Composing Unit Tests*

Markus Gälli, Orla Greevy, and Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
{gaelli,greevy,oscar}@iam.unibe.ch

Abstract. [1] If we were to apply the testing techniques of object-oriented systems prescribed by the XUnit framework to a car factory, the result would be an inefficient process: A tire would be created, quality assured and then thrown away, only to be recreated later to test the functionality of the whole car.
XUnit makes it difficult to reuse intermediate results of low level unit tests. As a consequence a higher level unit test is forced to recreate test scenarios which were already created by lower level unit tests. This duplicated testing effort is time-consuming both for setting up new scenarios and for running the tests. To address this problem we suggest a semi-automatic approach to compose tests. First we describe how we can detect candidates of composable test cases by partially ordering their sets of covered method signatures, then we present techniques to refactor unit tests accordingly.
Keywords: Unit testing, factories, XUnit, composition

1 Introduction

A software product line is defined as a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.
XUnit [BG98] in its various forms is a widely-used open-source unit testing framework. It has been ported to most object-oriented programming languages and is integrated in many common IDEs such as Eclipse.
We claim that units under test can not only be single methods or classes but also whole software components of a software product line. By allowing the unit test to deliver the tested core asset as a return value, we can reuse the tested core asset in assembling a test for a composed asset in order to facilitate scenario creation and to reduce testing time.
The XUnit framework does not allow low level unit tests themselves to be composed into higher level unit tests - whereas low level functionality is composed out of lower level functionality.

* We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "A Unified Approach to Composition and Extensibility" (SNF Project No. 200020-105091/1, Oct. 2004 - Sept. 2006)

[1] 2nd International Workshop on Software Product Line Testing pages 16–22, Technical Report ALR-2005-017

This makes the set up of test scenarios an unnecessary tedious task and leads to unnecessary long testing times. Our hypothesis is that a majority of unit tests can be refactored into composed test cases.
We will explain our approach with an illustrating example of a simplified university administration system, which consists of the following four XUnit test cases:

- PersonTest≫testNew tests if the roles of a person are defined.
- PersonTest≫testName tests if the name of a person was assigned correctly.
- UniversityTest≫testAddPerson tests if the university knows a person after the person has been added to it.
- PersonTest≫testBecomeProfessorIn tests if some person, after having been added as a professor, also has this role.

In Figure 1 one can see, that all methods called in UniversityTest≫testAddPerson are also called in PersonTest≫testBecomeProfesssorIn, but that neither the methods called in test case PersonTest≫testNew nor in test case PersonTest≫testName are also called completely in any other test case.

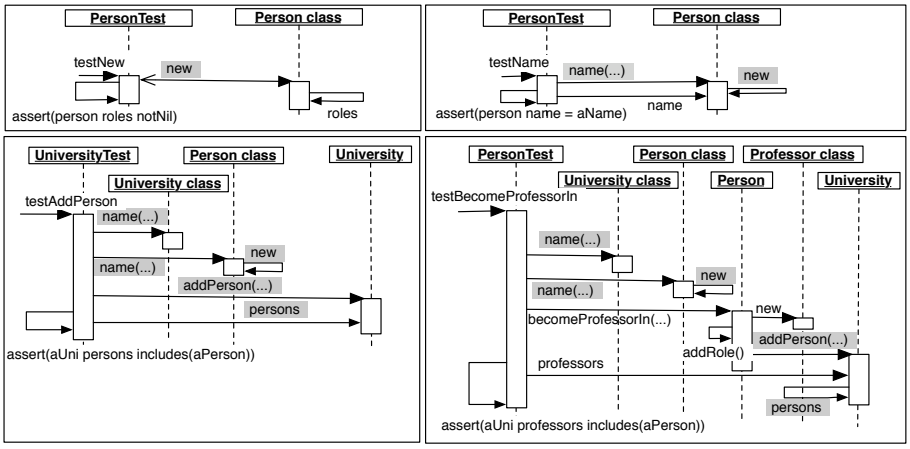

Fig. 1. The test for #becomeProfessorIn: covers the test for #addPerson:. The test for #new overlaps with the test for #name. Intersecting signatures are displayed gray.

2 Approach

We first present a technique to identify comparable test cases of existing test suites by sorting their sets of covered method signatures and then introduce two refactorings to tune these comparable tests.

