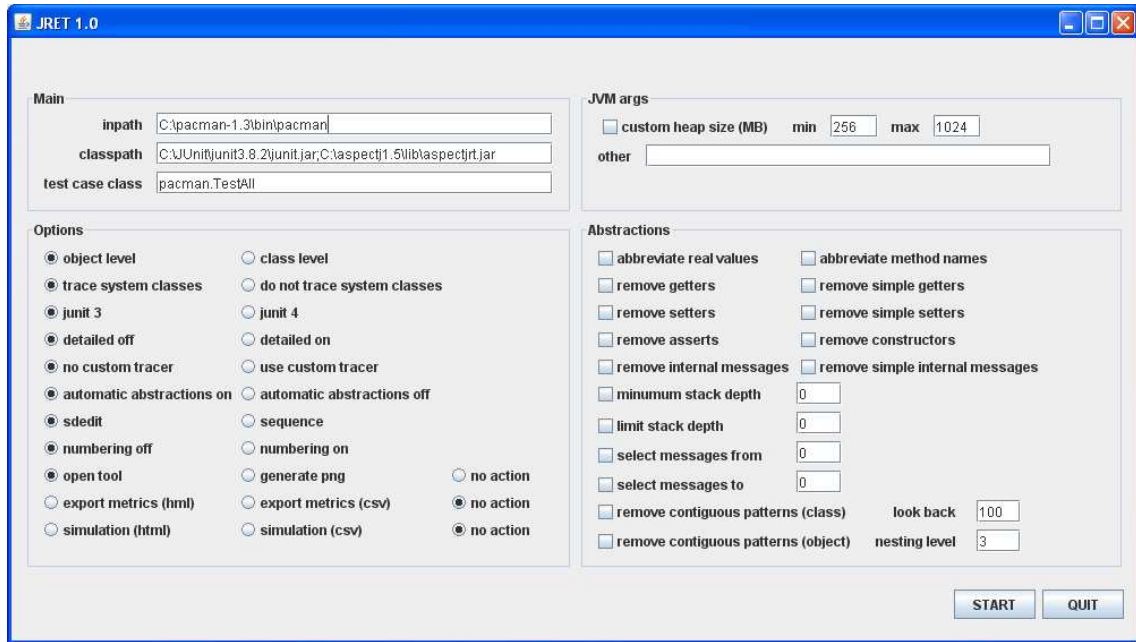


Figure 4.1: The JRET GUI.

reverse-engineering process. These test cases are instrumented and executed. As a consequence, execution traces are generated, both internally (*internal traces*) and externally (*external traces*). They are instances of the *trace metamodel* which we will describe in the following section.

External traces are plain text files containing the entire execution of the target system, and hence can be used as a logging repository. The internal traces, on the other hand, are converted to *internal diagrams* for the next stage in the pipeline: the application of abstractions. After abstractions have been applied, the internal diagrams are exported to *external diagrams* which, like external traces, are plain text files. Likewise, both internal diagrams as well as external diagrams are instances of the *sequence diagram metamodel*, which is also described in the following section. Finally, the external diagrams are visualized using our two aforementioned visualizers.

SDR, which is a similar approach as already described in chapter 2, is an extensible framework in that it makes a clear distinction between the tracing part, the abstraction part, and the rendering of sequence diagrams [13]. A limitation of using separate components, however, is the need for exporting and importing data after each stage. Our tool cuts down on these I/O operations for better performance. The only stage in which there is a need to import data by a separate component is when input is feed to the visualizers. Moreover, although the separation in SDR is an advantage regarding extensibility, it has been implemented using multiple programming languages (i.e., Java/AspectJ and Perl). Thus, as far as

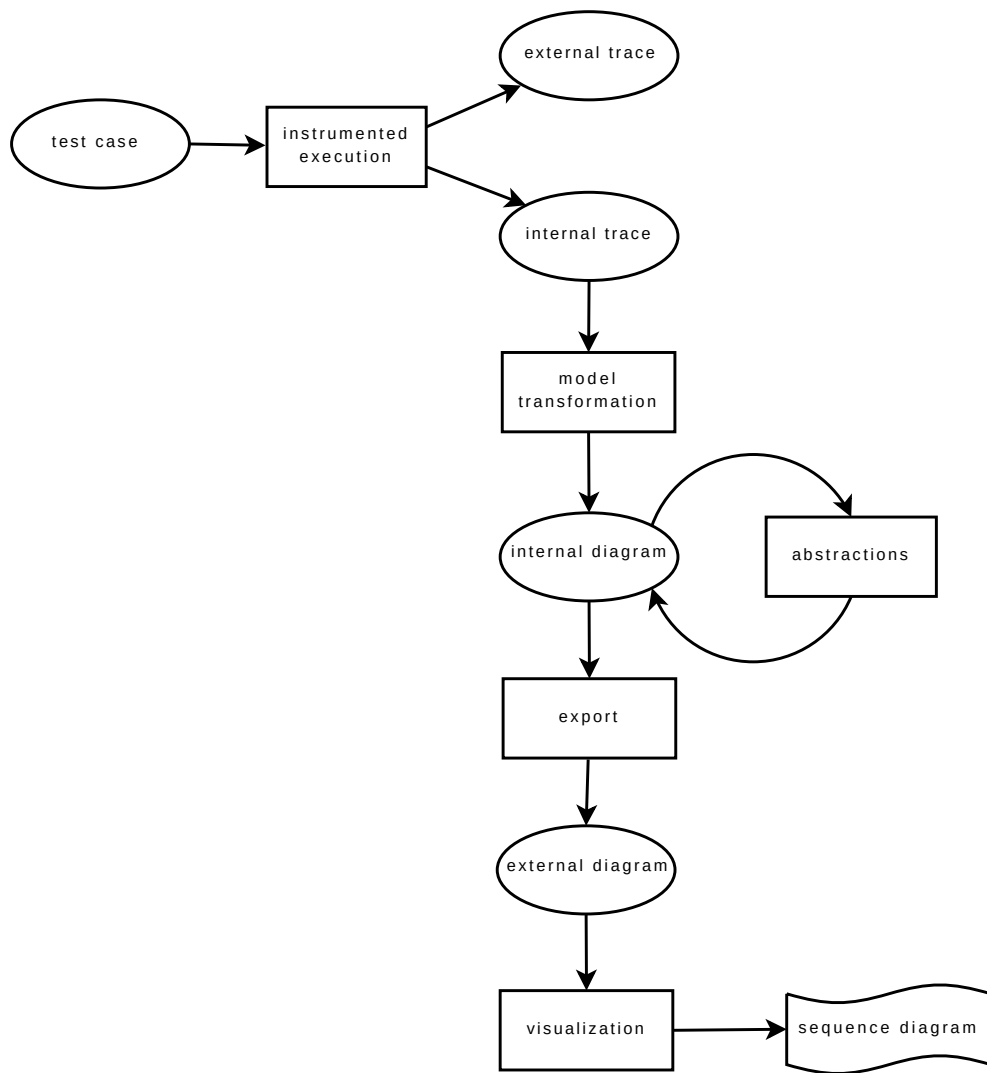


Figure 4.2: The JRET structure.

maintainability is concerned, our approach has an advantage in this respect, since it makes use of Java/AspectJ only.

4.2 Creating traces

In this section we explain how our tool handles the first phase of the dynamic reconstruction process: the creation of traces.

4.2.1 Instrumentation

JRET uses bytecode instrumentation using AspectJ in order to generate traces. We prefer bytecode instrumentation since a program's source code does not have to be available, as is the case when using source code instrumentation. Moreover, this instrumentation technique does not slow down a program's execution considerably, as is the case when using a customized debugger, for instance. It also removes the huge effort required to implement a new or even modify an existing JVM. Our choice for AspectJ is based on the fact that it provides a flexible way to instrument an application while the functionality is centrally managed in the *aspect*. The system-under-analysis therefore does not require any changes to its implementation in order to trace its execution, thereby increasing obliviousness. Using *pointcuts* and *aspects* to trace an application is an effective and uncomplicated option.

Furthermore, aspects allow us to specify which statements to trace very accurately. For example, AspectJ provides means to report on method calls, method entry / exit, and constructor calls. AspectJ allows us to retrieve information on the unique objects involved in certain method calls, the runtime arguments as well as the actual return values, if they exist. Additionally, using AspectJ it is easy to trace methods by using wildcards and specifying only part of the name of the specific method to be instrumented.

Our tracers distinguish between the beginnings and endings of *method* calls, *static method* calls, and *constructor* calls, since they are differently visualized in sequence diagrams.

4.2.2 Granularity

The instrumentation can be either at class level or object level. An advantage of the former method is the fact that it is relatively easy to implement, since it does not require an administration of all runtime objects. Also, the resulting sequence diagrams potentially need less abstractions as it is not necessary to display all different objects. After all, a class X represents all instances of X, and therefore interactions between different objects are not shown.

Instrumentation at object level, on the other hand, provides detailed information on object interactions, and exposes occurrences of polymorphism and late binding. A limitation, however, is that it requires an administration of all runtime objects, thus making it more difficult to implement when compared to instrumentation at class level.

As each method has its own advantages, we prefer the best of both worlds. We think it is important to have detailed information on object interactions, since it is then possible to detect occurrences of polymorphism and late binding. At the same time we like granularity at the class level as some form of abstraction method to make the resulting diagrams more readable. Hence, in JRET we have implemented both methods, and present the user with the opportunity to choose between the different levels.

4.2.3 JUnit test cases

As already mentioned in chapter 1, we use JUNIT tests to represent the different input scenarios. Consequently, the target system as well as its JUNIT test suite need to be instrumented. JRET identifies the various stages in a JUNIT test case (i.e., the fixture, the test execution, and the tear down) and traces them separately, such that they can be distinguished in the resulting sequence diagrams.

Our tool contains tracers which can handle both JUNIT 3 and JUNIT 4 test suites. The main difference between both versions is that the former relies on naming conventions for the methods that represent the different stages, whereas the latter makes use of Java annotations. These naming conventions are as follows:

- fixture $\implies$ void setUp()
- test $\implies$ void test*(), where the asterisk can be any string.
- tear down $\implies$ void tearDown()

As far as JUNIT 4 is concerned, the methods for each stage can be chosen randomly, as long as they are preceded by the correct annotation. These are as follows:

- fixture $\implies$ @org.junit.Before
- test $\implies$ @org.junit.Test
- tear down $\implies$ @org.junit.After

Hence, we created tracers that both exploit the naming conventions in JUNIT 3, as well as tracers which trace the aforementioned annotations.

4.2.4 Implementation

As already mentioned in section 4.1, execution traces are part of the trace metamodel. It is similar to the model described in [13] by Cornelissen et al., and is depicted in Figure 4.3.

As can be seen, a trace consists of one or more events:

- Beginning / ending of a method.

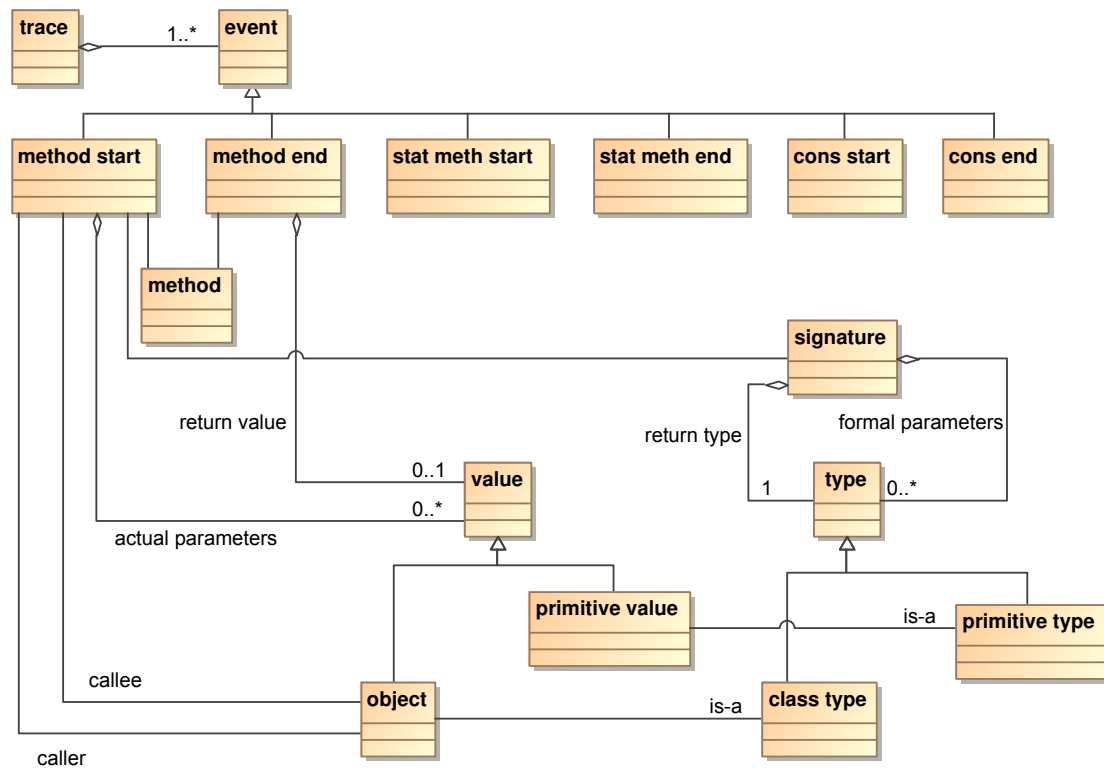


Figure 4.3: The trace metamodel.

- Beginning / ending of a static method.
- Beginning / ending of a constructor.

Associated with the beginning of each method is a signature, a caller, a callee, zero or more (runtime) parameters, and a method identifier. As far as the ending of a method is concerned, it features a method identifier, and possibly a return value. Due to space constraints only the regular method events have been shown in more detail. However, the described associations also go for the remaining ones.

The external traces in Figure 4.2 are plain text files containing the aforementioned events. The internal traces are implemented as sequences of `Message` and `ReturnMessage` objects. The former represents all beginnings of (static) methods and constructors, i.e., the so-called before-events, whereas the latter represents their endings, i.e., the so-called after-events. Internal traces are converted to internal diagrams which are instances of the sequence diagram metamodel. It is depicted in Figure 4.4.

As can be seen, an internal diagram consists of a sequence of messages. They are derived by merging the `Message` and `ReturnMessage` objects that represent the internal traces, thereby

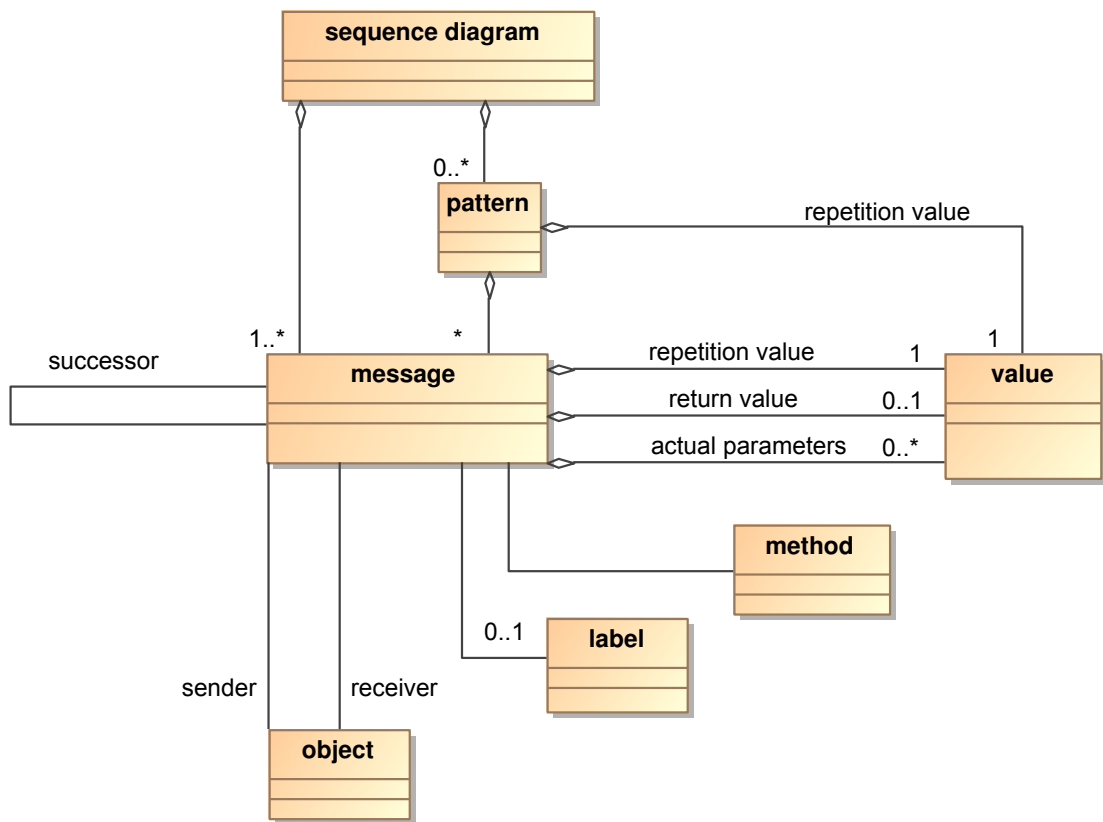


Figure 4.4: The sequence diagram metamodel.

producing Message objects that also contain return values. Moreover, each message is possibly associated with a label indicating the test case stage, and a repetition value that points out how many times it is repeated. Also note that a message can be part of a pattern which can be repeated a number of times. Both repetitions are related to our pattern recognition algorithm which is explained in the next section.

Since JRET-reconstructed sequence diagrams are derived from scenarios rather than use cases, conditions (e.g., *if* and *case*) have been omitted in our metamodel. Nevertheless, it does support repetitions as they are useful regarding abstractions. As such, the model in Figure 4.4 is a simplified version of the UML sequence diagram metamodel [50].

