Combining Formal Specifications with Test Driven Development*

Hubert Baumeister

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany
baumeist@informatik.uni-muenchen.de

Abstract. In the context of test driven development, tests specify the behavior of a program before the code that implements it, is actually written. In addition, they are used as main source of documentation in XP projects, together with the program code. However, tests alone describe the properties of a program only in terms of examples and thus are not sufficient to completely describe the behavior of a program. In contrast, formal specifications allow to generalize these example properties to more general properties, which leads to a more complete description of the behavior of a program. Specifications add another main artifact to XP in addition to the already existent ones, i.e. code and tests. The interaction between these three artifacts further improves the quality of both software and documentation. The goal of this paper is to show that it is possible, with appropriate tool support, to combine formal specifications with test driven development without loosing the agility of test driven development.

1 Introduction

Extreme Programming advocates test driven development where tests are used to specify the behavior of a program before the program code is actually written. Together with using the simplest design possible and intention revealing program code, tests are additionally used as a documentation of the program. However, tests are not sufficient to completely define the behavior of a program because they are only able to test properties of a program by example and do not allow to state general properties. The latter can be achieved using formal specifications, e.g. using Meyer's design by contract [21].

As an example we consider the function `primes`, that computes for a given natural number n a list containing all prime numbers up to and including n. Tests can only be written for special arguments of the primes function, e.g. that `primes(2)` should produce the list with the number 2 as its only element, and that `primes(1553)` is supposed to yield the list of prime numbers from 2 up to 1533. Actually, a program that behaves correctly w.r.t. these tests could have the set of prime numbers hard coded for these particular inputs and return arbitrary lists for all other arguments. One solution is to move from tests to specifications, which allow to generalize the tested properties. For example, the behavior of `primes` would be expressed by a formal specification stating that the result of the function `primes(n)` contains exactly the prime numbers from 2 up to n, for all natural numbers n.

* This research has been partially sponsored by the EC 5th Framework project AGILE: Architectures for Mobility (IST-2001-32747)

C. Zannier et al. (Eds.): XP/Agile Universe 2004, LNCS 3134, pp. 1–12, 2004.
© Springer-Verlag Berlin Heidelberg 2004

This example shows that formal specifications provide a more complete view on the behavior of programs than tests alone. However, while it is easy to run tests to check that a program complies with the tests, the task of showing that a program satisfies a given specification is in general more complex. To at least validate a program w.r.t. a specification, one can use the specification to generate run-time assertions and use these to check that the program behaves correctly.

The study of formal methods for program specification and verification has a long history. Hoare and Floyd pioneered the development of formal methods in the 1960s by introducing the Hoare calculus for proving program correctness as well as the notions of pre-/postconditions, invariants, and assertions [13, 10]. Their ideas were gradually developed into fully fledged formal methods geared towards industrial software engineering, e.g. the Vienna Development Method (VDM) developed at IBM [17], Z [23], the Java Modeling Language (JML) [19] and, more recently, the Object Constraint Language (OCL) [25] – which again originated at IBM – used to specify contraints on objects in UML diagrams. For an overview of formal methods and their applications refer to the WWW virtual library on formal methods [5].

An important use of formal specifications is the documentation of program behavior without making reference to an implementation. This is often needed for frameworks and libraries, where the source code is not available in most cases and the behavior is only informally described. In general, the documentation provided by a formal specification is both more precise and more concise compared to the implementation code because the implementation only describes the algorithm used by a method and not what it achieves. Not only the literature on formals methods, but also in the literature on the pragmatics of programming, e.g. [15, 20], recommends to make explicit the assumptions on the code using specifications because this improves the software quality.

The goal of this paper is to show that it is possible, with appropriate tool support, to combine formal specifications with test driven development without loosing the agility of the latter. This is done by using the tests, that drive the development of the code, also to drive the development of the formal specification. By generating runtime assertions from the specification it is possible to check for inconsistencies between code, specifications, and tests. Each of the three artifacts improves the quality of the other two, yielding better code quality and better program documentation in the form of a validated formal specification of the program.

Our method is exemplified by using the primes example with Java as the programming language, JUnit[1] as the testing framework, and the Java Modeling Language (JML) [19] for the formulation of class invariants and pre- and postconditions for methods. We use JML since JML specifications are easily understood by programmers, and because it comes with a runtime assertion checker, [6], which allows to check invariants and pre- and postconditions of methods at runtime.

2 Formal Specifications and Tests

As with test driven development, in our proposed methodology, tests are written before the code. Either now or after several iterations of test and code development, the prop-

[1] www.junit.org