2.1 Identifying Redundant Test Cases with Coverage Sets

We say that unit test A *partially covers* unit test B ($A \supseteq B$), if only up to *tolerance* method signatures covered by B are not included in the set of method signatures covered by A (1). We say that unit test A *overlaps* B ($A \equiv B$), if A *partially covers* B and B *partially covers* A (2). We say that two unit tests A and B are *comparable* if at least either one *partially covers* the other.

$$A \supseteq B \Leftrightarrow |Signatures(B) \setminus Signatures(A)| <= tolerance | tolerance \in \mathbb{N} \quad (1)$$

$$A \equiv B \Leftrightarrow A \supseteq B \wedge B \supseteq A \quad (2)$$

.
Based on this partial order we developed the following algorithm to identify comparable test cases, which are candidates for refactoring:
First we instrument the code and obtain traces of method calls that are invoked during the execution of the tests. Then we extract and store the set of method signatures of each test into an InstrumentedTestCase object. We sort all this InstrumentedTestCase objects according to the cardinality of the sets starting with the smallest. For each InstrumentedTestCase we detect the first covered test, that is both bigger than it and includes its method signatures tolerating *tolerance* methods not to be included in the bigger one. If we find one, we move the partially covered one into the covering one. If we find that both partially cover each other, we merge these two tests, building an equivalence relation between them. For our example using a *tolerance* 2 of we end up with a partial order of our tests depicted in Figure 2.

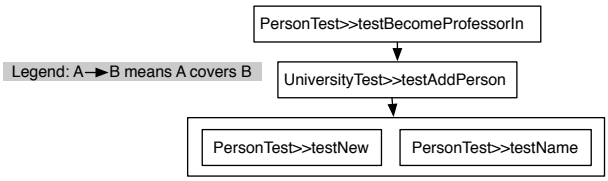

Fig. 2. A sample test hierarchy based on coverage sets.

2.2 Refactoring comparable test cases

Having identified comparable tests forming a partial order, we can refactor them: We start again with the smallest test case and try to include it into its next smallest comparable test case with either of the following two refactorings:

– *Abstract assertions:* Move the assertion from the test into a post condition of the method under test. In our toy example the assertions are already abstract enough to work directly as post conditions of the method under test. We thus could move the includes assertion of the UniversityTest≫testAddPerson into a post condition of University≫addPerson itself. Otherwise one can try to convert the concrete assertion of the unit test into an abstract assertion serving then as the post condition.
– *Publish Test Result:* If an object created by a low level test can be immediately used as parameter or receiver for the method under test of a higher level test, we can directly call the low level test from our higher level test, eliminating the need to run the low level test standalone and to recreate the scenario. This is certainly only possible, if the test framework allows us to let the tests return objects. In JUnit Version 4.0 all tests have to be void, thus this kind of easy test composition would not be possible there.

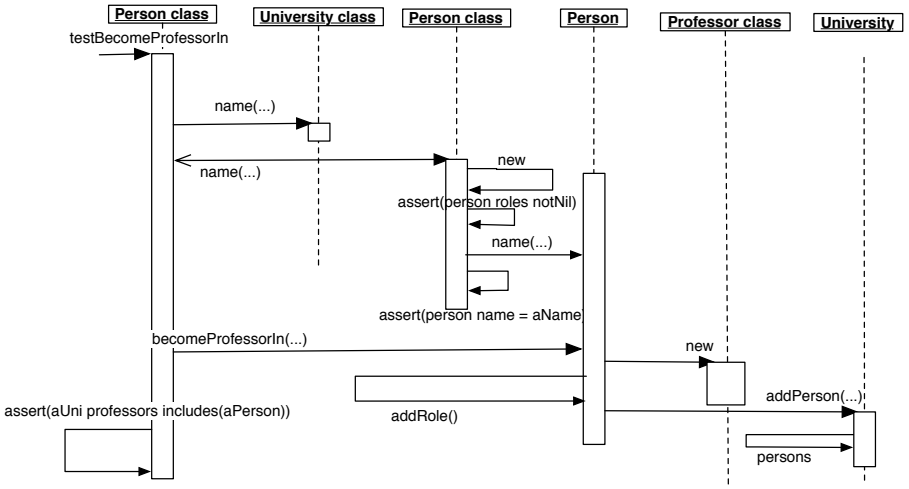

Fig. 3. The tests refactored. Only one test has to be run now, instead of four, as it captures all former subtests in post-conditions.

3 Status and Future Work

Having analyzed [GLNW04] the partial order of the unit tests of CodeCrawler, a code visualization tool, [Lan03] using a *tolerance*=0, the resulting graph looked like seen in Figure 4. Observing the three right subgraphs formed by the test covering relationship one can see that a

surprising high number of these tests are comparable, but we have not yet refactored them.