Tracers

JRET comes with several tracers that provide a combination of the following functionalities:

- Trace at object or class level.

- Support for JUNIT 3 or JUNIT 4.
- Include or exclude system classes (in the packages `java.lang` and `java.util`).
- Support for SEQUENCE or SEDIT.

At the top of the tracer hierarchy is the `AbstractTracer` aspect, which contains the abstract `startingPoint()` pointcut that has to be redefined in one of its sub-aspects. It contains listeners that listen to the different events, such as the beginnings and endings of method calls, static method calls, and constructor calls. Also, it initializes the properties which are specified by the parameters provided to JRET by the user. The `AbstractTracer` aspect is extended by either an `ObjectTracer` or a `ClassTracer` aspect, for tracing at object or class level, respectively. They contain abstract pointcuts for tracing constructor calls and (static) method calls, and are respectively extended by a `ObjectTestCaseTracer` and a `ClassTestCaseTracer` aspect. These aspects contain abstract pointcuts for tracing the fixtures and the tear downs of JUNIT test cases. For the actual test the `startingPoint()` pointcut is used.

JRET has 16 stock tracers that implement the aforementioned abstract pointcuts. The `ObjectTestCaseTracer` aspect is extended by 8 tracers, such as the `ObjectTestCase3SequenceTracer` aspect which traces JUNIT 3 tests at object level and generates SEQUENCE output. Likewise, the `ClassTestCaseTracer` aspect is extended by 8 tracers, such as the `ClassTestCase4NoSystemSEditTracer` which traces JUNIT 4 tests at class level, excludes system classes, and generates SEDIT output.

To exclude specific classes or packages from the trace, a custom tracer can be created which should be placed in the (`jret\tracing\tracers\custom`) directory. The easiest way is to copy one of the stock tracers that contain custom pointcuts for the exclusion of system classes, and simply redefine these pointcuts.

Listeners

Attached to each tracer are one or more listeners. The following listeners can be distinguished:

- `TracePrinter`: prints all events to the standard output.
- `TraceFileWriter`: writes all events to disk, i.e., creates trace files.
- `MetricsCollector`: creates metrics files.
- `SequenceListener`: creates input files for the SEQUENCE visualizer.
- `SEditListener`: creates input files for the SEDIT visualizer.

The stock tracers are all connected to a `TraceFileWriter` and a `MetricsCollector`. They are also linked to a `SequenceListener` or `SEditListener` depending on the type of tracer.

Runtime data

JRET generates IDs to make a distinction between the different runtime objects. It uses an object's memory address to create an ID which is appended to the object's class name. As such, object IDs like `Test3`, `HelloWorld11`, and `String128` could be generated. Our choice for the use of an object's internal memory address is based on the following:

- Every object has a memory address that uniquely identifies it.
- An object's memory address can be easily retrieved in Java.

In order to retrieve an object's internal address it is not possible to use the `toString()` method of Java's `Object` class. This is because this method could be redefined in any other class (since `Object` is the root of Java's class hierarchy). The same goes for `Object`'s `hashCode()` method. To overcome this problem, we used the static `identityHashCode(Object o)` function of Java's `System` class.

4.3 Applying abstractions

In this section we explain how our tool handles the second phase of the dynamic reconstruction process: the application of abstractions.

4.3.1 Abstraction techniques

JRET offers the following abstraction techniques:

- The possibility to choose between object and class level.
- Filter out system classes in the packages `java.util` and `java.lang`.
- Abbreviate method names, return values, and arguments.
- Remove getters (with or without their control flow).
- Remove setters (with or without their control flow).
- Remove constructors and their control flow.
- Remove *Asserts*.
- Remove internal messages (with or without their control flow).
- Stack depth limitation / set minimum stack depth.
- Message selection.
- Pattern recognition.

Now, let us explain these techniques.

Choose between object or class level

This technique has already been explained in section 4.2.2. The user is able to choose whether JRET should trace at either object or class level. If class level is chosen, then objects that belong to the same class are merged, thereby providing horizontal abstractions. This can be useful in case one is not interested in runtime objects that exist during the execution of a test case.

Filter out system classes

The user is able to choose whether Java system classes in the packages `java.lang` and `java.util` should be traced. Consequently, calls of `Vectors` or `Strings`, for example, will not be visible in the resulting diagrams. This technique is useful in case one merely wants to focus on the interactions within the target system itself, rather than on method/constructor calls of Java system classes.

Abbreviations

Method names can be abbreviated, as well as return values and method arguments. Using JRET it is possible to identify the returned objects and method arguments using object IDs. Moreover, their actual contents can be exposed as well using the “detailed” option. If these values are large (e.g., long `Strings` or long integers), then they can be abbreviated. This can be useful as it prevents the sequence diagrams from becoming excessively expanded which hinders comprehensibility.

Remove getters

Getters, i.e., methods whose names start with “get”, can be removed from the diagrams. A distinction is made between getters that have control flow (*getters*), and getters without control flow (*simple getters*). Removing simple getters is particularly useful in the situation where most calls are in the control flow of a getter at the beginning of a sequence diagram. Removing this getter including its control flow could result in too much information being erased, leaving the diagram useless. Hence, in such a case it would be better to remove simple getters instead.

Removing (simple) getters can be useful as these methods often can be omitted without the loss of essential information. Consequently, the comprehensibility of the generated sequence diagrams will potentially increase.

Remove setters

Setters, i.e., methods whose names start with “set”, can be removed from the diagrams. A distinction is made between setters that have control flow (*setters*), and setters without control flow (*simple setters*). Removing simple setters is particularly useful in the situation where most calls are in the control flow of a setter at the beginning of a sequence diagram. Removing this setter including its control flow could result in too much information being

erased, leaving the diagram useless. Hence, in such a case it would be better to remove simple setters instead.

Removing (simple) setters can be useful as these methods often can be omitted without the loss of essential information. Consequently, the comprehensibility of the generated sequence diagrams will potentially increase.

Remove constructors

It is possible to remove constructors including their control flow. This abstraction technique is particularly useful for the fixture in a JUnit test case, since the fixture involves the creation of objects that will be used in the actual test. Consequently, large fixtures can be removed from the generated sequence diagrams. If one is not interested in the creation of objects, or prefers method calls over constructor calls for program comprehension, then hiding constructors could be an effective abstraction to potentially increase diagram comprehensibility.

Remove Asserts

Asserts are used to verify the results of a test case. These methods (e.g., `assertTrue(boolean)` and `assertEquals(...)`) are test case specific and thus can be removed without the loss of essential information. Hence, this technique can be useful as it omits messages likely not needed for program comprehension.

Remove internal messages

If an object calls a method on itself, it is named an internal message. Our tool offers the possibility to remove such internal messages. Like with getters and setters, we make a distinction between *internal messages* (i.e., with control flow) and *simple internal messages* (i.e., without control flow). Again, the latter can be used to prevent the diagrams from becoming useless due to the removal of too much information.

This technique can be useful if (the test cases of) the target program initiate(s) a considerable number of (simple) internal messages. Omitting these messages could potentially increase diagram comprehensibility.

Stack depth limitation

It is possible to set a threshold for the stack depth, such that calls below or beyond a certain depth are removed from the resulting diagrams. Limitation of the stack depth is useful for filtering out interactions that tend to be too detailed. Setting a minimum depth, on the other hand, can be used to omit initialization messages that start up a certain scenario. As a consequence, in both situations one is able to focus on the essence of a particular test case. Hence, this technique could potentially increase comprehensibility of the generated sequence diagrams while retaining essential information.

Message selection

A range of messages can be selected in order to study the selection in more detail, thereby excluding the remaining messages. As such, it is possible to thoroughly examine a potentially important method call, for example. This abstraction technique is particularly useful in combination with JRET’s “message numbering” option. When this option is enabled, each message will be accompanied with its message number (from the original trace), which is between square brackets. These values can be used to calculate the number of calls omitted (due to abstractions) between two subsequent messages. To do so, their corresponding numbers simply have to be subtracted.

Pattern recognition

JRET is able to detect recurring contiguous patterns in the resulting diagrams, such as repetitions due to loops and recursion. We used the so-called *preprocessing algorithm* as mentioned in [21] and described [23] to iteratively identify these patterns. They are then visualized only once using a repetition construct. The first phase of the two-phase algorithm involves the identification of single-message patterns due to loops and recursion. The second phase, on the other hand, involves the detection of contiguous redundancies of a sequence of calls. In [23] an example of detection and replacement of nested sequences is given, which is shown in Figure 4.5.

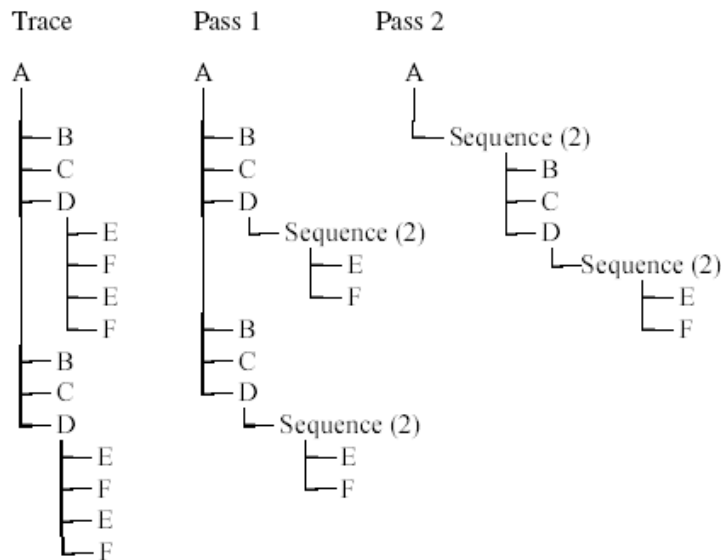


Figure 4.5: An example of detection and replacement of nested sequences in [23].

As can be seen in Figure 4.5, this particular execution comprises two steps. In the first step the EF pattern is detected, whereas the second step identifies the nested pattern. In JRET the corresponding sequence diagram would look like Figure 4.6.

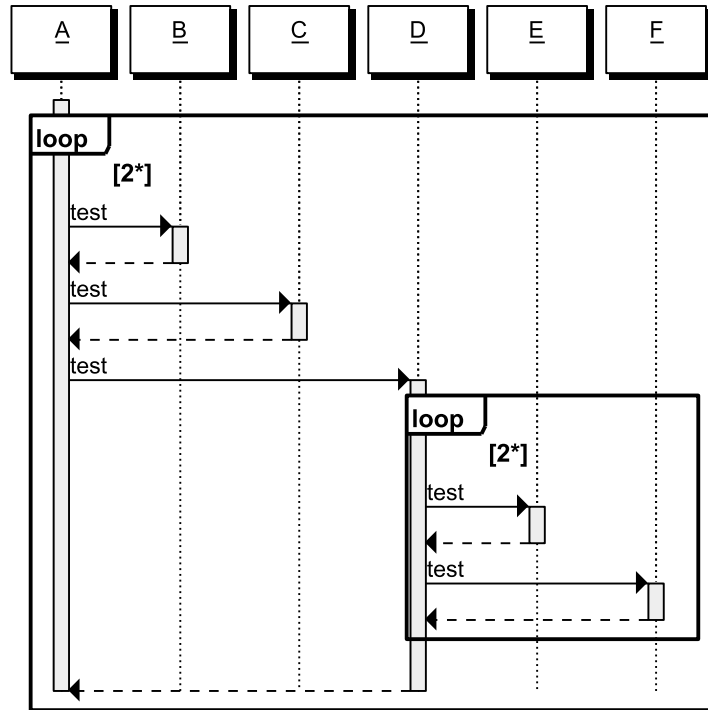


Figure 4.6: The result of previous example visualized using JRET.

The main variables in the detection algorithm are the parameter which indicates how many messages should be searched back, and the parameter that represents the nesting level. The “look back” value states the maximum length of the recurrent pattern in terms of the number of messages, whereas the nesting level indicates the number of iterations (i.e., passes in Figure 4.5) of the iterative algorithm. The latter thus represents the nesting level of loops, for example. In JRET both variables can be configured.

Moreover, JRET provides two variants of the algorithm. The distinction between both variants has to do with when two messages are considered equal. This is the case when:

1. They have the same source object (i.e caller).
2. They have the same method identifier.
3. The target (i.e., callee) is either the same object, or an object of the same class.
4. They have the same call depth.
5. They have the same counter (which is used to indicate the number of occurrences in single-message patterns).

6. The variables that indicate if a message is either the first or last in a (nested) pattern have the same values.

As stated in point 3, it is possible to chose whether the callee of two messages should be either the same object or the same class in order for the messages to be considered equal.

Pattern recognition is an effective abstraction technique as it is able to omit many messages without losing essential information. If a large recurrent contiguous pattern is visualized only once using a repetition construct, the comprehensibility of the particular sequence diagram will potentially increase.

4.3.2 Automatic versus manual abstractions

JRET offers the possibility to apply abstractions automatically based on runtime metrics, manual abstractions selected by the user, as well as a combination of both features. A sequence diagram is considered comprehensible if it contains 100 or fewer calls. This is based on the following:

- For the application of automatic abstractions a threshold should be set to determine whether or not abstractions are necessary. After examining several reconstructed sequence diagrams, we discovered that diagrams having more than around 100 calls became too complex to be properly understood.
- If a sequence diagram contains more than around 100 objects, then scrolling within the visualizers will cause stuttering. The degree of stuttering depends on the total number of objects/calls. Since a sequence diagram having 100 calls contains 101 objects in the worst-case scenario, our threshold guarantees proper navigation within the visualizers.

Automatic abstractions

With the “automatic abstractions” option enabled, the following sequence of abstraction techniques is applied automatically after checking whether the total number of calls is larger than our aforementioned threshold (except 1, 2, and 8):

1. Abbreviations: Method names, return values, and arguments are abbreviated regardless of the number of calls. This prevents the sequence diagrams from becoming excessively expanded.
2. Remove Asserts: This technique is always applied regardless of the number of calls, since Asserts can be omitted without the loss of essential information. After all, Asserts are test case specific and do not add to the understanding of the target system.
3. Remove constructors: This is a potentially effective technique for reducing the number of objects, which is particularly effective in case tests have large fixtures. Moreover, we prefer method calls over constructors calls for program comprehension, since the latter simply involve the creation of objects which are used by the former.

4. Remove getters: Getters usually are lightweight methods which simply return some value, and thus can often be removed without the loss of essential information.
5. Remove setters: The same goes for setters, although these methods set some value rather than returning one. Therefore, we placed this technique below removing getters in the abstraction hierarchy.
6. Stack depth limitation: This is an effective though rigorous technique and therefore placed below the other techniques in the hierarchy. The stack depth is iteratively reduced by 1, after which is checked whether the total number of calls is smaller than or equals 100. The algorithm stops when the stack depth reaches 1.
7. Remove internal messages: This is a rigorous technique which could remove too much information. Since these methods are not restricted by naming conventions, they might very well be the initiators of the entire control flow in a sequence diagram (e.g., when a test case contains a separate method for the actual execution that is directly called from within the test method). Removing such a message, consequently, could remove too much interactions leaving the sequence diagram useless. Therefore, it is only applied in case the sequence diagram still has more than 100 calls after the stack depth has been limited to 1.
8. Pattern recognition (class variant): This is an effective technique which is always applied regardless of the number of calls. It is able to remove many calls, usually without the loss of essential information. In fact, this technique should be given a higher priority in the abstraction hierarchy. However, it uses variables in messages that indicate the beginnings and endings of patterns. Placing other abstraction techniques below pattern recognition, consequently, could remove messages which contain such variables, thereby destroying the identified patterns. Due to time constraints we were not able to update our code in order to overcome this limitation.

Automatic abstractions are useful for the fast generation of a collection of human-readable sequence diagrams. Since these diagrams display a global overview of the behavior of a certain functionality of the system-under-analysis, they are a potentially good starting point for program comprehension.

Manual abstractions

Manual abstractions can be applied with or without the “automatic abstractions” option enabled. If it is enabled, then the manually selected abstraction techniques will be applied after the automatic abstractions are finished. In Figure 4.1, manual abstractions can be selected in the lower right panel. As can be seen, the “look back” value as well as the nesting level parameter of the pattern recognition algorithm can be configured. If these values are changed, then they are also updated for the automatic abstractions (regardless of whether the checkbox in front of the object or class variant is checked).

Manual abstractions are useful in case one knows beforehand which abstraction techniques

should be applied. Moreover, they can be used to study certain parts of a program's behavior in more detail. In this regard, message selection in combination with the "message numbering" option is an effective feature.

4.3.3 Implementation

Like traces, sequence diagrams are internally represented as sequences of `Message` objects. Except for the abbreviation techniques, all abstractions involve the removal of `Message` objects from these sequences. They extend the `Filter` class which has a single method: `updatePhase(Vector, int)`. This method is needed for the labels in sequence diagrams (see Figure 4.4), because a `Message` object contains variables which point out whether or not it is the first message of a test phase (i.e., fixture, actual test, or tear down). Before the removal of a "starting" message, the next message in line should be notified that it has become the first message of the particular test phase. Otherwise, the information is lost.

Abstractions are performed in the `SEditConverter` and `SequenceConverter` classes.

4.4 Visualizing the data

In this section we explain how our tool handles the third phase of the dynamic reconstruction process: the visualization of the abstracted data.

JRET provides a way to generate sequence diagrams in PNG format, as well as output that can be visualized as sequence diagrams using `SEQUENCE` and `SEDIT`. One is able to choose which visualizer to use to either view the diagrams, or generate the PNG files.

