Combining Formal Specifications with Test Driven Development* **

Hubert Baumeister

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, D-80538 München, Germany
E-mail: 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

* This research has been partially sponsored by the EC 5th Framework project AGILE: Architectures for Mobility (IST-2001-32747).
** to appear in Proc. XPAU 2004, August 15–18, Calgary, Canada.

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 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 properties that underly the tests are generalized into formal JML-specifications. We then generate assertions from these specifications using the JML runtime assertion checker. The invariants and pre- and postconditions are finally validated during test runs. Any inconsistency between code, tests, and formal specification will result in an exception. This leads to additional confidence in the code, the tests, and the specification. Making the specification underlying a set of tests explicit may reveal that some tests are still missing. On the other hand, an exception thrown by the assertion checker is the result of an error either in the code or in the specification. The method we propose has 5 steps:

1. Write the test
2. Implement the code
3. Refactor the code
4. Generalize the tests to a specification, and
5. Refactor the specification

Each of the steps is performed as needed, therefore not all the steps need to appear in each iteration of our method.

2.1 Example

We continue the primes examples, introduced in Section 1, with Java as implementation language and the JUnit test framework. The goal is to implement a static member function `primes(int n)` in class `Primes` that returns a list containing an integer object if, and only if it is a prime number in the range from 2 up to and including n. The sequence of tests used in this paper follows closely that of Beck and Newkirk [3], which uses the `primes` function as an example for refactoring.

Step 1: Write the test. A first test for the primes function is to assert that `primes(0)` returns the empty list.

[1] www.junit.org

```
public void testZero() {
   List primes = Primes.primes(0);
   assertTrue(primes.isEmpty());
}
```

Step 2: Implement the code. The obvious implementation just returns the empty list.

```
public static List primes(int n) {
   return new LinkedList();
}
```

Step 3: Refactor the code. Since the above code does not suggest any refactorings, we omit the code refactoring step.

Step 4: Generalize the tests to a specification. The following JML specification states that if n is 0 then the result should be the empty list.

```
public behavior
 requires n == 0;
 ensures \result.isEmpty();
```

The precondition is `n == 0` and is given by the `requires` clause, and the postcondition is `\result.isEmpty()` and is given by the `ensures` clause. The keyword `\result` represents the result of the method. The keywords `public behavior` indicate that the following is a public specification for the method `primes(int n)`. Note that the precondition `n == 0` makes it obvious that we are not yet done with specifying and implementing the primes method as we want our method to work also with other inputs than 0.

Using the JML assertion generator, assertions for the pre- and postconditions of the JML specification are generated and integrated in the class file of class Primes. Now the tests are run again and a JML exception is thrown if a precondition or a postcondition is violated.

Step 5: Refactor the specification. In the next step we generalize the precondition to $n \leq 1$.

```
public behavior
 requires n <= 1;
 ensures \result.isEmpty();
```

This generalization step shows that a test is missing, i.e., `primes(1).isEmpty()`. However, we choose not to add a new test because this new test would not fail and thus does not force us to change existing code [2].

This finishes the first iteration of our method. Since we are not done with the implementation of the primes method, we proceed with the next iteration of writing tests, code, and specifications.

Step 1: Write the test. The next test case tests that `primes(2)` returns the list that contains as its only element an integer object with value 2.

```
public void testTwo() {
   List primes = Primes.primes(2);
   assertEquals(1, primes.size());
   assertTrue(primes.contains(new Integer(2)));
}
```

Step 2: Implement the code. The following implementation validates this test:

```
public static List primes(int n) {
   List primes = new LinkedList();
   if (n == 2) primes.add(new Integer(2));
   return primes;
}
```

Step 4: Generalize the tests to a specification. The corresponding specification looks as follows.

```
public behavior
   requires n <= 1;
   ensures \result.isEmpty();
also
   requires n == 2;
   ensures \result.size() == 1 &&
           \result.contains(new Integer(2));
```

We use the `also` keyword to add a new pre- and postcondition specification to an existent one. In this case, either `n <= 1` is true, then `\result.isEmpty()` has to be true, or `n == 2` is true and then

```
\result.size() == 1 && \result.contains(newInteger(2))
```

has to hold. In the above case, both preconditions are disjoint; however, if both preconditions are satisfied, then both postconditions also have to hold. As in the first iteration, running the tests with generated assertions for the JML specification yields no error.

