Aiding in the Comprehension of Testsuites

Bas Cornelissen
Delft University of Technology
The Netherlands
s.g.m.cornelissen@tudelft.nl

Arie van Deursen
Delft University of Technology and CWI
The Netherlands
Arie.van.Deursen@cwi.nl

Leon Moonen
Delft University of Technology and CWI
The Netherlands
Leon.Moonen@computer.org

Abstract

An integral part of test-driven software development is utilizing testcases to ensure the software's quality. However, as testsuites grow larger, they tend to grow beyond control and are no longer easily comprehended. In this position paper, we propose to employ dynamic analysis and abstractions in reconstructing scenario diagrams from such testsuites. We discuss several challenges and suggest solutions to tackle these issues.

1. Introduction

When implementing and maintaining software systems, *testing* is of vital importance to help increase the quality and correctness of code. Test-driven development [1] implies creating and maintaining an extensive testsuite in order to guarantee that the various components work correctly, both individually (by means of unit tests) and as a whole (through use of testcases).

A testing framework for Java software that is commonly used is *JUnit* [2]. JUnit allows for the specification of both unit tests and full testcases and is easy to use. A JUnit testcase consists of several steps: the creation of a fixture, exercising the method under test, comparing the results, and the teardown. It can be run as part of a complete testsuite.

Our goal is to help developers in the course of understanding and maintaining testsuites, and perhaps even discover errors or mistakes [3]. To make such testcases easy to understand, one must come up with a visualization that is both detailed and human readable. This involves *analyzing* or *tracing* the testcases, applying certain *abstractions* and, finally, *presenting* the results.

UML sequence diagrams [4] are a potentially useful means to visualize a system's behavior. A *scenario diagram* is a somewhat simplified version of a sequence diagram that is derived from a specific scenario, i.e., containing one particular control flow. Scenario diagrams provide detailed information on interactions at either the class level or the object level, and are very readable because the chronological order is intuitive. However, if no abstractions are applied, scenario diagrams tend to become too large: the complete execution of a sizeable software system would result in a scenario diagram that contains more information than the reader can handle.

In this position paper, we propose to use reconstructed scenario diagrams for the purpose of making JUnit testsuites easier to comprehend. We obtain these diagrams by tracing the execution of a testsuite.

The next section outlines the issues and design choices that we will encounter. Section 3 discusses several solutions that we are proposing, and Section 4 describes related work. Finally, we draw conclusions and indicate future directions in Section 5.

2. Challenges

In the course of converting testsuites to scenario diagrams, we face several challenges. This section addresses the most prominent issues and design choices.

Dynamic vs. static In obtaining scenario diagrams from testcases, we can choose whether to capture the system's behavior by *static* analysis (i.e., analyzing the code) or through *dynamic* analysis (i.e., tracing the execution). The benefits of a static approach are the genericity and compactness, whereas a dynamic technique offers more detailed information on important aspects such as *late binding*. This is illustrated in Figure 1: this example of *dynamic dispatch* would not have been very readable in a static context because of the lack of object identities therein. A well-known drawback of scenario diagram reconstruction using dynamic analysis is that one needs specific scenarios and that the diagrams thus represent only part of a whole system's behavior; however, since testcases *are* basically scenarios we feel that, in this particular context, more accurate information outweighs genericity.

Test stage separation The second issue that arises is how to trace the *various phases* of the execution of a testcase,

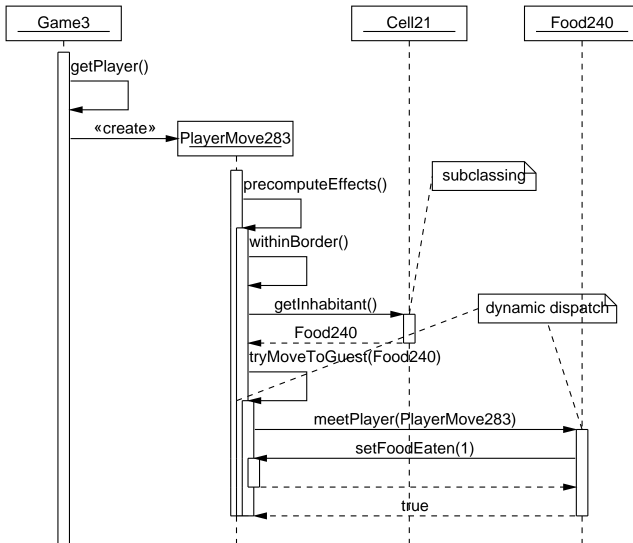

Figure 1. Dynamically reconstructed scenario diagram featuring an example of dynamic dispatch.

i.e., how we can distinguish between the test initialization, the test execution and the result validation. Traditional tools and techniques generally do not deal with this testcase-specific problem. It is an important issue, as the initialization and validation phases are potentially less relevant and should therefore be treated separately (i.e., be put in a separate diagram or even skipped).