4.4.1 Alex Moffat's Sequence

`SEQUENCE` is a lightweight visualization tool written in Java 1.4 to generate sequence diagrams from plain text input files. These input files can be directly converted to PNG format through command-line, or viewed in the visualizer itself. `SEQUENCE`'s main advantages are its simplicity and compatibility with Java 1.4. A sample sequence diagram is shown in Figure 4.7. It is one of the diagrams generated when using `JPACMAN` as target system, which is an experimental tool written for educational purposes [13]. It includes a `JUNIT` 3 test suite having a code coverage of 100%. `SEQUENCE` does not support the visualization of patterns consisting of multiple messages. Nevertheless, it is possible to view single-message patterns (i.e., the first phase of the iterative algorithm described in [23]) as can be seen in Figure 4.7. The constructor call is preceded by a number that is between square brackets indicating the number of occurrences.

4.4.2 Quick Sequence Diagram Editor: SEdit

`SEDIT` is a more advanced visualization tool written in Java 5.0 to generate sequence diagrams from plain text input files. Like previous tool, these input files can be directly con-

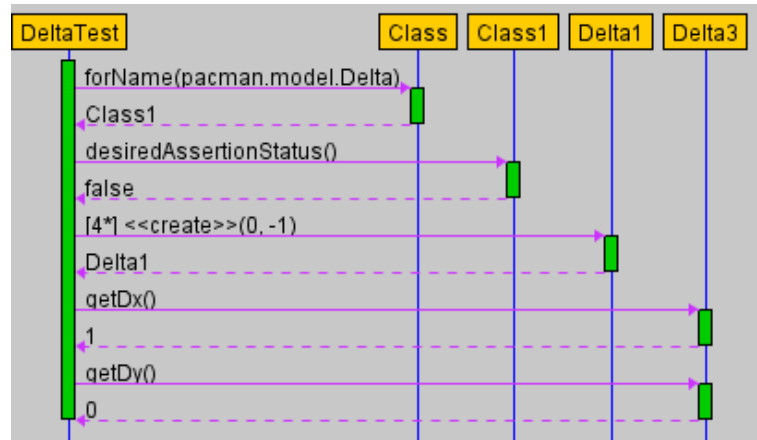


Figure 4.7: A sample sequence diagram in SEQUENCE.

verted to PNG format through command-line, or viewed in the visualizer itself. Although SDEDIT is not compatible with Java 1.4, and does not provide the simplicity that SEQUENCE offers, it has the following advantages:

- Support for patterns consisting of multiple messages.
- Support for labels in the sequence diagrams.
- Zooming functionality.
- Display diagrams in full-screen mode.
- Simultaneously open multiple diagram using tab pages.
- Export diagrams to various formats including eps, pdf, ps, swf, gif, jpeg, and bmp.
- Support for threads.

A sample sequence diagram is shown in Figure 4.8, which has also been derived from JPACMAN's test suite. As can be seen, the sequence diagram in Figure 4.8 contains two labels. The first label indicates that the very first call, i.e., the creation of a new Food object (constructor), is part of the fixture. The second label states that all calls that follow are part of the actual test.

Figure 4.9 depicts another sample sequence diagram in which a recurrent pattern consisting of multiple messages is visualized only once using a repetition construct. Again, it has been derived from JPACMAN's test suite.

Notice that the entire diagram consists of a pattern which normally would have been repeated eight times. Moreover, the fixture label is directly followed by the label that depicts the actual test. This means that the particular test case has a fixture, but all calls belonging

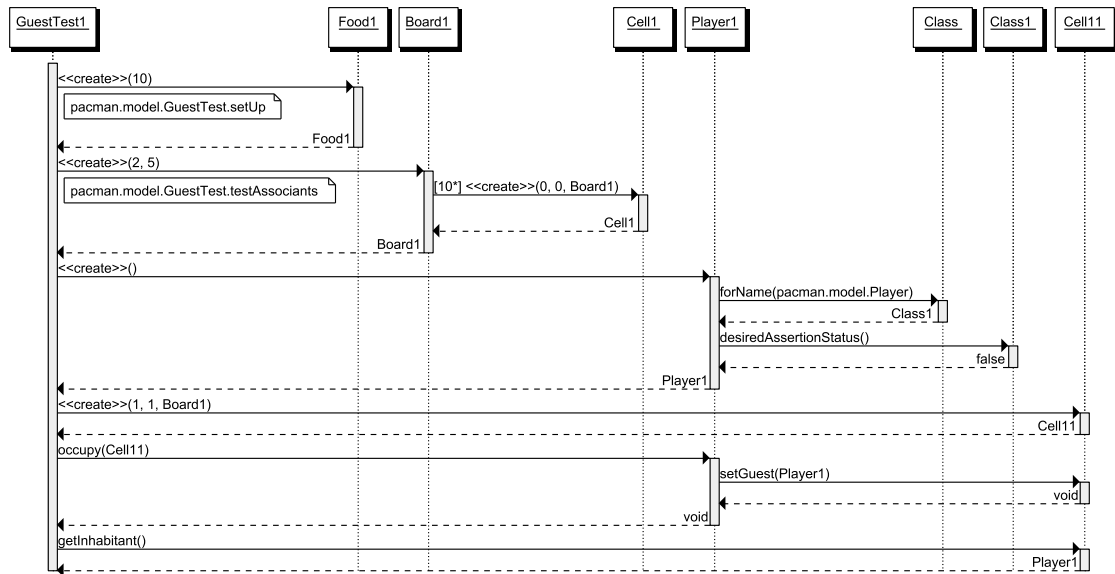


Figure 4.8: A sample sequence diagram in SEDIT.

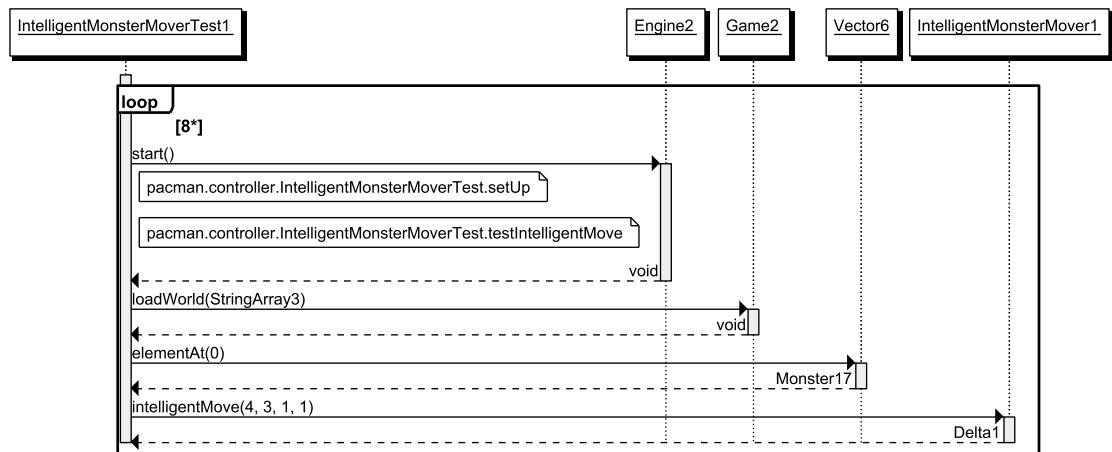


Figure 4.9: Pattern recognition in SEDIT.

to this fixture have been filtered out. Hence, all calls in the diagram are part of the actual test.

4.5 JRET's research package

For each generated sequence diagram a metrics file is created. These plain text files are produced by the MetricsCollector (see section 4.2.4) and contain information on:

- The number of entities (i.e., constructors, objects, (simple) getters, stack depth frequencies, et cetera).
- The number of calls after each consecutively applied abstraction.

These metrics files are generated by default. Additionally, JRET contains a research package providing two options:

1. Export tables containing metrics (HTML or CSV).
2. Export a table containing data on the effectiveness of abstraction techniques (HTML or CSV).

4.5.1 Tables containing metrics

With the “export metrics” option enabled two tables are exported. On the one hand, there is a table showing key statistics for the target program. These statistics include the minimum and maximum value, as well as the median and 75th percentile of the entities in the metrics files (i.e., constructors, methods, objects, (simple) getters, et cetera).

On the other hand, there is a table showing the number of calls after the consecutive application of automatic abstractions per test case.

4.5.2 Table concerning the effectiveness of abstractions

With the “simulation” option enabled one table is exported. Per test case it depicts the effectiveness, in terms of the percentage-wise reduction in calls, of various abstraction techniques in case they are separately applied to the original test. It calculates an average for the entire test suite as well. In order to generate this table, JRET runs a simulation of all implemented abstraction techniques.

4.5.3 Evaluation

The usage of the research package is illustrated as part of a case study, which we conducted in order to evaluate JRET on specific software comprehension tasks using a well-known medium-sized open source program. The case study is presented in the following chapter.

Chapter 5

Case Study: Checkstyle

Up to now we have been concentrating on testing JRET in combination with a relatively small Java program written for educational purposes, i.e., JPACMAN, but now the time has come to test our tool with a larger, more common program that is widely used: CHECKSTYLE [11].

In this chapter we use CHECKSTYLE, which is a development tool to help programmers write Java code that adheres to a coding standard, as an input to JRET and perform some experiments. First, in section 5.1, we give a short introduction to CHECKSTYLE. Secondly, in section 5.2, we use CHECKSTYLE to gather information on the need for abstractions, as well as information on the effectiveness of various abstraction techniques. In section 5.3 and 5.4, we demonstrate how JRET facilitates software maintenance. We do this by showing how JRET simplifies the process of becoming familiar with CHECKSTYLE's inner workings in order to implement two new features:

1. A new export functionality (section 5.3).
2. A new check (section 5.4).

Finally, in section 5.5, we discuss our experiments and results. For this case study we used CHECKSTYLE version 4.4.

5.1 Introduction to Checkstyle

CHECKSTYLE is an extensible framework which can be used to check whether the style of Java code is correct according to a standard specified by the user. The framework is written in Java and consists of 276 classes divided into 21 packages. In order to run CHECKSTYLE one has to specify an XML configuration file that indicates which checks will be executed. These checks can be related to coding, javadoc comments, headers, imports, size violations, modifiers, and so on. The result of an execution is exported to either the standard output or to an XML file as specified by the user.

CHECKSTYLE contains a JUNIT test suite written in JUNIT version 3. It consists of around 500 unit tests. No acceptance tests have been included and according to Rick Giles, one of the CHECKSTYLE developers, neither are there plans to create acceptance tests in the future.

5.2 Checkstyle metrics

In this section we use CHECKSTYLE to gather information on the need for abstractions, as well as information on the effectiveness of various abstraction techniques.

5.2.1 Experimental setup

Since CHECKSTYLE is larger and more complex than JPACMAN, we expect that an increase in Java heap memory of the Virtual Machine will be necessary in order for our tool to run flawlessly. Furthermore, CHECKSTYLE relies on several third party JAR files, that add much overhead in terms of interactions while distracting from the interactions that take place within CHECKSTYLE itself. Some of them cause heap space overflow errors as well. Hence, there is a need to create a custom tracer that traces CHECKSTYLE-related calls only.

We have created this custom tracer by copying one of the default tracers that contains custom pointcuts for filtering out specific calls, placing it in the custom tracers directory (`jret\tracing\tracers\custom`), and redefining its custom pointcuts (see Listing 5.1). For this purpose we have chosen to use the `ObjectTestCase3NoSystemSDEditTracer` aspect, since it contains custom pointcuts and has been designed for JUNIT 3. It traces the interactions at object level and generates output that can be visualized as sequence diagrams using the SDEDIT tool [61].

Of course it is also possible to use a tracer that produces sequence diagrams using the SEQUENCE tool [63], but for this case study we prefer SDEDIT over SEQUENCE because of its advanced features, such as the zooming functionality, which might come in handy. Using the zooming feature it is possible to get an overall overview of the execution of a test case (zooming out), after which certain parts can be studied in more detail by zooming in. Also, SDEDIT offers the possibility to open multiple diagrams using tab pages, which is a very useful feature in case multiple diagrams have to be examined at the same time. Detecting similarities (i.e., patterns) between test cases thus becomes much easier when compared to SEQUENCE in which diagrams have to be opened and opened again. It does not provide the possibility to switch between multiple diagrams.

As can be seen in Listing 5.1, constructor and method calls in packages whose names start with `antlr` and `org` are filtered out. The former represents a third party parser generator that causes heap memory overflow errors during the weaving process, whereas the latter represents multiple components of the Apache Commons project [1]. Moreover, the join-points in the `customWithin` pointcut are necessary as the `grammars` package as well as the

```
protected pointcut customCalls() :
    !cflow(call(* antlr..*(..))) && !cflow(call(* org..*(..)));

protected pointcut customCons() :
    !cflow(call(antlr..*.new(..))) && !cflow(call(org..*.new(..)));

protected pointcut customWithin() :
    !within(com.puppycrawl.tools.checkstyle.grammars..*) &&
    !within(com.puppycrawl.tools.checkstyle.api.Comment);
```

Listing 5.1: Custom pointcuts to trace CHECKSTYLE.

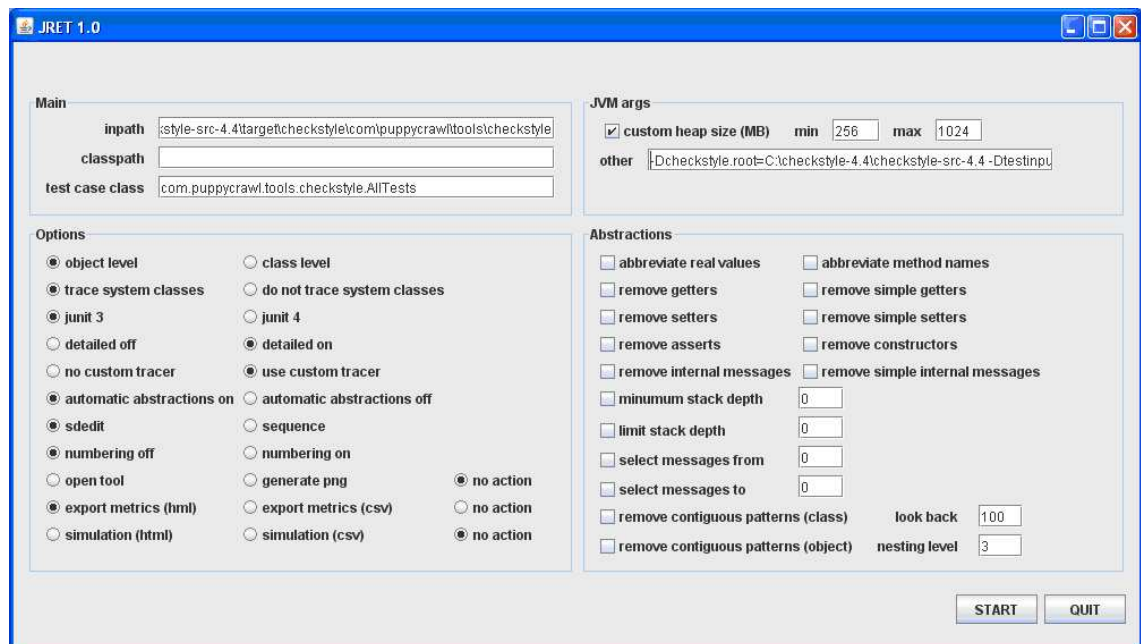


Figure 5.1: Settings when running JRET on CHECKSTYLE.

Comment class of the `api` package raise heap memory related errors while weaving our aspects through the CHECKSTYLE code. The same goes for the `DetailASTTest` class, though this time while executing the tests rather than during the weaving process. We therefore excluded this test from the test suite.

In Figure 5.1 all settings are shown for our initial run.

As can be seen in Figure 5.1, the “classpath” field is empty which means that the default classpath is used. Also, the Java heap space is increased to a maximum of 1024 MB. The other JVM arguments are system variables, namely the location of the test input files, and the location of the CHECKSTYLE root folder, which are needed for CHECKSTYLE’s test suite to run without producing errors. Furthermore, we choose to export metrics concerning

the applied abstractions to HTML. The remaining options speak for themselves.

5.2.2 The need for abstractions

Using this settings 493 trace files and, hence, 493 sequence diagrams are generated in approximately two minutes. Of these tests 132 contain a fixture whereas none of them contain a tear down. Dynamically obtained key statistics during this run are shown in Table 5.1, which is one of the exported tables in case the “export metrics” option has been enabled.

	Min	Max	Median	0.75-Perc.
# Constructors	0	3834	139	259
# Methods	3	65197	2424	5267
# Objects	3	5748	425	812
Max. Depth	2	42	15	18
# Getters	0	31205	1189	2781
# Simple Getters	0	21098	1124	2674
# Setters	0	598	13	14
# Simple Setters	0	595	10	10
# Internal Messages	0	14389	521	1218
# Simple Internal Messages	0	6157	37	91

Table 5.1: Key statistics for the CHECKSTYLE case study.

As can be seen Table 5.1 contains various metrics such as the number of constructors, objects and methods including getters and setters, as well as the maximum stack depth. Moreover, it becomes clear that half of the test cases use 425 or less objects (median 425), and three quarter of the test cases use fewer than 812 objects (0.75-percentile). Likewise, three quarter of the test cases have fewer than 5267 method calls. Also, the test cases contain relatively many getters when compared to the number of setters: three quarter of them have fewer than 2781 getters versus only 14 setters. This implies that removing getters is a potentially effective abstraction technique for the CHECKSTYLE framework.

As a result of the “export metrics” option another table is exported. It depicts the number of calls per test case after the consecutive application of automatic abstractions. Table 5.2 depicts a selection of the entire table that has 493 entries. The selection was made such that it represents a test case from each package.

Notice that Table 5.2 has 18 entries, rather than the total number of packages which equals 21. This is because CHECKSTYLE’s test suite does not cover all packages. As far as the priority of abstractions is concerned, please recall section 4.3.

