Agile Specification-Driven Development

Jonathan S. Ostroff[1], David Makalsky[1], and Richard F. Paige[2]

[1] Department of Computer Science, York University, Canada.
{jonathan, dm}@cs.yorku.ca
[2] Department of Computer Science, University of York, UK
paige@cs.york.ac.uk

Abstract. We present an agile approach to Specification-Driven Development, which combines features of Test-Driven Development and the plan-based approach of Design-by-Contract. We argue that both tests and contracts are different types of specifications, and both are useful and complementary for building high quality software. We conclude that it is useful for being able to switch between writing tests and writing contracts, and explain how Specification-Driven Development supports this capability.

1 Introduction

Traditional software development methods stress the elicitation and documentation of a "complete" set of requirements, followed by architectural and high-level design, coding, inspection and testing. This general approach is sometimes described as *plan-driven development*. Agile methods were a reaction to these traditional "documentation driven, heavyweight software development processes" [2], focusing on an iterative design process with rapid feedback in which code appears early [15].

In this paper, we describe an integrated approach, Specification-Driven Development (SDD), which combines the best features of the agile Test-Driven Development (TDD) methodology with the best features of the plan-driven approach of quality-first Design-by-Contract (DbC) [11]. The emphasis in TDD is the production of *executable tests* that act as restricted emergent specifications of collaborative behaviour. DbC emphasises a concept of *contract*, which can be represented using constructs such as preconditions, postconditions, and class invariants for explicitly specifying expected behaviour. At first glance, TDD and DbC conflict, or, as one authority put it:

> If it's a matter of gut feeling, then mine is that the two approaches, test first and Design by Contract, are the absolute extreme opposites with no combination possible or desirable. It's nice once in a while to see a real irreconcilable opposition [13].

We attempt to show that not only are TDD and DbC compatible, but that each can enhance the other. In SDD, both unit tests and contracts are specifications, and there are advantages to using each type of specification in producing reliable systems. TDD is superior for capturing complex emergent behaviour (e.g., trace behaviour) that cannot easily be expressed statically with contracts; DbC is superior for completely specifying behaviour. The two approaches are compatible: both TDD and DbC are iterative and are based on the view that it is important to produce working code as soon as possible.

We make our arguments in the context of the Eiffel language which has DbC built-in. But, DbC works in other languages such as Java as well [7].

2 Plan-Driven Development

The conventional, systematic plan-driven approach to software development is inherited from systems engineering. Plan-driven development approaches, such as DbC, stress the elicitation and documentation of a complete set of requirements, followed by architectural and high-level design. Code and tests often appear at the tail end of the process. The gap between requirements and code is thus bridged by *specifications*, which describe constraints on behaviour shared between the physical world and the system. Iterations between writing specifications and coding are often encouraged. Incremental approaches to plan-driven development have been adopted, but all still emphasise documentation and traceability between requirements, specification, and code.

Plans can be written in a variety of ways, including structured natural language, UML class and sequence diagrams, and formal methods. There is an associated cost with applying mathematical techniques; in general, it is much more than testing with the benefit of obtaining higher quality [3]. The economic reality is that for most software development, testing and inspections trump formal specifications.

In plan-driven approaches, complete documentation brings with it two main problems. First, there is the problem of keeping the documentation consistent with changes in the design and code. And second, there is the sheer volume of documentation that must be produced. Analysts must document the requirements, designers must create the design specifications, and programmers must document their code. At each stage, additional detail must be added as we do not know who will be reading the documentation; it may therefore be safer to err on the side of caution.

2.1 Design by Contract

DbC is a form of plan-driven development that naturally lends itself to agile development because of the way in which its documentation is expressed. It also has almost all the benefits of mathematical methods – and these are formidable for emphasising software quality first – without the associated cost. Contracts on software are written using preconditions, postconditions and class invariants, providing mathematical specifications. These contracts are written in the assertion language of the programming language itself, and are therefore *executable*; contracts are thus a form of the best kind of documentation, that which executes with the code, and which is always guaranteed to be consistent with the code (otherwise an assertion violation would arise at run-time).

Suppose we need to calculate the square root of a real number to four decimal places. Fig. 1 provides an example illustrating how this might be done using contracts.

There are many benefits to using contracts to document software: contracts are checked every time the code is executed (and violations are immediately flagged); components are self-documenting because the contracts are part the documentation (and

```
class MATH feature
  square_root(x: DOUBLE): DOUBLE is
    require x>=0
    do  -- your algorithm goes here, e.g., Newton's method
    ensure
      (Result*Result - x).abs <= epsilon;
      epsilon = old epsilon
    end
  epsilon: DOUBLE  -- accuracy
invariant
  0 < epsilon and epsilon <= 0.001
end -- MATH
```

Fig. 1. Example of a contract for class $MATH$

inconsistency between code and contracts is impossible). And The benefits of using contracts to document software are as follows: contracts provide design rules for maintaining and modifying the behaviour of components, cf., behavioural subtyping, and a basis for formal verification.

In [11] Meyer describes the "quality-first" DbC design method. Meyer implements quality-first DbC using Eiffel and the BON visual modelling language, both of which support contracts, and for which integrated tool support exists. A brief summary of quality-first DbC in BON/Eiffel follows.