Class vs. object level Another design choice concerns whether we want to trace the interactions on the *class* level or on the *object* level. The former is easier, whereas the latter provides more detailed information that is especially suitable for display in scenario diagrams.

Scalability However, despite the fact that in general the execution of a unit testcase is relatively short, *scalability* problems are inevitable. Most simple unit tests will probably fit within a single scenario diagram, whereas more complex testcases induce too many interactions to simply put in a diagram without applying any form of abstraction. Therefore, we will need abstractions that are both efficient and *useful*, i.e., we must determine which interactions are presumably irrelevant and can consequently be omitted. One way would be for the tool to suggest several abstractions, while ensuring the viewer remains in control of the level of detail.

3. Techniques

We have come up with several design choices and solutions to the issues discussed earlier. We are in the process of implementing these in a prototype tool that we are using for analyzing a range of existing test suites.

3.1. Tracing testcases

There exist several methods to obtain traces from software systems, among which the most commonly used are manually instrumenting code (e.g., [5]), using a debugger or profiler, and instrumentation through *aspects* [6]. The shortcomings of each of these techniques are mostly well known and are not discussed here. We feel that using aspects in our framework is the most flexible solution in our context, since it enables us to specify very accurately which parts of the execution are to be considered, where tracing must start and stop, and which steps need be taken afterwards.

Aspects can trace the execution of a testcase and produce detailed information on all interactions, such as the unique objects that are involved, the current thread, and the (runtime) arguments in case of method and constructor calls. Being able to distinguish between objects has certain advantages, as it provides detailed information on object interactions and exposes occurrences of polymorphism and late binding. Figure 1 shows an example[1].

In addition, aspects allow for the precise definition of which objects and interactions are to be traced, and enable us to make a distinction between the various stages in a testcase. Among other things, this distinction offers us the opportunity to filter the assertions in the comparison stages, in case the viewer considers them unnecessary. Moreover, we have a means to create *separate* scenario diagrams of the various phases for the viewer to browse through.

3.2. Abstractions

In order to make large scenario diagrams easier to read, we need several types of abstractions to reduce the amount of information. In the context of scenario diagrams, one intuitively thinks of omitting objects and classes and hiding interactions to shrink the dimensions of the diagram. But which messages and objects can be omitted while maintaining the general idea of the testcase?

One technique that we propose is to limit the *stack depth* of the execution. By use of a maximum stack depth, we can hide all interactions above a certain threshold, thus omitting messages and (potentially) the objects involved. Intuitively, this filters low level messages that tend to be too detailed, at least for an initial viewing. This is illustrated by Figures 2 and 3: the former diagram depicts the testcase in full detail, whereas the latter shows only the essence. A similar

[1] The scenario diagrams in this research were created using UMLGraph [7].

Figure 2. Reconstructed scenario diagram at a low abstraction level.

abstraction was applied in [8], in which (in a static context) the length of a call chain is considered.

Another abstraction method is to *hide constructors* if there are too many. This is especially applicable in the initialization stages of complex testcases. Additionally, it would be even more useful to hide *irrelevant* constructors and the associated objects, i.e., to filter them in case they are never used later on. That way, we will presumably have reduced the dimensions without loss of important information.

As was briefly mentioned in Section 2, we feel that the viewer must ultimately decide which abstractions are to be applied and which are not. We therefore plan to conclude the processing of all testcases with *recommended* scenario diagram specifications: based on several testcase statistics such as the amount of objects, constructors, and stack depth frequencies, it is automatically determined which of the abstractions are highly advised. Such specifications must be subject to adjustments at the viewer's discretion.

Figure 3. Reconstructed scenario diagram at a high abstraction level (with a maximum stack depth of 2).