Without abstractions only two of all eighteen test cases in Table 5.2 are small enough to be comprehended. The other 16 test cases contain more than hundred calls and thus need abstractions in order for them to become human-readable. As far as the entire test suite

	Total Calls	Remove Asserts	Remove Constructors	Remove Getters	Remove Setters	Limit Stack Depth	Remove Patterns (Class)
PackageNamesLoaderTest.testNoFile	9	8					7
api.AbstractViolationReporterTest.testGetMessageBundleWithPackage	9	8					6
checks.NewlineAtEndOfFileCheckTest.testNewlineAtEndOfFile	296	295	204	120	102	96	87
checks.blocks.EmptyBlockCheckTest.testDefault	6530	6519	6078	2064	1971	10	9
checks.coding.IllegalInstantiationCheckTest.testIt	7407	7400	6929	2123	2054	33	29
checks.design.InterfaceIsTypeCheckTest.testDefault	996	994	844	361	304	92	82
checks.duplicates.StrictDuplicateCodeCheckTest.testSmallMin	626	625	525	341	323	63	58
checks.header.HeaderCheckTest.testNoHeader	258	258	183	76			61
checks.imports.ImportOrderCheckTest.testDefault	2160	2155	1990	955	894	78	45
checks.indentation.IndentationCheckTest.testTabs	4720	4718	3992	1202	993	96	83
checks.j2ee.EntityBeanCheckTest.testCreate	6030	6018	5452	2124	2008	18	16
checks.javadoc.JavadocMethodCheckTest.testTags	4140	4139	3306	857	776	69	62
checks.metrics.JavaNCSSCheckTest.test	4631	4626	4309	1852	1687	20	13
checks.naming.MethodNameCheckTest.testDefault	5206	5204	4640	1451	1393	92	82
checks.sizes.FileLengthCheckTest.testOK	5003	5002	4446	1327	1274	85	78
checks.whitespace.WhitespaceAroundTest.testIt	14849	14813	14055	5694	5437	10	9
filters.SuppressElementTest.testDecideByLine	122	122	58				52
grammars.VarargTest.testCanParse	702	701	589	259	201	80	74

Table 5.2: The number of calls after the consecutive application of automatic abstractions to a selection of test cases.

is concerned, of all 493 test cases only 54 (which equals $\sim 11\%$) are small enough to be comprehended without applying abstractions. Hence, in 89% of all test cases abstractions are necessary.

Example diagram

Figure 5.2 depicts an SEDIT sequence diagram of a test case that does not need abstractions. It represents the first test case in Table 5.2. In all diagrams Asserts are removed and contiguous patterns are visualized only once using a repetition construct regardless of the number of calls. As mentioned in section 4.3, this is because usually both abstraction techniques can be applied without loss of essential information.

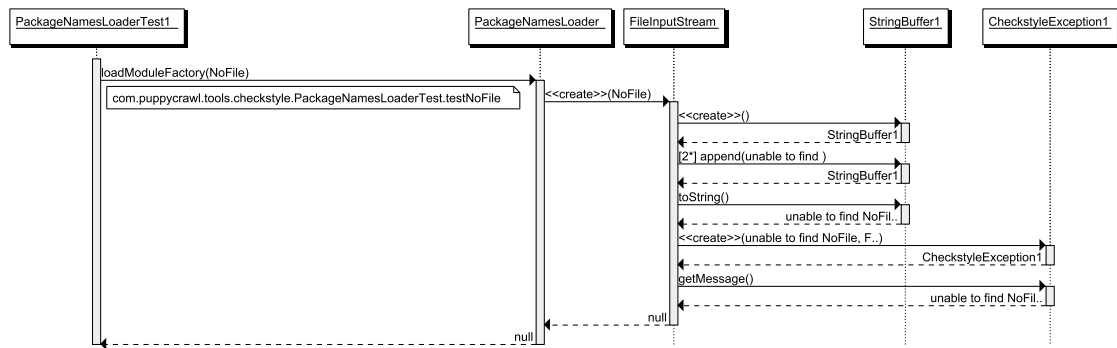


Figure 5.2: A sequence diagram in SEDIT that does not need abstractions.

As can be seen the sequence diagram in Figure 5.2 is rather easy to comprehend. It comprises an attempt to load a file that does not exist after which a so called CheckstyleException is thrown. Notice that the exact error message has been abbreviated in order to prevent the diagram from being excessively expanded. If one wants to know the entire error message, the automatic abstractions option has to be disabled and manual abstractions will have to be selected instead. Figure 5.3 depicts the same diagram in case a SEQUENCE tracer was chosen instead of an SEDIT tracer.

Notice that the diagrams in Figure 5.2 and Figure 5.3 are quite similar. Apart from the graphical and coloring techniques, the only difference between both diagrams is the label in Figure 5.2 which indicates that all calls belong to the testNoFile test case. This is because Alex Moffat’s SEQUENCE does not support this feature.

5.2.3 Effectiveness of abstractions

If we run JRET again using the same settings plus the “simulation” option enabled, another table is exported, which contains information on the effectiveness of various abstraction techniques in case they are separately applied to the original trace. Thus, in contrast to Table 5.2, these abstractions are not applied consecutively. Table 5.3 depicts a selection of the

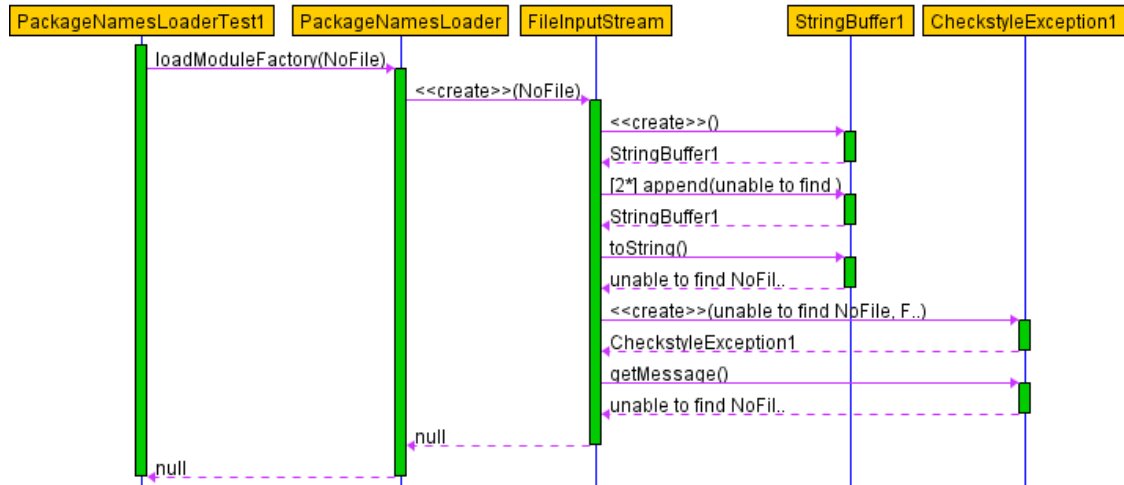


Figure 5.3: The sequence diagram in Figure 5.2 when using SEQUENCE.

entire table that has 493 entries. It contains the same test cases as in Table 5.2.

As can be seen in Table 5.3, removing getters indeed appears to be an effective abstraction technique that yields an average of 51% reduction in calls. Removing internal messages and limiting the stack depth to 1, on the other hand, appear to be even more effective techniques yielding an average of 90% and 95% reduction in calls, respectively. As far as removing internal messages is concerned, the effectiveness of this technique most likely is the result of many test cases being initiated by such an internal message, and removing them will remove their control flow as well. A major limitation of this technique, however, is the fact that too much information could be removed such that the diagrams become useless. Consequently, we placed this technique after stack depth limitation in our abstraction hierarchy.

The effectiveness of limiting the stack depth to 1 implies that most interactions take place at a higher depth. Also, from Table 5.1 it becomes clear that the minimum value of the maximum stack depth equals 2, which means that test cases having calls at depth 1 only do not exist in CHECKSTYLE's test suite.

5.2.4 Conclusion

With this experiment we have shown that:

1. Abstraction techniques are necessary when dynamically obtaining sequence diagrams from a JUnit test suite: In 89% of all traces abstractions were needed in order for the resulting sequence diagrams to become human-readable.
2. Limitation of the call depth to 1, removing internal messages, and removing getters are the most effective abstraction techniques in quantitative aspect. They yield an average reduction in calls of 95%, 90%, and 51%, respectively.

	Total Calls	Remove Constructors	Remove Asserts	Remove Getters	Remove Simple Getters	Remove Setters	Remove Simple Setters	Remove Internal Msgs	Remove Simple Internal	Limit Stack Depth = 1	Remove Patterns (Class)	Remove Patterns (Object)
PackageNamesLoaderTest.testNoFile	9	88%	11%	11%	11%	0%	0%	0%	0%	88%	11%	11%
api.AbstractViolationReporterTest.testGetMessageBundleWithPackage	9	11%	11%	88%	0%	0%	0%	0%	0%	77%	22%	22%
checks.NewlineAtEndOfFileCheckTest.testNewlineAtEndOfFile	296	30%	0%	32%	21%	22%	2%	89%	3%	95%	90%	2%
checks.blocks.EmptyBlockCheckTest.testDefault	6530	6%	0%	61%	51%	3%	0%	99%	1%	99%	49%	21%
checks.coding.IllegalInstantiationCheckTest.testIt	7407	6%	0%	65%	44%	2%	0%	99%	0%	99%	49%	19%
checks.design.InterfaceIsTypeCheckTest.testDefault	996	15%	0%	49%	36%	16%	1%	98%	1%	99%	34%	16%
checks.duplicates.StrictDuplicateCodeCheckTest.testSmallMin	626	15%	0%	31%	12%	10%	0%	97%	2%	98%	38%	9%
checks.header.HeaderCheckTest.testNoHeader	258	29%	0%	43%	29%	60%	1%	98%	0%	99%	13%	4%
checks.imports.ImportOrderCheckTest.testDefault	2160	7%	0%	48%	40%	7%	0%	99%	1%	99%	31%	11%
checks.indentation.IndentationCheckTest.testTabs	4720	15%	0%	63%	30%	15%	0%	99%	8%	99%	26%	7%
checks.j2ee.EntityBeanCheckTest.testCreate	6030	9%	0%	56%	42%	6%	0%	99%	1%	99%	50%	21%
checks.javadoc.JavadocMethodCheckTest.testTags	4140	20%	0%	63%	23%	4%	0%	99%	2%	99%	47%	24%
checks.metrics.JavaNCSSCheckTest.test	4631	6%	0%	53%	44%	7%	0%	99%	1%	99%	36%	12%
checks.naming.MethodNameCheckTest.testDefault	5206	10%	0%	61%	51%	3%	0%	99%	0%	99%	59%	21%
checks.sizes.FileLengthCheckTest.testOK	5003	11%	0%	62%	52%	3%	0%	100%	0%	99%	63%	22%
checks.whitespace.WhitespaceAroundTest.testIt	14849	5%	0%	56%	47%	3%	0%	99%	4%	99%	40%	16%
filters.SuppressElementTest.testDecideByLine	122	50%	1%	28%	12%	48%	0%	22%	2%	90%	13%	4%
grammars.VarargTest.testCanParse	702	15%	0%	48%	37%	24%	1%	93%	1%	99%	27%	10%
Average (493 test cases)	5021	13%	1%	51%	36%	8%	0%	90%	2%	95%	37%	16%

Table 5.3: The effectiveness of various abstraction techniques when separately applied to the original trace.

5.3 Understanding exports in Checkstyle

In this section we engage JRET to facilitate a software maintenance task. This task involves the implementation of a new export feature: The possibility to export CHECKSTYLE output data to HTML. Rather than inspecting CHECKSTYLE's source code, as would be the case in static analysis, we use JRET because of the following reasons:

- JRET offers the possibility to get a global insight into CHECKSTYLE's inner workings, whereas at the same time certain parts can be studied in the highest detail possible. Specifically, the runtime objects, as well as the actual method arguments and return values are available. It is therefore possible to expose occurrences of late binding, for instance.
- Analyzing program behavior through a sequence diagram is more intuitive than source code analysis because of the chronological ordering. Moreover, a single sequence diagram is able to capture the interactions that take place between multiple (objects of) classes. By using source code alone, one would have to switch between multiple files, which makes the task more complex and time-consuming.
- Test suites, typically written in JUNIT, provide various execution possibilities for the system at hand. Unit tests induce scenarios that effectively decompose a system's functionalities. Thus, a careful selection of test cases effectively focuses attention to a particular feature / functionality of the system to be studied. Looking up appropriate source files, on the other hand, could be an extensive task on itself.

In the following subsection we give an example of a CHECKSTYLE export to XML. In subsection 5.3.2 the diagram selection phase is described. Subsection 5.3.3 concerns the analysis phase, followed by subsection 5.3.4 in which we implemented the new export feature. Finally, in subsection 5.3.5 a conclusion is presented.

5.3.1 XML export example

CHECKSTYLE provides the possibility to export the result of an audit to either the standard output (stdout) or an XML file. If we run CHECKSTYLE using the `sun_checks.xml` configuration file and `Hello.java` (see Listing 5.2) as the file to be audited, and export the result to XML, it will look like Figure 5.4 when viewed in an internet browser.

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

Listing 5.2: Simple Java file `Hello.java`.

```

<?xml version="1.0" encoding="UTF-8" ?>
- <checkstyle version="4.4">
- <file name="C:\checkstyle-4.4\checkstyle-4.4\package.html">
  <error line="0" severity="error" message="Missing package documentation file." source="com.puppycrawl.tools.checkstyle.checks.javadoc.PackageHtmlCheck" />
</file>
<file name="Hello.java" />
- <file name="Hello.java">
  <error line="1" severity="error" message="Missing a Javadoc comment." source="com.puppycrawl.tools.checkstyle.checks.javadoc.JavadocTypeCheck" />
  <error line="1" column="1" severity="error" message="Utility classes should not have a public or default constructor."
    source="com.puppycrawl.tools.checkstyle.checks.design.HideUtilityClassConstructorCheck" />
  <error line="2" column="1" severity="error" message="'{' should be on the previous line."
    source="com.puppycrawl.tools.checkstyle.checks.blocks.LeftCurlyCheck" />
  <error line="3" column="3" severity="error" message="Missing a Javadoc comment."
    source="com.puppycrawl.tools.checkstyle.checks.javadoc.JavadocMethodCheck" />
  <error line="3" column="27" severity="error" message="Parameter args should be final."
    source="com.puppycrawl.tools.checkstyle.checks.FinalParametersCheck" />
  <error line="4" column="3" severity="error" message="'{' should be on the previous line."
    source="com.puppycrawl.tools.checkstyle.checks.blocks.LeftCurlyCheck" />
  <error line="7" severity="error" message="Line has trailing spaces." source="com.puppycrawl.tools.checkstyle.checks.GenericIllegalRegexpCheck" />
</file>
</checkstyle>

```

Figure 5.4: XML result of an audit using Hello.java.

Since JRET provides the possibility to export various metrics to HTML, we would like to implement such a feature in CHECKSTYLE as well, i.e., the possibility to export the result of an audit to HTML. Let us engage JRET to facilitate this exercise, since we are not familiar with CHECKSTYLE’s inner workings.

5.3.2 Selection of diagrams

A first glance at the generated sequence diagrams tells us that the largest part concerns CHECKSTYLE-checks in which we are not interested in this section. The names of these diagrams start with `com.puppycrawl.tools.checkstyle.checks` which indicates that they are part of the `com.puppycrawl.tools.checkstyle.checks` package (or one of its subpackages). If we take a peek at the source files in this package, then most files happen to have the word “Check” in their name. Also they extend some sort of Check class, e.g., `Check` or `AbstractFileCheck`.

Since we intend to implement an HTML export functionality, alternatively formulated as an HTML logging feature, next to the already existing XML logging feature, we intuitively inspect the `XMLLoggerTest` test cases.

5.3.3 Analyzing the diagrams

Table 5.4 depicts the number of calls after the consecutive application of automatic abstractions to the `XMLLoggerTest` test cases.

Half of these eight test cases are comprehensible without the application of abstractions. If we take a look at the sequence diagram of the `testFileStarted` test case, we first notice that it has a fixture in which a `ByteArrayOutputStream` is created. The actual test consist of the following actions:

1. An `XMLLogger` object is created.

	Total Calls	Remove Asserts	Remove Constructors	Remove Getters	Remove Setters	Limit Stack Depth	Remove Patterns (Class)
XMLLoggerTest.testEncode	127	120	101	101	101	93	40
XMLLoggerTest.testIsReference	121	112	98				64
XMLLoggerTest.testCloseStream	52	48					40
XMLLoggerTest.testNoCloseStream	50	46					38
XMLLoggerTest.testFileStarted	66	61					48
XMLLoggerTest.testFileFinished	60	55					43
XMLLoggerTest.testAddError	277	272	243	29			12
XMLLoggerTest.testAddException	268	259	237	205	205	43	21

Table 5.4: The number of calls after the consecutive application of automatic abstractions to the XMLLoggerTest test cases.

2. An `auditStarted(null)` method is executed on this object, which in turn yields several calls that build and print a `String`. Using their actual parameters, we identify this `String` to be the first two lines of the XML file in Figure 5.4.
3. An `AuditEvent` object is created.
4. A `fileStarted(AuditEvent)` method is executed on the `XMLLogger` object. This method also yields several calls that build and print a `String`. Using their actual parameters, we identify this `String` to be the name of the file that is being audited.
5. An `auditFinished(null)` method is executed on the `XMLLogger` object. This method yields one call that prints the last line of our XML output in Figure 5.4. Again, this `String` has been identified using the call's actual parameters.
6. Some verification method calls which check whether the created `Strings` are as expected. After all, we are dealing with a test case which are usually concluded by some sort of verification phase.