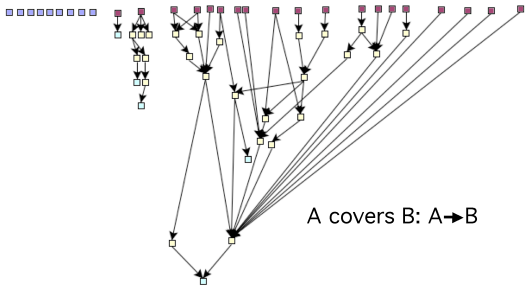

Fig. 4. The coverage hierarchy of the Code Crawler tests visualized with Code Crawler.

In [GLNW04] we automatically detected several covering relationships between unit tests like in the following interesting two examples (the former ist always covering the later):

- LoaderTest≫testConvertXMIToCDIF
  (LoaderTest≫testLoadXMI)
- SystemHistoryTest≫testAddVersionNamedCollection
  (SystemHistoryTest≫testAddVersionNamed)

A possible refactoring suggested by the first covering relationship is to test the loader of some XMI data structure and, letting the test giving back the XMI structure, reusing the structure to convert it to another structure called CDIF in the covering test.
The second covering relationship indicates an n-to-1 relationship between addVersionNamedCollection and addVersionNamed, where reusing the result of addVersionNamed or changing the concrete assertion of addVersionNamed into some post condition can lead to a composed unit test.
We plan to refactor a big case study after having analysed it with our approach and show that we can efficiently reduce testing time and provide reusable tests. With this case study we want to answer the following questions:

- How much can we speed up the execution of the test suite?
- Can we decide automatically between several refactorings?
- Can we use our partial ordering of coverage to give us a hint, if a refactoring was successful and all former tests are still run?
- How can we prioritize our tests in an efficient way so that atoms are only run within their calling tests but not stand alone?

4 Related Work

In previous work we showed that failing unit tests are presented in a random order, whereas they could be presented in a meaningful order using

the partial order of covered method signatures. [GLNW04] We also defined a taxonomy of unit tests [GLN05], where we manually categorized more than 1000 unit tests of Squeak, an open source object oriented development system. Our results from this large case study show that most unit tests are either atoms, which we call one-method tests, or composable out of these one-method-tests. In [GND04] we suggested a Smalltalk browser where one can integrate tests with the methods under test and where tests are stored as factory methods on the class side of the returned object. Liebermann and Hewitt also tightly integrate testing and programming in [LH80] and reuse tests.
McGregor [McG01] suggested a way to compose partial tests along variation points.
Edwards also underlined the importance of examples [Edw04].
Test case prioritization [RUCH99] has been successfully used in the past to increase the likelihood that failures will occur early in test runs. The tests are prioritized using different criteria, the criterion which most closely matched our approach was *total function coverage* [EMR00]. Here a program is instrumented, and, for any test case, the number of functions in the program that were exercised by this test case is determined. The test cases are then prioritized according to the total number of functions they cover by sorting them in order of total function coverage achieved, starting with the highest.

5 Conclusion

We have presented a partial order using sets of covered method signatures to detect comparable test cases. We have introduced two refactorings which can be applied to some of these comparable test cases in order to reduce testing time and increase reuse of test scenarios. We have given first evidence that relevant portions of test cases do partially cover each other, and that results obtained by the partial order are semantically meaningful. We have not yet applied our approach to a big case study.

References

[BG98] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[Edw04] Jonathan Edwards. Example centric programming. In *OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 124–124. ACM Press, 2004.

[EMR00] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis*, pages 102–112. ACM Press, 2000.

[GLN05] Markus Gälli, Michele Lanza, and Oscar Nierstrasz. Towards a Taxonomy of SUnit Tests. In *Proceedings of ESUG Research Track 2005*, September 2005. To appear.

[GLNW04] Markus Gälli, Michele Lanza, Oscar Nierstrasz, and Roel Wuyts. Ordering broken unit tests for focused debugging. In *20th International Conference on Software Maintenance (ICSM 2004)*, pages 114–123, 2004.

[GND04] Markus Gälli, Oscar Nierstrasz, and Stéphane Ducasse. One-method commands: Linking methods and their tests, October 2004. OOPSLA Workshop on Revival of Dynamic Languages.

[Lan03] Michele Lanza. Codecrawler — lessons learned in building a software visualization tool. In *Proceedings of CSMR 2003*, pages 409–418. IEEE Press, 2003.

[LH80] Henry Lieberman and Carl Hewitt. A session with tinker: Interleaving program testing with program writing. In *LISP Conference*, pages 80–99, 1980.

[McG01] John D. McGregor. Testing a software product line. Technical report, Carnegie Mellon University, 2001.

[RUCH99] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings ICSM 1999*, pages 179–188, September 1999.