4. Related work

There is a great deal of ongoing research regarding the visualization of dynamic information. Among this research is a comparison of dynamic visualization techniques by Pacione et al. [9] and of trace summarization tools in particular by Hamou-Lhadj et al. [10].

On the subject of testing, an interesting research was conducted by Gälli et al. [11], in which the authors focus on ordering (unit) tests based on the number of methods that are called by these tests.

With respect to reverse engineering UML scenario diagrams, several attempts have been made in the past. They include both static and dynamic approaches and, as was discussed in section 2, each of these techniques has its advantages and drawbacks.

Various approaches reconstruct scenario and interaction diagrams based on static analysis of program code [8, 12, 13, 14]. The techniques that are used vary from mapping of control flow graphs [13] to interprocedural dataflow analysis [8]. A comparison of various approaches is presented in [14].

There have been various reports of dynamic approaches in the literature as well, of which some are discussed by Briand et al. [15]. In the same paper they present a strategy that is aimed at capturing the objects, the messages that are exchanged, conditions, and repetitions when executing a scenario. This is accomplished through manual intrumentation of the source code. The paper does not describe how the scenarios are obtained, i.e., how they are distilled from a use case or filtered from the reconstructed scenario diagram.

Systä et al. [16] aid in the understanding of Java systems in an environment called Shimba, which uses both static analysis and dynamic information. They reason at the level of classes and obtain the required information from a system's bytecode and by using a customized debugger. Their focus is primarily on maintaining consistent views on both

structural (static) and behavioral (dynamic) aspects of the system.

Riva et al. [17] combine static and dynamic analysis to reconstruct message scenario charts. In their trace-based approach, they provide an abstraction mechanism based on the decomposition hierarchy that is extracted from the system's source code. It is not described how the scenarios are defined, and in dealing with large diagrams, they only offer manual abstraction techniques.

5. Conclusions

We have proposed to employ dynamic analysis and scenario diagrams with the goal of aiding in the comprehension of testsuites. We have discussed the issues and design choices that we will encounter and, through several examples, elaborated on our choices for these techniques. Dynamic analysis is the most logical choice for us as it provides the most details; and scenario diagrams, as long as they are not too large, are an excellent means of showing this detailed behavior. By means of certain metrics, a set of recommended abstractions is determined that aims towards presenting a scenario diagram (for each testcase stage) that is both readable and contains the desired amount of detail.

Our next step is to implement these techniques in a framework and to examine whether we can obtain meaningful results for both interested viewers and experienced developers. To this end, we are planning to perform at least two case studies.

JPACMAN is a simple game consisting of 25 Java classes and is mainly used for educational purposes. Though being a small system, it is complicated enough to give us an indication as to the the usefulness of our approach, since JPACMAN involves polymorphic methods and large traces. It also has a testsuite comprising over 50 testcases.

Another case that we are currently investigating is *CroMod*, an industrial Java system featuring both simple unit tests and complex testcases. By means of extensive feedback from the developers, we want to discover which abstractions are generally required and hope to improve our techniques.

References

[1] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[3] J.A. Jones and M.J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05)*, pages 273–282, 2005.

[4] OMG. UML 2.0 infrastructure specification. Object Management Group, http://www.omg.org/, 2003.

[5] InsectJ: A generic instrumentation framework for collecting dynamic information within Eclipse, http://insectj.sourceforge.net/.

[6] AspectJ: The AspectJ project at Eclipse.org, http://www.eclipse.org/aspectj/.

[7] D. Spinellis. On the declarative specification of models. *IEEE Software*, 20(2):94–96, march/april 2003.

[8] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineerd sequence diagrams. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 254–263, 2005.

[9] 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, 2003.

[10] 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, 2004.

[11] M. Gälli, M. Lanza, O. Nierstrasz, and R. Wuyts. Ordering broken unit tests for focused debugging. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, pages 114–123, 2004.

[12] 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, 2001.

[13] A. Rountev, O. Volgin, and M. Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. In *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)*, pages 96–102, 2005.

[14] 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, 2002.

[15] 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, 2003.

[16] T. Systä, K. Koskimies, and H. Müller. Shimba - an environment for reverse engineering Java software systems. *Software - Practice and Experience*, 31(4):371–394, 2001.

[17] C. Riva and J. V. Rodriguez. Combining static and dynamic views for architecture reconstruction. In *Proc. 6th Conf. on Software Maintenance and Reengineering (CSMR'02)*, pages 47–55, 2002.