Thus, it indeed appears that we have chosen the right part of test cases. It has become clear that the `XMLLogger` class contains methods that generate XML output (as in Figure 5.4) and, hence, is (one of) our class(es) of interest.

If we take a look at the `testFileFinished` test case, we notice that it is quite similar to the `testFileStarted` test case. The main difference between both diagrams lies in point 4. The `fileStarted(AuditEvent)` method is replaced by a `fileFinished(AuditEvent)` method which, on its turn, yields one line being printed, namely an XML file closing tag (identified using the call's actual parameters).

The sequence diagram of the `testAddError` test case is shown in Figure 5.5.

In contrast to the `testFileStarted` and `testFileFinished` test cases the fixture has been removed: the fixture label in Figure 5.5 is directly followed by the label that indicates that the following calls belong to the actual test. This is because removing constructors is one of the abstraction techniques that have been applied to this test case. Nevertheless, one could easily see a similarity to both aforementioned test cases, namely the execution of the `auditStarted(null)` method on the `XMLLogger` object. Here it is followed by the execution of an `addError(AuditEvent)` method on the `XMLLogger` object. It yields several calls that build and print an error message (see the actual parameters of the `append(String)` and `print(String)` methods in Figure 5.5).

The `testAddExeption` test case, on its turn, is quite similar to the `testAddError` test case. The main difference is the `addError(AuditEvent)` method being replaced by an `addException(AuditEvent, Throwable)` method which yields several method calls that build and print an exception (identified using their actual parameters).

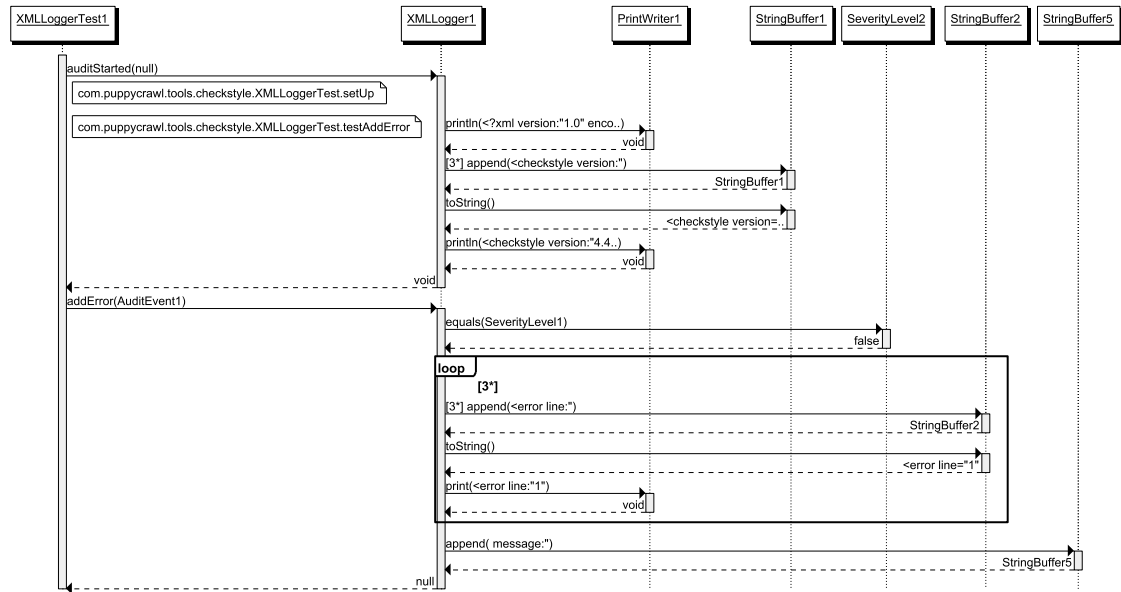


Figure 5.5: Sequence diagram of the testAddError test case.

5.3.4 Implementation

From all this we have learned that an XMLLogger object is some kind of listener that listens to the following events:

1. auditStarted(null)
2. fileStarted(AuditEvent)
3. addError(AuditEvent)
4. addException(AuditEvent, TestThrowable)
5. fileFinished(AuditEvent)
6. auditFinished(null)

Now all we have to do is create a new Java class, which we name HTMLLogger, and implement the actions that should be performed in case one of these six events occurs. As we have learned these actions only involve building and printing Strings, and in our case these Strings should contain HTML text.

The actual implementation of our HTMLLogger class turned out to be rather easy, since we were able to reuse most of the code of the XMLLogger class. Additionally, we included a counter to keep track of the number of errors per file, as well as the total number of errors in an audit. After the insertion of our new listener in the CHECKSTYLE Main class, the new

feature was ready to be tested. The result of an audit using the `sun_checks.xml` configuration file and `Hello.java` (Listing 5.2) as the file to be audited, together with our new export feature, resulted in Figure 5.6 when viewed in an internet browser.

checkstyle version 4.4

File: C:\Documents and Settings\Administrator\Bureau\blad\checkstyle-src-4.4\target\checkstyle\com\puppcrawl\tools\checkstyle\package.html

- Error on line 0 where severity equals error, message is "Missing package documentation file.", and source is com.puppcrawl.tools.checkstyle.checks.javadoc.PackageHtmlCheck.

Total number of errors: 1

File: com\puppcrawl\tools\checkstyle\Hello.java

Total number of errors: 0

File: com\puppcrawl\tools\checkstyle\Hello.java

- Error on line 1 where severity equals error, message is "Missing a Javadoc comment ", and source is com.puppcrawl.tools.checkstyle.checks.javadoc.JavadocTypeCheck.

- Error on line 1 at column 1 where severity equals error, message is "Utility classes should not have a public or default constructor.", and source is com.puppcrawl.tools.checkstyle.checks.design.HideUtilityClassConstructorCheck.

- Error on line 2 at column 1 where severity equals error, message is "'(' should be on the previous line.", and source is com.puppcrawl.tools.checkstyle.checks.blocks.LeftCurlyCheck.

- Error on line 4 at column 3 where severity equals error, message is "Missing a Javadoc comment.", and source is com.puppcrawl.tools.checkstyle.checks.javadoc.JavadocMethodCheck.

- Error on line 4 at column 27 where severity equals error, message is "Parameter args should be final.", and source is com.puppcrawl.tools.checkstyle.checks.FinalParametersCheck.

- Error on line 5 at column 3 where severity equals error, message is "'(' should be on the previous line.", and source is com.puppcrawl.tools.checkstyle.checks.blocks.LeftCurlyCheck.

- Error on line 8 where severity equals error, message is "Line has trailing spaces.", and source is com.puppcrawl.tools.checkstyle.checks.GenericIllegalRegexpCheck.

Total number of errors: 7

Total number of errors in all files: 8

Figure 5.6: HTML result of an audit using `Hello.java`.

5.3.5 Conclusion

With this experiment we have shown that the sequence diagrams generated by JRET are useful for analyzing the behavior of the CHECKSTYLE export functionality. We discovered that the implementation of a new new export feature, i.e., a new logger, is rather easy. It consist of writing a new Logger class that listens to six different events:

1. `auditStarted(null)`
2. `fileStarted(AuditEvent)`
3. `addError(AuditEvent)`
4. `addException(AuditEvent, TestThrowable)`
5. `fileFinished(AuditEvent)`
6. `auditFinished(null)`

The actions to be performed in case these events occur simply involve printing Strings in the export language of your choice. These Strings were identified using the actual parameters of String related method calls such as `append(String)` and `print(String)`. For this implementation it is possible to reuse most of the code of the existing `XMLLogger` class. Finally, the new listener has to be inserted in CHECKSTYLE's Main class.

5.4 Understanding checks in Checkstyle

In this section we engage JRET to facilitate another software maintenance task. We use our tool to expose how CHECKSTYLE-checks work in order to simplify the implementation of a new check. As already stated in the previous section, determining this behavior only by inspecting all source files most likely would be a complex and time-consuming task. Using JRET it is possible to:

- Get a global overview of the inner workings of CHECKSTYLE-checks.
- If necessary, study certain parts in more detail. In contrast to static analysis, the run-time objects, as well as the actual method arguments and return values are available.

Please refer to section 5.3 for a more detailed overview concerning the advantages of our tool over static analysis.

In the following subsection we describe the diagram selection phase. Subsection 5.4.2 concerns the diagram analysis phase, followed by subsection 5.4.3 in which we implemented the new check. Finally, a conclusion is presented in subsection 5.4.4.

5.4.1 Selection of diagrams

Recall that the largest part of all generated sequence diagrams concerns CHECKSTYLE-checks, in which we are interested in this section as opposed to the former one. These are the diagrams whose names start with `com.puppycrawl.tools.checkstyle.checks` which indicates that they belong to the `com.puppycrawl.tools.checkstyle.checks` package (or one of its subpackages). This goes for 411 of all 493 sequence diagrams.

Instead of inspecting all 411 diagrams, we randomly pick five of them under the condition that their names clearly imply what the checks are supposed to do. Moreover, we make sure multiple (sub)packages are included in our selection. We picked the following ones:

- `checks.coding.PackageDeclarationCheckTest.testDefault`
- `checks.coding.DefaultComesLastCheckTest.testIt`
- `checks.UncommentedMainCheckTest.testDefaults`
- `checks.NewlineAtEndOfFileCheckTest.testNoNewlineAtEndOfFile`
- `checks.imports.AvoidStarImportTest.testDefaultOperation`

These checks, taken from three (sub)packages, involve testing if a package declaration exists, testing if default comes after the last case in a switch statement, testing if there exists a main method without comments, testing if a Java file ends with a newline character, and testing if there are import statements using an asterisk, respectively.

5.4.2 Analyzing the diagrams

Identify patterns between diagrams

Table 5.5 shows the number of calls after the consecutive application of automatic abstractions to our selection of test cases.

One thing worth mentioning when looking at Table 5.5 is that four of all five test cases happen to have the same number of calls (96) after limitation of the stack depth. This suggests that these four test cases follow a similar pattern below some stack depth threshold, although this does not necessarily have to be true.

First four diagrams

If we take a look at the sequence diagrams of these four test cases, then it indeed appears that they follow a similar pattern. One of them, the visualization of the `UncommentedMainCheckTest.testDefaults` test case, is shown in Figure 5.7. The identified pattern, in summary, consists of the following steps:

1. A `Checker` object is created and configured by the test class.
2. A listener is added to the `Checker` object.
3. The `Checker` object executes a `fireAuditStarted()` method on itself.
4. The `Checker` object notifies its listener using an `auditStarted(AuditEvent)` method call.
5. The `Checker` object executes a `process(File[])` method on a `TreeWalker` object.
6. The `TreeWalker` object initiates several calls. These calls, however, are not the same in all diagrams. They all start with a `filter(File[])` method, which is followed by a `process(File)` method call on the `TreeWalker` object. However, in only two of all four diagrams the control flow of this `process(File)` internal message is visible (which includes performing a `fireFileStarted(String)` and `fireFileFinished(String)` method call, as well as a `walk(DetailAST, FileContents)` method execution on the `TreeWalker` object itself).
7. The `Checker` object destroys the `TreeWalker` object.
8. The `Checker` object performs a `fireAuditFinished()` method call.
9. The `Checker` object notifies its listener using an `auditFinished(AuditEvent)` method call.
10. An error message is built.
11. The `Checker` object is destroyed.

	Total Calls	Remove Asserts	Remove Constructors	Remove Getters	Remove Setters	Limit Stack Depth	Remove Patterns (Class)
PackageDeclarationCheckTest.testDefault	576	574	462	242	185	96	84
DefaultComesLastCheckTest.testIt	1451	1449	1305	491	434	96	84
UncommentedMainCheckTest.testDefaults	3871	3867	3585	1376	1310	96	70
NewlineAtEndOfFileCheckTest.testNoNewlineAtEndOfFile	344	342	266	154	136	76	65
AvoidStarImportTest.testDefaultOperation	4340	4336	4062	1630	1573	96	70

Table 5.5: The number of calls after the consecutive application of automatic abstractions to five random test cases.

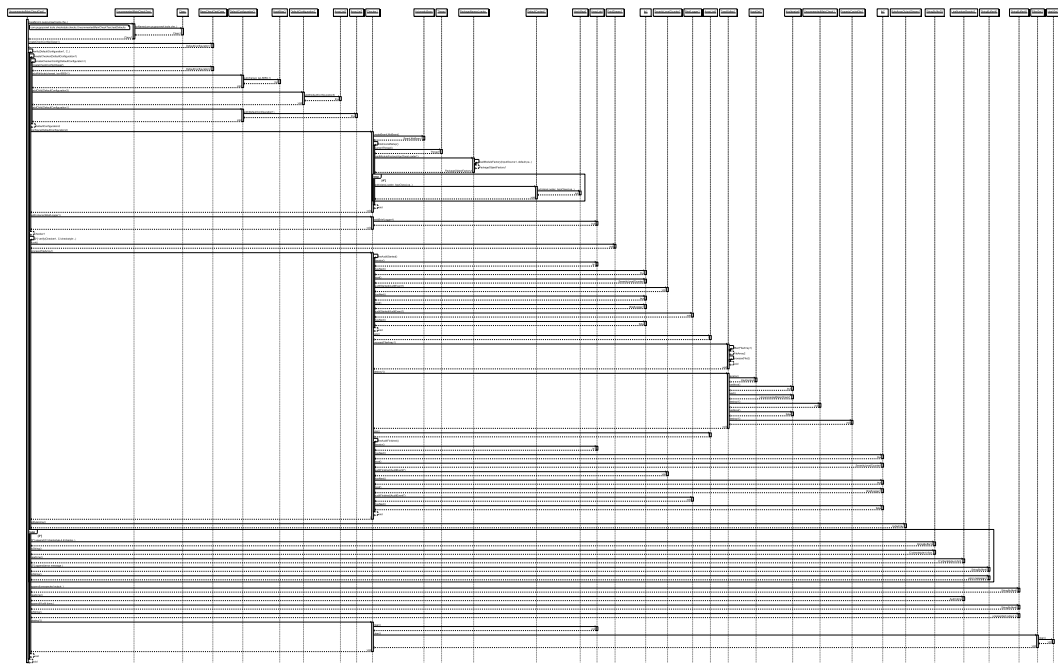


Figure 5.7: Visualization of the `UncommentedMainCheckTest.testDefaults` test case.

Pattern analysis

The first two steps in this pattern are initiated by the test class and appear to be some sort of initialization steps prior to the actual execution of a check. During this phase a `Checker` object is created and a listener is added to it. Thereafter, the audit is started and a `process(File[])` method is executed on a `TreeWalker` object. This call has a `File[]` object as parameter, which most likely contains the files to be audited. The `TreeWalker` object then, on its turn, executes a `process(File)` method on itself, having a single file as parameter.

However, in only two of all four diagrams the control flow of this `process(File)` internal message is visible (see Figure 5.8 and Figure 5.9), which might be due to an applied abstraction. It includes executing a `fireFileStarted()` and a `fireFileFinished()` method on the `Checker` object, as well as a `walk(DetailAST, FileContents)` method on the `TreeWalker`

object itself.

Since these calls are in the control flow of the `process(File)` method, they could have been filtered out in the diagrams in which they do not exist due to stack depth limitation. In order to verify this, we take a look at the generated metrics files. The maximum stack depth of the diagrams in which the control flow is not visible equals 6 versus 7 of the remaining diagrams. Hence, stack depth limitation indeed appears to be the cause. In Listing 5.3 (part of) the metrics file of one of the diagrams with visible control flow is shown. Listing 5.4, on the other hand, concerns a metrics file of a diagram without control flow.

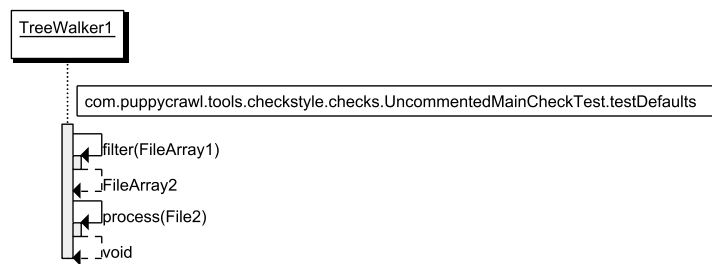


Figure 5.8: `process(File)` internal message without control flow.

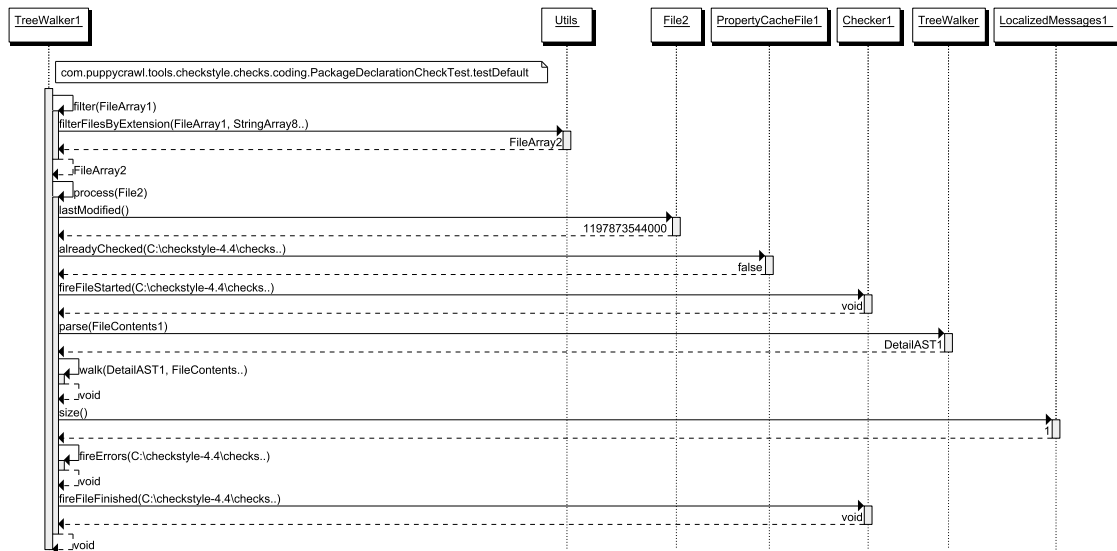


Figure 5.9: `process(File)` internal message with its control flow.