Step 1: Write the test. After having dealt with the simple cases, we now deal with more complex situations: we write a test that ensures that all the prime numbers from 2 to 1000 are contained in the result of `primes(1000)`.

```
public void testLots1() {
   int n = 1000;
   List primes = Primes.primes(n);
   for (int i = 1; i <= n; i++) {
      if (isPrime(i))
         assertTrue(primes.contains(new Integer(i)));
   }
}
```

The boolean function `isPrime(int i)` is an auxiliary function that returns true if the argument is a prime number and false otherwise. It is given by the following specification, which directly reflects the definition of prime numbers.

```
public behavior
ensures
 \result <==> (n > 1 &&  (\forall int i; i > 1 && i < n;  n % i != 0));
```

The expression `\forall var-decl; range-pred; pred;` asserts that for all values of the variables occurring in `var-decl` which satisfy `range-pred`, the predicate `pred` has to hold. The range predicate `range-pred` and the predicate `pred` may contain boolean expressions which in turn may contain Java methods of type boolean which do not modify the state (called *pure* methods in JML). Logical equivalence is written `<==>`.

An implementation satisfying the specification of `isPrime` is the following (validated using the method presented in this paper):

```
public static boolean isPrime(int n) {
   if (n < 2) return false;
   for (int i = 2; i <= n/2; i++) {
      if (n % i == 0) return false;
   }
   return true;
};
```

Step 2: Implement the code. The simplest implementation that passes `testLots1` just returns a list with integer objects representing the integers from 2 to n.

```
public static List primes(int n) {
   List primes = new LinkedList();
   for (int i = 2; i <= n; i++) primes.add(new Integer(i));
   return primes;
}
```

Step 4: Generalize the tests to a specification. Instead of writing a specification for a fixed number (in our case 1000), we directly express the desired property for arbitrary integers n.

```
public behavior
   requires n <= 1;
   ensures \result.isEmpty();
also
   requires n == 2;
   ensures \result.size() == 1 &&
           \result.contains(new Integer(2));
also
   requires n > 2;
   ensures  (\forall int i; i >= 2 && i <= n;
               isPrime(i) ==>
                   \result.contains(new Integer(i)));
```

In the above, the symbol `==>` denotes logical implication.

Looking at the specification we see that the implementation of the primes function is not yet complete. We have checked that all prime numbers occur in

the result of `primes(n)`, but not that each number in the result of `primes(n)` is a prime number. Therefore, we need an additional test.

Step 1: Write the test.

```
public void testLots2() {
   int n = 1000;
   List primes = Primes.primes(n);
   for (Iterator e = primes.iterator(); e.hasNext();) {
      int prime = ((Integer) e.next()).intValue();
      assertTrue(isPrime(prime));
      assertTrue(prime <= n);
   }
}
```

Step 2: Implement the code. This test forces us to implement a more sophisticated primes function. In our example, we use the sieve of Eratosthenes to compute prime numbers. The idea is to remove all numbers k in the list from 2 to n which are dividable by some number occurring before k. The following is a possible implementation:

```
public static List primes(int n) {
   List primes = new LinkedList();
   for (int i = 2; i <= n; i++)
      primes.add(new Integer(i));
   for (int i = 0; i < primes.size(); i++) {
      int prime = ((Integer) primes.get(i)).intValue();
      for (int j = i + 1; j < primes.size(); j++) {
         int value = ((Integer) primes.get(j)).intValue();
         if (value % prime == 0)
            primes.remove(j);
      }
   }
   return primes;
}
```

Step 4: Generalize the tests to a specification. Again, the corresponding part of the specification allows to abstract from the number 1000 to an arbitrary integer n. Accordingly, our specification expresses that each element in the result is a prime number.

```
public behavior
   requires n <= 1;
   ensures \result.isEmpty();
also
   requires n == 2;
   ensures \result.size() == 1 &&
         \result.contains(new Integer(2));
also
   requires n > 2;
```

```
   ensures (\forall int i; i >= 2 && i <= n;
            isPrime(i) ==>
               \result.contains(new Integer(i)));
also
   requires n > 2;
   ensures (\forall Integer i; \result.contains(i);
                    isPrime(i.intValue()));
```

After generating the assertions into the Primes class-file and running all the tests, we see that no pre-/postcondition pair is violated.

Step 5: Refactor the specification. In contrast to Beck [1], who argues that tests should not be refactored, we want to refactor specifications because we want to use specifications also as program documentation. The result of the refactoring yields a more concise specification as several pre- and postcondition pairs can be eliminated: We can delete all those pairs that are logical consequences of other, remaining pre/post pairs.

```
public behavior
   ensures
      (\forall int i; i >= 2 && i <= n;
          isPrime(i) ==>
          \result.contains(new Integer(i)));
      &&
      (\forall Integer i; \result.contains(i);
         isPrime(i.intValue()));
```

By running the JUnit tests instrumented with the corresponding run-time assertions generated from the JML specifications, we can be certain that we have not produced a specification that conflicts with the code. However, we have oversimplified the specification. This is not detectable by the tests. In this case the specification does not ensures that `primes(0).isEmpty()` holds, as the list containing new Integer(2) is in compliance with the specification. Considering this case reveals a missing assertion in the specification and the tests: `primes(n)` may contain prime numbers greater than n. It is questionable if we should modify the tests as with the above implementation the tests would not reveal a failure. Modifying the tests would therefore be in violation of the principle that test should only be written if they fail first [2]. On the other hand, it would make sense to include this condition in `testLot2` to document this condition. In any case, the condition needs to be added to the JML specification.

```
public behavior
   ensures
      (\forall int i; i >= 2 && i <= n;
         isPrime(i) ==>
         \result.contains(new Integer(i)))
      &&
      (\forall Integer i; \result.contains(i);
        isPrime(i.intValue()) && i.intValue() <= n);
```

The result of the presented process is a set of tests, code, and a specification which ensure that the code implements the desired behavior, which is documented by the specification. We can now use this specification of `primes` to describe its behavior without making reference to all the test cases and/or the code. Note that the resulting primes specification is both much shorter and much easier to understand than the tests and the code alone.

2.2 Advantages of Using Specifications

The advantage of specifications is that they provide an additional view on the software which complements the test and implementation view. While the test view describes the properties of a software in terms of examples, specifications distill specific examples into more general properties. The description of the behavior of a program using a formal specification is more abstract than the tests and the implementation (which, in addition, is not always available) and more precise than an informal textual description. A precise description of the behavior of a program which is given independently from its implementation is important, e.g., in the documentation of libraries or frameworks.

When the behavior of a program is given formally, that is, in computer understandable form, it is easy to derive properties of programs from the specification or to use the specification to generate black-box tests that can be used by a quality assurance team either automatically or at least semi-automatically. In contrast to tests written by the developer during test driven development, the generated tests are not biased by the programmer who has written the code. Furthermore, we can also use these specifications as input to tools which allow to verify that the code implements the specifications is actually correct, as we will demonstrate below.

One problem with test driven development is that it is possible to write non-tested code. This risk is minimized by the XP practices. With pair programming, four eyes are looking at the code, and a rule of thumb with test driven development is that each line of the production code has to be justified by tests. Our method poses a similar problem: it may happen that a specification, consisting of class invariants, pre- and postconditions, is not strong enough to logically imply the tests. That is, a specification might not express everything that is covered by the tests. The programmers who design the tests and the specification therefore have to make sure that the tests are actually implied by the specification. This is usually done by instantiating the abstract specification to concrete examples. E.g. the test that `primes(2)` returns the list with 2 as its only element can be obtained by instantiating n with 2 in our last specification of the primes method. On the other hand, the specification may be too strong, that is, it could impose stronger conditions than the tests. This case usually leads to failed assertions, i.e. a violated pre-/postconditions or invariant, in the present or a later iteration.

2.3 Validation vs Verification

The basic idea of the presented method is to annotate code with assertions generated from a specification. During test runs, inconsistencies between tests, specification, and code are detected, in which case an exception is thrown. Note, that this method is only able to increase the confidence in the correctness of the code and the specification, but does not guarantee that the code satisfies the specification. As with tests, this method helps finding bugs but does not prove the absence of errors. Still, it leads to more complete specifications and more correct code compared to just separating the process of writing the specification and of implementing the code. In addition, our method can be accompanied with other tools, for example ESC/Java [9] for extended static type checking using a sublanguage of JML, and Krakatoa [7] and the LOOP tool [24] for verifying that the implementation meets its specification (cf. [16] for a more complete overview of available tools for Java). Of course the effort to prove code correct with these tools is considerably higher than the effort of validating the specification.

3 Conclusion

The method in this paper describes a practical way of combining formal specifications with test driven development which is geared towards XP. There are already several approaches (cf. [12, 8, 11]) combining XP and design by contract. These approaches try to replace tests by formal specifications by considering tests as special kinds of specifications. The problem with these approaches is that they need some means to either prove the code correct with respect to the specification (which requires a considerable effort), or to generate test cases from the specification. In our method, the test cases are designed in the usual way within test driven development. This accounts for the observation that it is easier to start with concrete examples and scenarios first and then generalize the examples into specifications in a second step. In addition, we get a third view, the specification view, on the software that complements the implementation and the test view. Our method hence improves the quality of all three views. A similar line of reasoning to the one presented here has been independently developed by Ostroff et al. in the context of Eiffel [22].

For applications where security is relevant the specification view helps, on the one hand, to develop more complete test suites than one usually gets with test driven development. For example, one can generate tests from the specification and the code (white-box and black-box tests), e.g. [4]. This is because the test generation strives for a complete set of tests while the goal of tests in test driven development is to drive the process of writing the program code. On the other hand, the specification view is a prerequisite for proving programs correct. This has been done, for example, in the context of smart cards using the JavaCard API and JML, e.g. [14].