Taking this all into account, it thus seems that we have to study the method calls that are initiated by the `TreeWalker` object in more detail. After all, the differences between the four diagrams are related to these calls.

```

abbreviate real values
abbreviate method names
remove asserts => #objects = 198, #calls = 574
remove constructors => #objects = 145, #calls = 462
remove getters => #objects = 72, #calls = 242
remove setters => #objects = 58, #calls = 185
limit stack depth: 13 => #objects = 58, #calls = 185
limit stack depth: 12 => #objects = 58, #calls = 184
limit stack depth: 11 => #objects = 58, #calls = 183
limit stack depth: 10 => #objects = 58, #calls = 182
limit stack depth: 9 => #objects = 51, #calls = 167
limit stack depth: 8 => #objects = 47, #calls = 129
limit stack depth: 7 => #objects = 41, #calls = 96
remove contiguous sequence patterns (class) => #objects = 41, #calls = 84

```

Listing 5.3: Part of a metrics file (stack depth limited to 7).

```

abbreviate real values
abbreviate method names
remove asserts => #objects = 585, #calls = 3867
remove constructors => #objects = 418, #calls = 3585
remove getters => #objects = 152, #calls = 1376
remove setters => #objects = 139, #calls = 1310
limit stack depth: 17 => #objects = 139, #calls = 1302
limit stack depth: 16 => #objects = 139, #calls = 1290
limit stack depth: 15 => #objects = 135, #calls = 1275
limit stack depth: 14 => #objects = 131, #calls = 1262
limit stack depth: 13 => #objects = 130, #calls = 1253
limit stack depth: 12 => #objects = 91, #calls = 1159
limit stack depth: 11 => #objects = 73, #calls = 1070
limit stack depth: 10 => #objects = 73, #calls = 1050
limit stack depth: 9 => #objects = 56, #calls = 893
limit stack depth: 8 => #objects = 50, #calls = 141
limit stack depth: 7 => #objects = 44, #calls = 108
limit stack depth: 6 => #objects = 38, #calls = 96
remove contiguous sequence patterns (class) => #objects = 34, #calls = 70

```

Listing 5.4: Part of a metrics file (stack depth limited to 6).

Remaining (fifth) diagram

Before doing that, let us first examine the remaining sequence diagram of the `testNoNewlineAtEndOfFile` test case. It appears to be quite similar to the other four diagrams and contains a pattern that is rather equivalent to the one outlined in steps 1 to 11. The main difference lies in step 5: the `process(File[])` method is executed on a `NewLineAtEndOfFileCheck` object rather than on a `TreeWalker` object.

This `process(File[])` method yields several calls including:

1. `filter(File[])`
2. `fireFileStarted(String)`
3. `endsWithNewLine(RandomAccessFile)`

4. `log(..)`
5. `fireErrors(String)`
6. `fireFileFinished(String)`

The entire control flow is depicted in Figure 5.10.

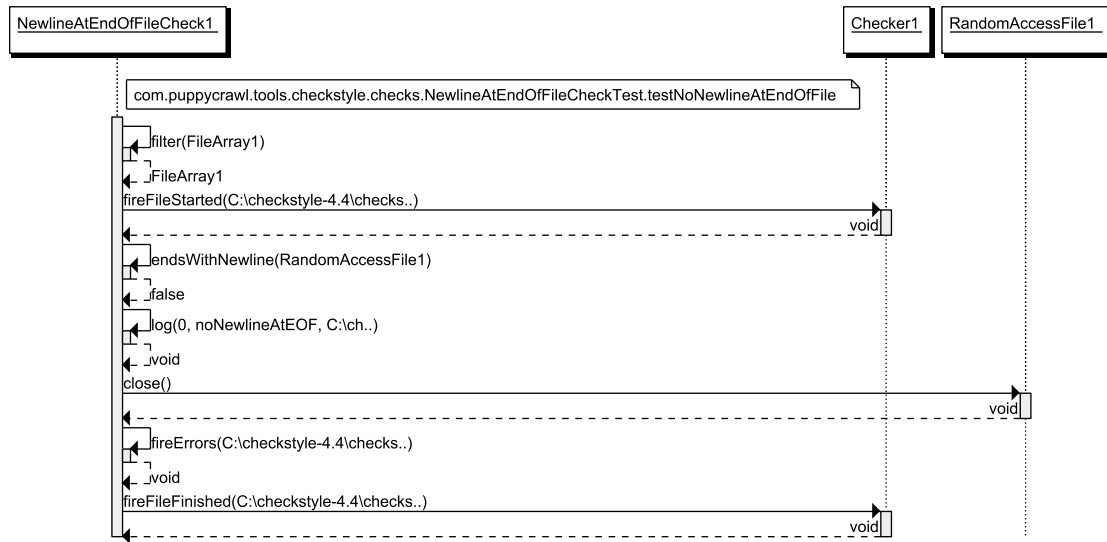


Figure 5.10: Calls in the control flow of the `process(FileArray)` method call performed by the `Checker` object on the `NewLineAtEndOfFileCheck` object.

Now if we compare Figure 5.9 and Figure 5.10, another pattern can be identified. This pattern starts with a `filter(File[])` and `fireFileStarted(String)` method, and ends with a `fireErrors(String)` and `fireFileFinished(String)` method. In between those calls there is an internal message that, most likely as its name implies, initiates the main action of the specific check. In Figure 5.9 it concerns the `walk(DetailAST, FileContents)` method of the `TreeWalker` object, and in Figure 5.10 it concerns the `endsWithNewline(RandomAccessFile)` method of the `NewLineAtEndOfFileCheck` object.

If we take a peek at (the comments in) the source code of these methods, then the former appears to involve walking an AST [33] generated from the contents of the file to be audited. The latter involves checking whether the file's contents ends with the platform specific line separator. Hence, both methods seem to be important for the specific checks, and thus need to be studied in more detail.

Checks in more detail

Now let us examine the `walk(DetailAST, FileContents)` method call of the `TreeWalker` object in more detail. To do so, we pick one of the test cases that contains this method call

(e.g., `PackageDeclarationCheckTest.testDefault`) and run JRET having this test case as input. This time, however, we also enable the “message numbering” option. The rest of the settings remain the same as in Figure 5.1.

The methods in the generated sequence diagram are now annotated by numbers that are between square brackets. These values indicate the numbers of the messages in the original trace (i.e., before any abstractions are applied). If we lookup the `walk(DetailAST, FileContents)` method, we notice that its corresponding message number equals 366 (see the fragment in Figure 5.11). It is followed by a `size()` method, which is annotated by number 460. Consequently, 94 messages have been omitted in the diagram due to abstractions (i.e., 460 subtracted by 366). Now it is time to run JRET again using the options as in Figure 5.12.

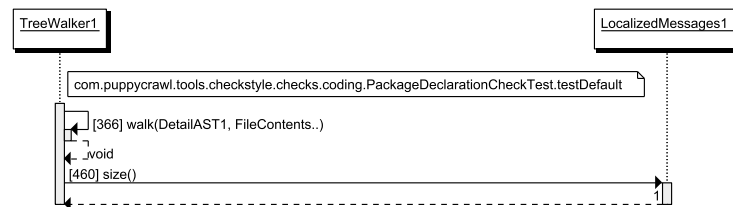


Figure 5.11: The `walk(DetailAST, FileContents)` method call including its message number.

As can be seen in Figure 5.12, we select 94 messages from message number 366 to 460 in order to study the `walk(DetailAST, FileContents)` method call in more detail. These are the messages that have been omitted due to applied abstractions. Recall from chapter 4 that a sequence diagram consisting of 94 messages is considered human-readable (i.e., ≤ 100 calls). If more than hundred messages had been filtered out, we could have solved this by performing multiple runs (thereby producing multiple diagrams), or by studying only the first 100 messages in more detail, for example. As far as our run is concerned, notice that we select both abbreviation abstraction techniques in order for the diagram to remain comprehensible, and disable automatic abstractions. After all, we want our selected fragment to remain intact (i.e., without messages being omitted), preserving the details.

At this point, let us examine the generated sequence diagram using the settings in Figure 5.12. The `walk(DetailAST, FileContents)` method call consists of 94 messages and therefore, as already stated, fits in a single (comprehensible) sequence diagram. It has a `DetailAST` and a `FileContents` object as parameters, and indeed appears to involve walking an abstract syntax tree (AST).

In short, it consists of a `setFileContents(FileContents)` method call on a `PackageDeclarationCheck` object, followed by a `beginTree(DetailAST)` method call on the same object. After that the AST is walked using `notifyVisit(DetailAST)` and

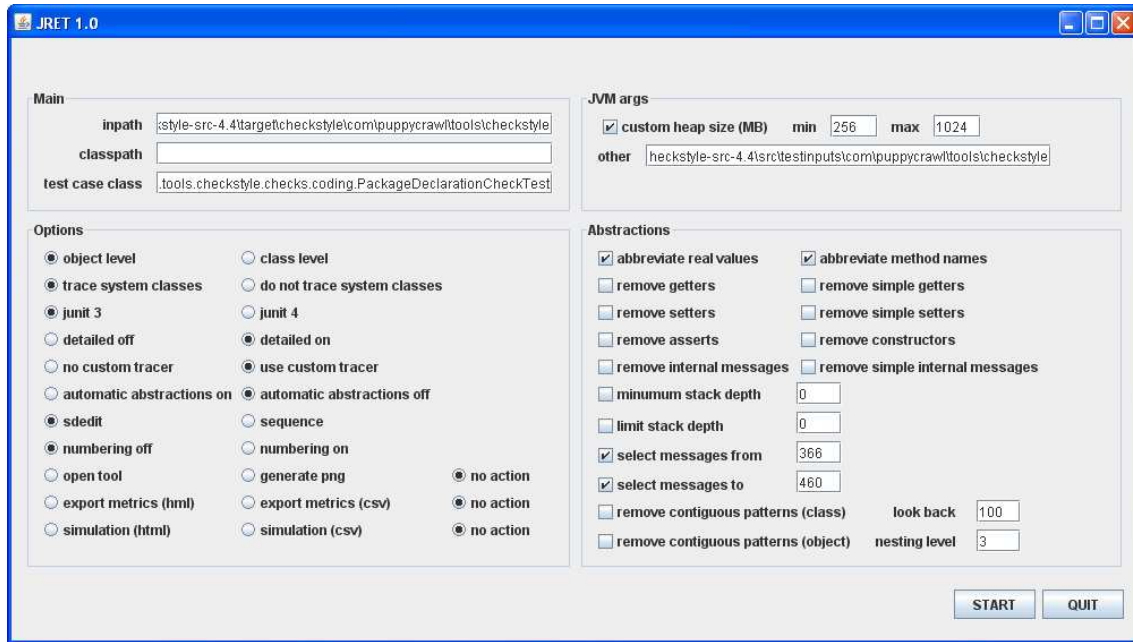


Figure 5.12: Settings for studying the `walk(DetailAST, FileContents)` method call in more detail.

`notifyLeave(DetailAST)` methods to visit and leave the nodes, respectively. The `getTokenName(int)` method is used to retrieve the tokens. Among these are a `CLASS_DEF`, a `MODIFIERS`, and an `OBJBLOCK` token. The walk of the tree is concluded by a `finishTree(DetailAST)` method executed on the `PackageDeclarationCheck` object, which initiates several calls that log error messages. The entire `walk(DetailAST, FileContents)` is depicted in Figure 5.13.

Since the sequence diagram shown in Figure 5.13 was obtained dynamically, we are able to examine the `walk(DetailAST, FileContents)` method call in the highest detail possible. Recall that using dynamic analysis the runtime objects, as well as the actual method arguments and return values are available. As such, we are able to identify eight different `DetailAST` objects which together form the entire AST. The `DetailAST1` object appears to be the root of the tree, since it is passed to the `beginTree(DetailAST)` and `finishTree(DetailAST)` methods. In between those methods calls, the root including its subtrees is walked. When it comes to the actual method arguments, they are useful because of the following:

- Using the actual parameters of the `notifyVisit(DetailAST)` and `notifyLeave(DetailAST)` method calls it is easy to follow the walking process. We can recognize when each of the eight (sub)trees is either visited or left.
- Using the parameters of the `log(..)` method call we are able to expose the actual (abbreviated) error message that is logged after the tree is walked. If necessary, ab-



abbreviation of the arguments can be disabled in order to show the entire message.

The actual return values, on the other hand, are useful because of the following:

- Using the return values of the `getTokenName(int)` method call, we are able to identify the exact names of the actual retrieved tokens, such as a `CLASS_DEF`, `LCURLY`, and `RCURLY` token. These values provide deeper insight into the walking process.
- Using the return values of the `getParent()` method call, the parents of the eight `DetailAST` objects can be identified. Since `DetailAST1` does not have a parent (i.e., its return value is `null`), it indeed appears to be the root of the AST.

If we follow the same procedure to study the `endsWithNewline(RandomAccessFile)` method call in more detail (in this case messages from number 205 to 215 had to be selected), then it appears that this method call is quite simple (see Figure 5.14). It has a `RandomAccessFile` object as parameter and simply checks whether the last byte in a byte array equals the newline character.

In this particular case, it is not. Using the actual return value of the `length()` method call, it becomes clear that the `RandomAccessFile` object consists of 431 characters. Next, the last character is retrieved by calling `seek(430)` (since indexes start at zero). The actual return value of the `String` constructor, on the other hand, shows that the last character equals `'}'`, which indicates the end of the class. Hence, the file does not end with the platform specific newline character, and an error is logged (just like in the `finishTree(DetailAST)` method call in the control flow of `walk(DetailAST, FileContents)`). Again, the actual (abbreviated) error message is visible.

5.4.3 Implementation

Overview

From all this, we have discovered that `CHECKSTYLE` contains (at least) two different kinds of checks:

1. A check that reads tokens of the AST of the file(s) to be audited using the `walk(DetailAST, FileContents)` method of the `TreeWalker` object. This walk starts with a `beginTree(DetailAST)` method and ends with a `finishTree(DetailAST)` method, both performed on the specific `Check` object. Between those calls tokens are inspected. Logging errors takes place after the tree has been walked.

From now on, we will denote this class of checks as *AST-checks*.

2. A check that has a particular method defined in the specific `Check` object which performs the check. This method call is executed from within `process(File[])`, which is also located in this `Check` object. Instead of walking an AST (i.e., splitting a file

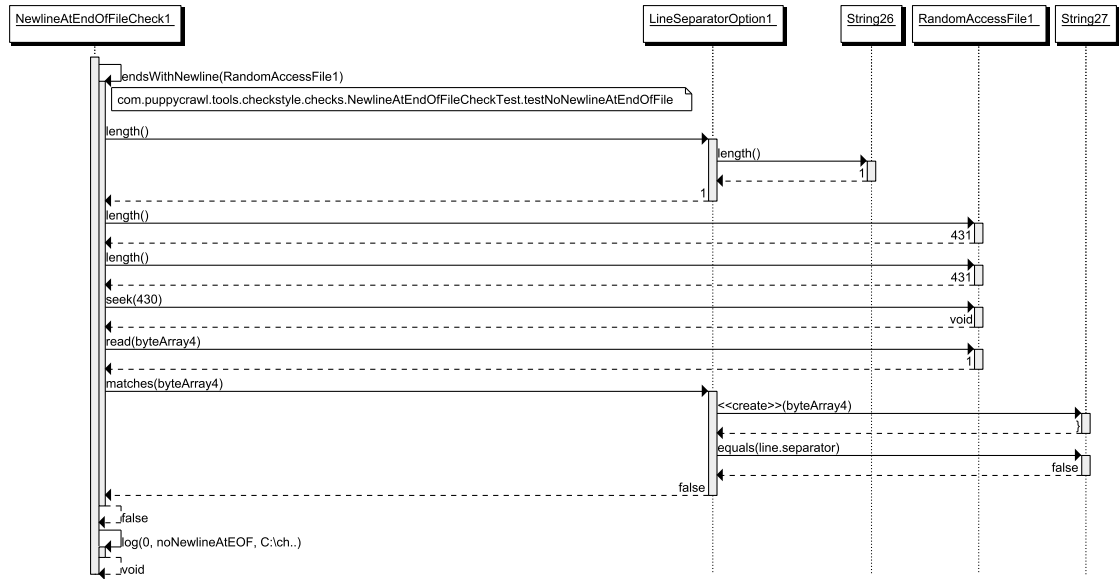


Figure 5.14: The `endsWithNewline(RandomAccessFile)` method call in more detail.

token by token) the check is directly performed on the file to be audited. Logging errors takes place within the `process(File[])` method, right after the particular check method has been executed.

From now on, we will denote this class of checks as *non-AST-checks*.

Since all five checks follow a similar pattern with the exception of the behavior of the aforementioned methods (i.e., in point 1 and 2), we expect that these methods are the ones we have to (re)define when creating a new check in `CHECKSTYLE`.

Now let us take a look at the source files of the classes on which these particular methods are performed, as can be seen in the diagrams: `PackageDeclarationCheck.java`, `DefaultComesLastCheck.java`, `UncommentedMainCheck.java`, `AvoidStarImportCheck.java`, and `NewLineAtEOFCheck.java`.

AST-checks

The four AST-checks all extend the `Check` class. Moreover, they all contain a `visitToken(DetailAST)` method which defines the action to be performed when a token is visited, as well as a `getDefaultTokens()` method which returns the token types of interest in an integer array. Two of them contain a `beginTree(DetailAST)` method which appears to do some initialization before the tree is walked, and just one of these two classes also contains a `finishTree(DetailAST)` method. Consequently, two check classes contain neither of these two methods. The `beginTree(DetailAST)` and `finishTree(DetailAST)` methods thus seem to be optional.

In order to verify this we take a peek at CHECKSTYLE's Javadoc:

- `beginTree(DetailAST)`: "Called before the starting to process a tree. Ideal place to initialise information that is to be collected whilst processing a tree."
- `finishTree(DetailAST)`: "Called after finished processing a tree. Ideal place to report on information collected whilst processing a tree."

Hence, as expected, these methods involve pre- and postprocessing which obviously is not necessary in all checks. The authors mentioning "ideal place to..." implies them to be useful though not obligatory. The `beginTree(DetailAST)` method simply initializes a single boolean value in the `PackageDeclarationCheck` class, and three variables in the `UncommentedMainCheck` class. The only `finishTree(DetailAST)` method (i.e., in the `PackageDeclarationCheck` class) is used for logging errors in case the value of the initialized boolean is false. In the other classes errors are logged within the `visitToken(DetailAST)` method.

Moreover, if we take a peek at the source code, it becomes clear that these methods exist in the `Check` class, and can be redefined in one of its subclasses. Since the methods are not abstract, and the `Check` class is not an interface, they indeed appear to be optional.

The `PackageDeclarationCheck` class is the only class that contains all aforementioned methods, which are shown in Listing 5.5.

```
/** {@inheritDoc} */
public int[] getDefaultTokens()
{
    return new int[] {TokenTypes.PACKAGE_DEF};
}

/** {@inheritDoc} */
public void beginTree(DetailAST aAST)
{
    mDefined = false;
}

/** {@inheritDoc} */
public void finishTree(DetailAST aAST)
{
    if (!mDefined) {
        log(aAST.getLineNo(), "missing.package.declaration");
    }
}

/** {@inheritDoc} */
public void visitToken(DetailAST aAST)
{
    mDefined = true;
}
```

Listing 5.5: Methods of the `PackageDeclarationCheck` class.

As can be seen in Listing 5.5, the `PackageDeclarationCheck` class is interested in the `PACKAGE_DEF` token. Prior to visiting tokens it sets a Boolean `mDefined` to false, which is an attribute that indicates whether a package declaration exists. If this token is found, then this attribute is set to true in the `visitToken(DetailAST)` method. After the tokens have been visited, an error is logged in case no package declaration has been found (in the `finishTree(DetailAST)` method).

Hence, implementing a similar check involves redefining the aforementioned methods in a new class that extends the `Check` class. We tested this by creating a new check which checks whether a class contains at least one method, and our findings appeared to be correct.

non-AST-checks

Now if we take a look at the source file of the non-AST-check, i.e., the `NewlineAtEndOfFileCheck` class, then it appears that this class extends an `AbstractFileSetCheck` class. It contains a `process(File[])` method that has a `File` array as parameter, and simply retrieves all files in a loop and performs an `endsWithNewline(RandomAccessFile)` method on each file. It returns a boolean and in case its value is false, an error is logged. The largest part of this `process(File[])` method is shown in Listing 5.6.

```

public void process(File[] aFiles)
{
    final File[] files = filter(aFiles);
    final MessageDispatcher dispatcher = getMessageDispatcher();
    for (int i = 0; i < files.length; i++) {
        final File file = files[i];
        final String path = file.getPath();
        dispatcher.fireFileStarted(path);
        RandomAccessFile randomAccessFile = null;
        try {
            randomAccessFile = new RandomAccessFile(file, "r");
            if (!endsWithNewline(randomAccessFile)) {
                log(0, "noNewlineAtEOF", path);
            }
        }
        catch (final IOException e) {
            ///CLOVER:OFF
            logIOException(e);
            ///CLOVER:ON
        }
        fireErrors(path);
        dispatcher.fireFileFinished(path);
    }
}

```

Listing 5.6: The process method of the `NewlineAtEndOfFileCheck` class.

Hence, implementing a similar check involves redefining the `process(File[])` method in a new class that extends the `AbstractFileSetCheck` class. The `getMessageDispatcher()`, `fireFileStarted(String)`, `fireErrors(String)`, and `fireFileFinished(String)` meth-

ods seem to be obligatory, since they involve firing various events, and thus have to be included in a new implementation of the `process(File[])` method. We tested this by creating a new check which checks whether the input file array does not contain more than 100 files. Again, our findings appeared to be correct.

5.4.4 Conclusion

With this experiment we have shown that the sequence diagrams generated by JRET are useful for analyzing the behavior of checks in CHECKSTYLE. First, we generated a global overview of the inner workings of five (randomly chosen) checks. Next, we studied certain method calls in more detail, making use of the actual objects, parameters and return values (which are available in dynamically obtained sequence diagrams), in order to increase program understanding.

We discovered that two different types of checks can be distinguished:

1. AST-checks, i.e., checks that examine an abstract syntax tree. Such an AST is walked, thereby retrieving and inspecting tokens. Typical checks that belong to this category: testing if a package declaration exists, testing if `default` comes after the last case in a switch statement, and testing if there does not exist a main method without comments.
2. non-AST-checks, i.e., checks that can be performed directly on the contents of the Java files to be audited. Typical checks that belong to this category: testing if a file ends with a newline character, and testing if a file contains tabs.

In order to implement a new check in CHECKSTYLE, one has to determine whether it belongs to category 1 or category 2. Next, the following steps have to be carried out:

AST-checks:

1. Create a new class that extends the `Check` class.
2. Define and implement the `getDefaultTokens()` method, which indicates the token types of interest.
3. Define and implement the `visitToken(DetailAST)` method, in which the token types of interest are retrieved using methods of the `DetailAST` class. Consequently, this method can be used to check the tokens. Also, errors can be logged using the `log(..)` function.
4. *Optional.* Define and implement the `beginTree(DetailAST)` method, in which initialization can be performed prior to walking the AST.
5. *Optional.* Define and implement the `finishTree(DetailAST)` method, which is executed after the AST is walked. It can be used to log errors using the `log(..)` function, for example.

non-AST-checks:

1. Create a new class that extends the `AbstractFileSetCheck` class.
2. Define a `process(File[])` method which has a void return type. All methods in the following steps should be defined within this `process(File[])` method.
3. Add the methods that filter the `File[]` object, and retrieve the `MessageDispatcher` object (see Listing 5.6).
4. Create a loop that retrieves all files in the file array.
5. Add the method that performs the `fireFileStarted(String)` method call prior to the actual check (see Listing 5.6).
6. Write the code that performs your check, preferably in a separate method, which should be executed on each retrieved file.
7. Add the `log(. . .)` function that logs the errors in case the check returns false.
8. Add the methods that perform the `fireErrors(String)` and the `fireFileStarted(String)` method calls (see Listing 5.6)

5.5 Discussion

In this section we discuss (the results of) our experiments in chronological order. First, we discuss the quantitative analysis in which we gathered data on the need for abstractions, and information on the effectiveness of various abstraction techniques. Next, we address the qualitative analysis in which we gained insight into CHECKSTYLE's export functionality and CHECKSTYLE-checks.

5.5.1 Quantitative analysis

Experimental setup

Before running our tool, we defined a custom tracer in order to:

1. Exclude third party software from the trace.
2. Overcome errors (such as Java heap memory overflow).

The exclusion of third party software is not necessary unless the external JAR files cause (heap memory) problems during the execution of our tool, as is the case with the `antlr` parser generator. Nevertheless, we decided to exclude them all from the trace, since they added much overhead in terms of interactions while distracting from the interactions that take place within CHECKSTYLE itself. Consequently, if we included them in the trace, more abstractions would be needed, which could remove (important) calls from the core classes as well.

Errors (such as heap memory overflows) can occur during the weaving process, and during the execution of the test cases. In the former case, it is necessary to exclude the classes that cause these errors. If the weaving phase is not completed successfully, it is not possible to run the tests, and therefore JRET will exit. In the latter case, on the other hand, exclusion is not necessary, but sometimes recommended. For example, a test case can run for a considerable time and fail eventually due to a heap space overflow. Excluding such a test would decrease execution time and, hence, increase JRET's performance.

The need for abstractions

We have shown that abstraction techniques are necessary when dynamically obtaining sequence diagrams from a JUNIT test suite: In 89% of all traces abstractions were needed in order for the resulting sequence diagrams to become human-readable. The result of this experiment, however, is influenced by:

1. The choices we have made during the experimental setup: Which classes are excluded from the trace?
2. The question when a sequence diagram is considered human-readable.

If we excluded less classes from the trace, the total number of interactions would increase. As a consequence, sequence diagrams that were originally human-readable might need abstractions after their expansion, thereby potentially increasing the aforementioned percentage.

Moreover, recall that we consider a sequence diagram to be human-readable in case it consists of 100 or less calls. If we increase this threshold, less abstractions might be necessary, whereas more abstractions might be needed in case we decrease its value. Hence, both adaptations could change the aforementioned percentage.

Another question one could ask is whether our result is applicable to software systems in general. Since CHECKSTYLE is a medium-sized open source program (276 classes divided into 21 packages) based on an extensible framework that is widely used, we think our findings are quite representative of medium-sized software systems. Moreover, CHECKSTYLE is available as an open source project at [Sourceforge.net](https://sourceforge.net), resulting in many programmers being actively involved in its implementation. The same goes for its accompanied test suite.

Hence, rather than just one person or a single company updating the code, programmers from all over the world are able to work on the project, thereby increasing its generality. In order to test whether our result is applicable to large-sized software systems, one could perform similar experiments using a more complex program such as, for example, Azureus [3].

The effectiveness of abstraction techniques

We have shown that limitation of the call depth to 1, removing internal messages, and removing getters are the most effective abstraction techniques in quantitative aspect. They yield an average reduction in calls of 95%, 90%, and 51%, respectively. Again, these results could be different in case we excluded more or less classes from the trace. For example, consider the situation where more classes are included representing a considerable number of internal messages at stack depth = 1. As a consequence, it then could be possible that removing internal messages becomes more effective than limitation of the stack depth to 1.

Like previous experiment in which we gathered data on the need for abstractions, we think our findings are quite representative of medium-sized software systems, because of the reasons already outlined.

5.5.2 Qualitative analysis

In our qualitative analysis, we investigated whether JRET can be used to become familiar with the behavior of a program in order to implement new features. With these experiments we have shown that dynamically obtained sequence diagrams provide deeper insight into the inner workings of exports and checks in CHECKSTYLE. We gained sufficient knowledge to implement both a new export functionality, as well as a new check.

Using test suites

The usefulness of our tool depends on the quality of a system's test suite. In this regard, factors of importance are, e.g., code coverage, the number of unit tests and acceptance tests, and descriptiveness of the names of test cases. The code coverage of a test suite obviously is important as it states the amount of code that can be included in the dynamic analysis. Hence, the higher the code coverage, the higher the usefulness of our tool.

Unit tests are important to study the behavior of particular features / functionalities of a target system. This is especially the case when implementing feature requests, as we illustrated during our case study (recall that CHECKSTYLE's test suite contains unit tests only). Acceptance tests, on the other hand, are important to study the behavior of a complete run of a particular target system. They provide a global overview of a real-world execution and expose interactions between different components of a system. Hence, the importance of the existence (and number) of unit and acceptance tests depends on the specific tasks a dynamic analyst has to perform.

The descriptiveness of the names of test cases is important to select the right tests for analysis. If these names are not descriptive enough, the creation of an initial selection will be more difficult, and potentially more diagrams will have to be examined. In this regard, proper documentation of a system's test suite (e.g., commented code or Javadoc) could compensate for poorly descriptive names.

Advantages

If we compare the use of JRET to static analysis (i.e., inspecting the source files), our tool has the following advantages:

- JRET offers the possibility to get a global insight into CHECKSTYLE's inner workings, whereas at the same time certain parts can be studied in the highest detail possible. Specifically, the runtime objects, as well as the actual method arguments and return values are available, which appeared to be very useful for gaining a deeper understanding of both exports and checks. As for the export feature, we were able to expose the actual Strings being printed, thereby exactly identifying the exported text. When it comes to checks, the runtime values simplified the understanding of the walking process of an AST. We could exactly follow the entire walk, exposing the actual tokens being inspected and nodes being visited. Moreover, we were able to identify the actual error message logged after the tree had been walked.
- Analyzing program behavior through a sequence diagram is more intuitive than source code analysis because of the chronological ordering. In addition, a single sequence diagram is able to capture the interactions that take place between multiple (objects of) classes. For example, the sequence diagram that depicts the `walk(DetailAST, FileContents)` method call (see Figure 5.13) shows interactions between 18 different objects. By using source code alone, one would have to switch between multiple files, which makes the task more complex and time-consuming.
- Test suites, typically written in JUNIT, provide various execution possibilities for the system at hand. Unit tests induce scenarios that effectively decompose a system's functionalities. Thus, a careful selection of test cases effectively focuses attention to a particular feature / functionality of the system to be studied. As for the export feature, we selected the `XMLLoggerTest` test cases, since they concern the already existing XML export feature. By examining the behavior of these test cases, we gained enough understanding in order to implement a new export functionality. Likewise, we made a selection of test cases that concern CHECKSTYLE-checks. Again, they provided sufficient insight into their inner workings in order to implement a new check. Looking up appropriate source files, on the other hand, could be an extensive task on itself. Also keep in mind that a single test case often involves multiple classes, which reduces the effort required to lookup and examine these classes separately, as would be the case in static analysis.

Limitations

Our approach has the following limitations:

- Unlike static analysis in which source files can be directly inspected, our tool has to be configured in order to produce sequence diagrams (see section 5.2.1). If necessary, a custom tracer should be created that excludes specific functions, classes, or packages from the trace. This appeared to be the case when tracing CHECKSTYLE, since certain

classes caused heap memory overflows, and thus had to be omitted. Creating a new tracer, however, is rather easy, since one of JRET's existing stock tracers can be copied and adapted by redefining its custom pointcuts. Moreover, if a custom tracer is not needed, a default stock tracer can be selected either using the available GUI or through command-line.

- Multiple runs may be necessary in order to gain sufficient insight into the behavior of the system-under-analysis. Such a run could take some time depending on the number of test cases and their complexity. 493 sequence diagrams were generated from CHECKSTYLE's test suite in approximately two minutes. These diagrams appeared to be sufficient for the examination of the export functionality, but when it came to checks, more runs were needed:

1. A run resulting in messages being annotated with their numbers.
2. A run that resulted in a selection of a fragment of messages.

However, rather than running the entire test suite again, we only executed the test cases of interest. As such, the resulting sequence diagrams were generated in less than 10 seconds in the successive runs.

- Since test cases mainly serve to test a particular function/class/feature (unit tests) of a program, or a combination of these (acceptance tests), a test execution in certain cases may not be appropriate for program comprehension. An example in this regard could be a suite that is merely geared towards testing of boundary values. Examining sequence diagrams that depict such boundary situations may not be sufficient for understanding how a "normal" execution works. Hence, the usefulness of our approach could be limited by the kind of test suite and its purpose.

Another factor that might change a test execution from a real-world run is the separation of test cases into different stages, including the fixture and tear down. For example, the fixture's main purpose is setting up the environment for the actual test by creating and initializing all objects. During a normal execution, on the other hand, these objects could be defined gradually rather than in a separate setup phase, or may not even be created at all if they are test case specific (such as, e.g., in the verification phase). Hence, a test case might present a somewhat distorted view of a program's dynamic behavior, which has to be taken into account.

In our case, however, the unit tests in CHECKSTYLE were adequate for the understanding of exports and checks. Also recall that Agile Development methods such as XP advocate the use of tests as a form of documentation, and thus implicitly even encourage the usage of test cases for program comprehension.

- Rather than relying merely on the dynamically obtained sequence diagrams, we sometimes verified our findings by taking a peek at the source code or Javadoc. For example, using the source code, we verified the importance of the

walk(DetailAST, FileContents) and endsWithNewline(RandomAccessFile) methods for understanding the behavior of checks in CHECKSTYLE. Since one knows beforehand which class or method to consult (as shown in the sequence diagrams), such a verification takes little effort. Consequently, although some verifications may be performed by generating and consulting more sequence diagrams, a short glance at the source code could likely be a much faster solution. We therefore encourage the combination of sequence diagrams generated by JRET with static analysis in order to verify particular results acquired during the dynamic analysis.

Validity threats

During our qualitative analysis, we relied on the names of JUNIT test cases in order to select the behavior of interest. For example, when examining the behavior of the export functionality, we selected the XMLLoggerTest test cases. When studying the inner workings of checks, on the other hand, we made a selection of test cases under the condition that their names clearly implied what the checks are supposed to do. If these names happened to be less descriptive, we probably would have to select more test cases in order to find and isolate the behavior of interest. However, from experience, we know that names of test cases usually are sufficiently descriptive. Moreover, if this is not the case, one could take a peek at (the comments in) the source code of specific tests in order to find out what they are supposed to do.

As far as our second qualitative experiment is concerned, after inspecting more source files it indeed appears that CHECKSTYLE contains two types of checks as we discovered during our case study. Although the number of AST-checks is quite larger than the number of non-AST-checks, we discovered both types because they both happened to be part of our five selected checks. Also, consider the situation where CHECKSTYLE contains more than five checks. In this particular case, our initial selection would never uncover them all.

In order to increase the chance of exposing all types of checks, a larger initial sample size could be used. Additionally, after having gained knowledge about the program's behavior using the initial selection, one may inspect appropriate source files to verify the findings. In our case, we verified the existence of two different checks. If we, however, by doing so detected the existence of more than five checks, we would have expanded the initial selection to include more diagrams for inspection. Hence, again, a combination of dynamic and static analysis appears to be an adequate solution. The latter can be used to verify results acquired by the former.