The presented method was used in the EU-project AGILE[2] to develop a multi-user dungeon (MUD) game played by several players using their mobile

[2] Architectures for Mobility; www.pst.ifi.lmu.de/projekte/agile.

phones. Players can interact with each other when they are in the same virtual room. They can, for example, trade objects, fight, or talk. Writing the specification revealed bugs in the code that were not detected by just using the tests alone and also helped to find new tests because the specification provides a more abstract view on the methods to be implemented. Vice versa, the use of tests showed that often the first attempt on writing a specification fails, usually because some specific cases are omitted.

The presentation of our method in this paper uses Java as the programming language and the Java Modeling Language (JML) as the specification language. However, the method is not restricted to the use of JML, Java, or even design by contract. In the MUD game, for example, the Hugo model-checker [18] was used in addition to JML to verify liveness and safety properties, e.g., that the protocol for trading objects among players is deadlock free and that both players agree on the outcome of a trade (i.e. successful or not successful).

Note that it does not always warrant the effort to maintain a specification view on the code. One has to balance the quality of the software with the work of maintaining the specification view. In situations where a concise and precise documentation of the behavior of a program independent from the code is needed, or where an improved software quality is needed, e.g. in applications where security is critical, the gain is worth the effort.

Acknowledgments. I would like to thank Hakan Erdogmus, Alexander Knapp, Dirk Pattinson, and the anonymous referees for helpful comments on earlier versions of this paper.

References

1. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
2. K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.
3. K. Beck and J. Newkirk. Baby steps, safely. article.PDF at groups.yahoo.com/group/testdrivendevelopment/files, February 2002.
4. R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
5. J. Bowen. The World Wide Web virtual library: Formal methods. www.afm.sbu.ac.uk, 2004.
6. Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H. R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice (SERP'02)*, pages 322–328. CSREA Press, Las Vegas, 2002.
7. E. Contejean, J. Duprat, J.-C. Filiâtre, C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program verification. Available at krakatoa.lri.fr, October 2002.
8. Y. A. Feldman. Extreme design by contract. In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 2003*, volume 2675 of *LNCS*, pages 261–270. Springer, 2003.
9. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI), volume 37, pages 234–245. ACM, 2002.
10. R. W. Floyd. Toward interactive design of correct programs. In C. V. Freiman, J. E. Griffith, and J. L. Rosenfeld, editors, *Information Processing 71, Proceedings of IFIP Congress 71, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28*, pages 7–10. North-Holland, 1972.
11. H. Heinecke and C. Noack. Integrating extreme programming and contracts. In K. Beck, M. Marchesi, and G. Succi, editors, *2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering, XP 2001, May 20–23, 2001, Villasimius, Sardinia, Italy*, pages 24–27, 2001.
12. A. Herranz and J. J. Moreno-Navarro. Formal extreme (and extremely formal) programming. In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 2003*, volume 2675 of *LNCS*, pages 88–98. Springer, 2003.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
14. E. Hubbers, M. Oostdijk, and E. Poll. Implementing a formally verifiable security protocol in Java Card. In *Proc. of SPC'2003, 1st International Conference on Security in Pervasive Computing, Boppard, Germany, March 12-14, 2003*, 2003.
15. A. Hunt and D. Thomas. *The Pragmatic Programmer*. Addison–Wesley, 2000.
16. B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In F. S. d. Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects, 1st International Symposium, FMCO 2002, The Netherlands, Novermber 5–8, 2002, Revised Lectures*, volume 2852 of *LNCS*, pages 202–219, 2003.
17. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall international series in computer science. Prentice Hall, New York, 2nd edition, 1990.
18. A. Knapp, S. Merz, and C. Rauh. Model checking timed UML state machines and collaborations. In W. Damm and E. R. Olderog, editors, *Proc. 7th Inernational Symposium Formal Techniques in Real-Time and Fault Tolerant Systems*, volume 2469 of *LNCS*, pages 395–416. Springer, Berlin, 2002.
19. G. T. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.
20. S. McConnell. *Code Complete*. Microsoft Press, 1993.
21. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, New Jersey, 1997.
22. J. Ostroff, D. Makalsky, and R. Paige. Agile specification-driven development. In *Extreme Programming and Agile Processes in Software Engineering, 5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 2004*, volume 3092 of *LNCS*. Springer, 2004.
23. J. M. Spivey. *The Z Notation: A Reference Manual*. International series in computer science. Prentice Hall, New York, 2nd edition, 1992.
24. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 299–312. Springer, Berlin, 2001.
25. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1st edition, 1998.