Moreover, in most cases one will probably know beforehand what kind of new checks have to be developed, such that existing checks with similar behavior can be selected as part of the sample collection to be examined, rather than picking checks randomly as we did. In that case, detection of all existing checks may not be necessary at all. Our intention of this case study was just to illustrate how JRET facilitates software maintenance tasks. As such, we have shown how JRET exposes the behavior of an unknown program in order to, for

example, implement feature requests. And in this regard JRET has proven to be effective.

Points of improvement

As previously stated, defining a custom tracer is sometimes necessary to overcome Java heap memory overflows. One might prevent such errors from occurring by updating JRET's source code. They could be related to the aspects (woven through the target system's byte-code during the weaving phase), or the remaining source code (executed when running the test cases). Either way, one could use a profiler to examine which parts of JRET's code are potentially suitable for an optimization. After optimizing the code, it might not be necessary to exclude particular classes from the trace as in a previous situation (e.g., the exclusion of CHECKSTYLE's third party parser generator). Likewise, the total execution time could be decreased as well.

Another improvement concerns the abstraction techniques implemented. Recall that pattern recognition can be performed only after other abstractions have been applied (unless it is the only abstraction technique used; see section 4.3.2). Due to time constraints we were not able to update our code to overcome this limitation. Pattern recognition is a very useful technique which could remove messages from a sequence diagram while retaining essential information. When applied in the beginning of the abstraction phase, it may reduce the need for other abstraction techniques. It would therefore be desirable to solve this shortcoming.

Currently, the pattern recognition algorithm used is only able to detect contiguous redundancies of a sequence of calls. In order to detect and replace separated recurrent patterns in sequence diagrams (e.g., the co-called *subscenarios* in [30] and [66]), one could implement another or a more advanced detection algorithm, like the one described in [21]. Such a modification could potentially increase the quality of the diagrams produced, since rather than erasing information, the recurrent pattern is pasted in a separate diagram. It then can be viewed using references which are placed in the abstracted diagram. Consequently, the comprehensibility increases while retaining all information.

JRET does not have thread support. If the target system makes use of multiple threads, our tool is not able to identify and distinguish them. Adding thread support would therefore be a nice improvement. Using AspectJ it is possible to identify which call belongs to which thread. Moreover, the SDEDIT visualizer supports the visualization of multiple threads in a single sequence diagram. Hence, JRET already provides the right facilities to include thread support, such that updating the code would not be a very complex task.

Chapter 6

Conclusions and Future Research

In this thesis, we presented a tool to help software maintainers to gain a richer understanding of a software system and its components. This is achieved by reconstructing sequence diagrams using dynamic analysis to provide deep insight into a program's dynamic behavior. We developed the tool during this research and named it JRET, which stands for Java Reverse-Engineering Tool. JRET is documented and made publicly available as an open source project through SourceForge.

Dynamic analysis involves the execution of an instrumented program to record its runtime behavior. In order to induce execution scenarios for the system to be reverse-engineered, our tool makes use of JUNIT test cases. This is because by understanding a system's test suite, one can gain a great deal of knowledge about its inner workings. Moreover, in Agile Development methods, such as eXtreme Programming, test cases are advocated as a form of documentation.

JRET was elaborated and dissected in terms of three stages, which together form the dynamic reconstruction process:

1. Creating traces
2. Applying abstractions
3. Visualizing the data

As for the first stage, we implemented bytecode instrumentation, because of the advantages over its source code counterpart and instrumentation of the runtime environment (see section 4.2).

When it comes to the second stage, we implemented various abstraction techniques to compensate for scalability issues associated with dynamic analysis and sequence diagrams. These include the following:

- The possibility to choose between object and class level.

- Filtering out system classes in the packages `java.util` and `java.lang`.
- Abbreviation of method names, return values, and arguments.
- Removing getters (with or without their control flow).
- Removing setters (with or without their control flow).
- Removing constructors and their control flow.
- Removing *Asserts*.
- Removing internal messages (with or without their control flow).
- Stack depth limitation / setting a minimum stack depth.
- Message selection.
- Pattern recognition.

One is able to apply abstractions manually, as well as automatically based on runtime metrics. These metrics are used to determine whether or not and which abstraction techniques should be performed to keep the sequence diagrams human-readable, while preserving the desired amount of detail. The employed techniques and our design choices were described and explained in section 4.3.

As far as the third stage is concerned, the (abstracted) data is visualized as sequence diagrams using two different visualizers:

- Alex Moffat's *SEQUENCE*.
- *SEDIT*: Quick Sequence Diagram Editor.

Each program has its own advantages and disadvantages. For example, whereas *SEQUENCE* supports Java 1.4, *SEDIT* offers more advanced features such as a zooming functionality and support for patterns. The entire list of advantages and disadvantages of both visualizers was given in section 4.4.

In order to evaluate our tool, we conducted a case study using *CHECKSTYLE*, which is a widely known open source program. Specifically, it concerns an extensible framework to help programmers write Java code that adheres to a coding standard. The case study involved a quantitative and a qualitative analysis in which we sought to answer the research questions that were posed in the introduction. In the former experiment, we acquired data on the need for abstractions, and information about the effectiveness of various abstraction techniques. The latter, on the other hand, was performed to illustrate *JRET*'s contribution to program comprehension. In particular, we engaged our tool to gain insight into *CHECKSTYLE*'s inner workings in order to implement two new features.

Based on our experiences with this case study, we can now formulate answers to the aforementioned research questions:

- *How urgent is the need for abstractions?*

Abstraction techniques are necessary in most cases when dynamically obtaining sequence diagrams from a JUnit test suite. Using a medium-sized program, we determined that in 89% of all traces abstractions were needed in order for the resulting sequence diagrams to become human-readable.

- *Which abstraction techniques are effective?*

Limitation of the call depth to 1, removing internal messages, and removing getters are the most effective abstraction techniques in quantitative aspect. They yield an average reduction in calls of 95%, 90%, and 51%, respectively, when using a medium-sized program.

- *Are sequence diagrams generated by JRET useful for software maintenance?*

Dynamically obtained sequence diagrams from test cases facilitate software maintenance tasks. They effectively expose a program's inner workings in order to simplify the implementation of feature requests.

Next, we outline directions for future research.

Future work

In previous chapter, we listed a number of possible improvements to increase JRET's performance, scalability, and applicability. These are, in summary:

- Optimize the source code to prevent memory overflow errors from occurring in complex situations (e.g., by using a profiler).
- Extend the possibilities of the implemented pattern detection algorithm.
- Add a pattern recognition technique that is capable of detecting separated patterns.
- Add thread support.

Optimization of the source code may be necessary in case a larger, more complex target system is used. Apart from a possible solution to memory overflows, the execution time could potentially decrease, thereby improving the performance. Moreover, a large-sized system may lead to different results compared to our case study, and will likely raise new issues. We therefore advocate to perform similar experiments in future research using a more complex program (such as, e.g., Azureus [3]).

In JRET we implemented a whole range of abstraction techniques. Their suitability in the context of testing should be further investigated by performing more experiments. In addition, we encourage the examination of new abstractions to extend the possibilities of

our tool. An example in this regard is the detection of separated patterns (i.e., replacement by subscenario boxes). Likewise, further research on a set of diverse case studies is necessary in order to assess the adequacy of new abstraction methods.

Finally, our tool may be enhanced with support for threads in future work. Examining the behavior of different threads could rise new issues such as the need for new abstractions. Recall that JRET already provides the right facilities to implement such a feature, so that its addition will not take great effort.

Bibliography

- [1] Apache Commons: The Apache Commons project, <http://commons.apache.org/>
- [2] AspectJ: The AspectJ project at Eclipse.org, <http://www.eclipse.org/aspectj/>
- [3] Azureus: The Azureus Bittorrent Client, <http://azureus.sourceforge.net/>
- [4] C. Batty. Using JUnit in Eclipse. *Eclipse Tutorial, part 10*, 2003.
- [5] K. Beck. Test Driven Development: By Example. Addison-Wesley, 2003.
- [6] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51-56, 1998.
- [7] R.S. Boyer and J.S. Moore. A Fast String Searching Algorithm. In *Communications of the ACM*, 20(10):762-772. ACM, 1977.
- [8] L.C. Briand, Y. Labiche, and J. Leduc. Towards the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. In *IEEE Transactions on Software Engineering*, 32(9):642-663. IEEE Press, 2006.
- [9] L.C. Briand, Y. Labiche, and Y. Miao. Towards the Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 57-66. IEEE Computer Society, 2003.
- [10] The C++ Programming Language. Bell Labs, <http://www.cplusplus.com/>
- [11] Checkstyle: The Checkstyle Framework, <http://checkstyle.sourceforge.net>
- [12] W. Citrin, A. Cockburn, J. von Kanel, and R. Hauser. Using Formalized Temporal Message-Flow Diagrams. *Software-Practice & Experience*, 25(12):1367-1401. John Wiley & Sons, Inc., 1995.
- [13] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing Testsuites to Aid in Software Understanding. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 213-222. IEEE Computer Society, 2007.

- [14] R. Delamare, B. Baudry, and Y. Le Traon. Reverse-engineering of UML 2.0 Sequence Diagrams from Execution Traces. <http://smallwiki.unibe.ch/woor2006/woor2006paper1/?action=MimeView>, 2006.
- [15] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object Oriented Reengineering Patterns. Morgan Kaufmann, 2002.
- [16] M. Deters and R. K. Cytron. Introduction of Program Instrumentation using Aspects. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01) Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, <http://www.cs.ubc.ca/kdvolder/Workshops/OOPSLA2001/submissions/15-deters.pdf>, 2001.
- [17] The Eclipse Test & Performance Tools Platform (TPTP) Project, <http://www.eclipse.org/tptp/>
- [18] A. Forward and T.C. Lethbridge. The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering (DocEng'02)*, pages 26-33. ACM, 2002.
- [19] P. Gestwicki and B. Jayaraman. Methodology and Architecture of JIVE. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 95-104. ACM, 2005.
- [20] T. Gschwind and J. Oberleitner. Improving Dynamic Data Analysis with Aspect-Oriented Programming. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, page 259. IEEE Computer Society, 2003.
- [21] A. Hamou-Lhadj and T.C. Lethbridge. An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls. In *Proceedings of the 1st International Workshop on Dynamic Analysis (WODA)*, ACM Press, 2003.
- [22] A. Hamou-Lhadj and T.C. Lethbridge. A Survey of Trace Exploration Tools and Techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'04)*, pages 42-55. IBM Press, 2004.
- [23] A. Hamou-Lhadj and T.C. Lethbridge. Compression Techniques to Simplify the Analysis of Large Execution Traces. In *Proceedings of the 10th International Workshop on Program Comprehension (IWPC'02)*, pages 159-168. IEEE Computer Society, 2002.
- [24] InsectJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse, <http://insectj.sourceforge.net/>
- [25] The Java Debug Interface (JDI). Sun Microsystems Inc., <http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/>

BIBLIOGRAPHY

- [26] The Java Platform Debugger Architecture (JPDA). Sun Microsystems Inc., <http://java.sun.com/javase/technologies/core/toolsapis/jpda/>
- [27] The Java Programming Language. Sun Microsystems Inc., <http://java.sun.com/>
- [28] The Java Virtual Machine Profiler Interface (JVMPI), <http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/>
- [29] The Java Virtual Machine Tool Interface (JVM TI), <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>
- [30] D.F. Jerding and S. Rugaber. Using Visualization for Architectural Localization and Extraction. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE'97)*, pages 56-65. IEEE Computer Society, 1997.
- [31] D.F. Jerding and J.T. Stasko. The Information Mural: A Technique for Displaying and Navigating Large Information Spaces. In *Proceedings of the 1995 IEEE Symposium on Information Visualization*, page 43. IEEE Computer Society, 1995.
- [32] D.F. Jerding, J.T. Stasko, and T. Ball. Visualizing Interactions in Program Executions. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 360-370. ACM, 1997.
- [33] J. Jones. Abstract Syntax Tree Implementation Idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLOP'03)*, hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf, 2003.
- [34] JRET: The Java Reverse-Engineering Tool, <http://sourceforge.net/projects/jret/>
- [35] JUnit: Unit testing framework, <http://junit.sourceforge.net>
- [36] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327-353. Springer-Verlag, 2001.
- [37] R. Kollmann and M. Gogolla. Capturing Dynamic Program Behaviour with UML Collaboration Diagrams. In *Proceedings of the 5th Conference on Software Maintenance and Reengineering (CSMR'01)*, pages 58-67. IEEE Computer Society, 2001.
- [38] R. Kollmann, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A Study on the Current State of the Art in Tool-Supported UML-based Static Reverse Engineering. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, pages 22-32. IEEE Computer Society, 2002.
- [39] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. SCED: A Tool for Dynamic Modeling of Object Systems. *Technical Report A-1996-4*, <http://www.cs.uta.fi/reports/pdf/A-1996-4.pdf>, 1996.

- [40] K. Koskimies and H. Mössenböck. Scene: Using Scenario Diagrams and Active Text for Illustrating Object Oriented Programs. In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, pages 366-375. IEEE Computer Society, 1996.
- [41] H. Leroux, A. Réquillé-Romanczuk, and C. Mingins. JACOT: A Tool to Dynamically Visualise the Execution of Concurrent Java Programs. In *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java*, pages 201-206. Computer Science Press, Inc., 2003.
- [42] M. López, A. Alfonzo, J. Pérez, J.G. González, and A. Montes. A Metamodel to Carry Out Reverse Engineering of C++ Code into UML Sequence Diagrams. In *Proceedings of the Conference on Electronics, Robotics and Automotive Mechanics*, pages 331-336. IEEE Computer Society, 2006.
- [43] R.C. Martin. UML Tutorial: Collaboration Diagrams. *Engineering Notebook Column*, 1997.
- [44] R.C. Martin. UML Tutorial: Sequence Diagrams. *Engineering Notebook Column*, 1998.
- [45] K. Mehner. JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs. *Revised Lectures on Software Visualization, International Seminar*, pages 163-175. Springer-Verlag, 2002.
- [46] M. Merdes and D. Dorsch. Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, pages 125-134. ACM, 2006.
- [47] K.W. Miller. Test Driven Development on the Cheap: Text Files and Explicit Scaffolding. *Journal of Computing Sciences in Colleges*, 20(2):181-189. Consortium for Computing Sciences in Colleges, 2004.
- [48] H.A. Müller and K. Klashinsky. Rigi - A System for Programming In-the-Large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE'88)*, pages 80-86. IEEE Computer Society, 1988.
- [49] R. Oechsle and T. Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). *Revised Lectures on Software Visualization, International Seminar*, pages 176-190. Springer-Verlag, 2001.
- [50] OMG. UML 2.0 Infrastructure Specification. Object Management Group, <http://www.omg.org/>, 2003.
- [51] OMG. XML Metadata Interchange (XMI). Object Management Group, <http://www.omg.org/technology/documents/formal/xmi/>, 2001.

BIBLIOGRAPHY

- [52] M.J. Pacione, M. Roper, and M.Wood. Comparative Evaluation of Dynamic Visualisation Tools. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 80-89. IEEE Computer Society, 2003.
- [53] W. de Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *ACM SIGPLAN Notices*, 28(10):326-337. ACM, 1993.
- [54] W. de Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the Execution of Java Programs. *Revised Lectures on Software Visualization, International Seminar*, pages 151-162. Springer-Verlag, 2002.
- [55] Rational Rose: IBM Rational Rose Family, <http://www-306.ibm.com/software/awdtools/developer/rose/>
- [56] M. Reiser. The Oberon System - User Guide and Programmer's Manual. Addison-Wesley, 1991.
- [57] T. Richner and S. Ducasse. Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. In *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, pages 34-43. IEEE Computer Society, 2002.
- [58] A. Rountev and B.H. Connell. Object Naming Analysis for Reverse-Engineered Sequence Diagrams. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 254-263. ACM, 2005.
- [59] A. Rountev, O. Volgin, and M. Reddoch. Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams. In *ACM SIGSOFT Software Engineering Notes*, 31(1):96-102. ACM, 2005.
- [60] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1998.
- [61] SDEdit: Quick Sequence Diagram Editor, <http://sourceforge.net/projects/sedit>
- [62] A. Seesing and A. Orso. InsectJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proceedings of the 2005 Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Eclipse Technology Exchange*, pages 45-49. ACM, 2005.
- [63] Sequence: Alex Moffat's Sequence, http://www.zanthan.com/itymbi/archives/cat_sequence.html
- [64] The Smalltalk Programming Language. Xerox PARC, www.smalltalk.org/
- [65] T. Souder, S. Mancoridis, and M. Salah. Form: A Framework for Creating Views of Program Executions. In *Proceedings of the 2001 IEEE International Conference on Software Maintenance (ICSM'01)*, pages 612-620. IEEE Computer Society, 2001.
- [66] T. Systä, K. Koskimies, and H.A. Müller. Shimba - An Environment for Reverse Engineering Java Software Systems. In *Software Practice & Experience*, 31(4):371-394. John Wiley & Sons, Inc., 2001.

- [67] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting Sequence Diagram from Execution Trace of Java Program. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 148-154. IEEE Computer Society, 2005.
- [68] Together: Borland Together, <http://www.borland.com/us/products/together/>
- [69] R.J. Walker, G.C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing Dynamic Software System Information through High-Level Models. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 271-283. ACM, 1998.
- [70] L. Wall. Programming Perl. 3rd Ed., O'Reilly & Associates, 2000.
- [71] J. Warmer and A. Kleppe. The Object Constraint Language. Addison-Wesley, 1999.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AOP: Aspect Oriented Programming

CASE: Computer-Aided Software Engineering

GUI: Graphical User Interface

JDI: Java Debug Interface

JRET: Java Reverse-Engineering Tool

JVM: Java Virtual Machine

LTW: Load-Time Weaving

SDR: Scenario Diagram Reconstruction

TMFD: Temporal Message-Flow Diagram

UML: Unified Modeling Language

VM: Virtual Machine