1. Write Eiffel code or produce BON diagrams as soon as possible, because then supporting tools immediately do syntax, type, and consistency checking.
2. Get the current unit of functionality working before starting the next. Deal with abnormal cases, e.g., violated preconditions, right away.
3. Intertwine analysis, design, and implementation.
4. Always have a working system.
5. Get cosmetics and style right.

DbC can be seen as an instance of plan-driven development, but unlike some approaches it does not suffer from the "big design up front" problem, in part because the plans in DbC are validated code. There are two vague steps in the quality-first DbC approach: (a) in step (2) we must get the current unit of functionality working, but how do we progress from informal requirements to a contract or a BON diagram? (b) in step (4) we are told to constantly compile, execute, and test the system but how the testing is to be performed is not explained. These two problems can at least partially be alleviated with the use of TDD techniques.

3 Test-Driven Development

Test Driven Development (TDD) is one of the popular evolving agile methods [1]; it emphasises testing first as a replacement for up-front design. Like all agile methods, TDD stresses the development of working code over documentation, models and plans. The TDD cycle proceeds as follows: (1) write the test first (without worrying if it does not compile); (2) write enough code to make the test pass; (3) *refactor* the code to eliminate redundancies and other design flaws introduced by making the test pass.

A striking aspects of this approach is the idea that the code that is implemented may not be behaviorally correct: it just has to pass the test. Correctness means passing all the tests. The test is therefore the specification. Another striking aspect is refactoring as a replacement for up-front design (sometimes pejoratively called a "big up front design") [5]. Testing, with tool support, occurs all the time: before and after refactoring, and whenever new functionality is implemented.

Tests are a form of specification, typically (though not exclusively) dealing with normal and expected behaviour. Tests do not provide precise documentation of class interfaces. Thus, they are useful in capturing traces of valid behaviour for scenarios of the system, but may miss the big picture, i.e., the architecture and component views. Thus tests cannot be described as complete requirements. Tests encompass both unit and regression tests, and also what we call *collaborative tests*. These latter tests are related to UML sequence and collaboration diagrams in that they show the messages (method calls) sent between a number of specific objects within a use case. A good example of a collaborative test is shown below, in Fig. 2, for a simple banking system. An account is initialised and withdrawal is made, with the expected result of the account checked for correctness.

```
test_teller_withdrawal_request: BOOLEAN is
  local a: ACCOUNT; t:TELLER_TRANSACTION
  do
    -- initial balance $900 in John's account
    create a.make("John Doe",900)
    check a.balance=900 end
    create t

    -- test scenario
    t.request(a,500)
    t.withdrawal_request
    result := a.balance=400 and t.succeeded
  end
```

Fig. 2. Collaborative test for banking system

The benefits of TDD are many. For one, the cost of verification is spread across the development process. The TDD process also provides low-level information about test failures, on the operation or even statement level, thus making debugging easier. Experience has shown that designs driven by tests tend to exhibit high cohesion and loose coupling, perhaps possibly due to the frequent refactoring and the requirement to keep the design as simple as possible. TDD also allows predictive specification of what code will do, independent of the existence of the code itself. Finally, the tests produced using TDD provide documentation of the design and the design process. The latter, in particular, will be essential for any requisite auditing and review.

The limitations of TDD come in part from the incompleteness of tests: requirements cannot be completely captured by tests in general without enumerating all scenarios. Further, tests cannot deal with phenomena that are in the environment of the system, whereas contracts can express constraints on such constructs.

3.1 Collaborative vs. Contractual Specifications

Test-based unit and collaborative specifications are incomplete, because they consider only specific scenarios. Consider the following unit test, written in Eiffel.

```
test_integers_sorted:BOOLEAN is
    local sa1,sa2: SORTABLE_ARRAY[INTEGER]
    do
       sa1 :=  <<4, 1, 3>>; sa2 := <<1,3,4>>;
       sa1.sort;
       Result := equal(sa1, sa2)
    end
```

in which we create an unsorted array `sa1`, execute routine `sort`, and then assert that the array is `equal` to the expected sorted array `sa2`. The unit test does three things for us. The test is a precise specification of a unit of functionality (the sort function in the special case of array `<<4, 1, 3 >>`). The test also drives the design. It induces the public interface of class *SORTABLE_ARRAY* with features such as `sort`. However,

- The unit test specifies that array `<<4, 1, 3>>` must be sorted. But what about tests for all the other (possibly infinite) arrays of integers?
- The unit test does not test arrays of `REAL`, or arrays of `PERSON` (say by age). After all, the class `SORTABLE_ARRAY[G]` has a generic parameter `G`.
- It is hard to describe preconditions with unit tests. For example, we might want the sort routine to work only in case there is at least one non-void element in the array. (We could make the sort routine have no precondition, but that would then force us to always program defensively [10, p344].)

By contrast, the contractual specification in Fig. 3 is a precise and detailed specification of the sorted array. The quantifiers can be expressed using Eiffel's agent notation.

The generic parameter `G` of class `SORTABLE_ARRAY` is constrained to inherit from `COMPARABLE`. This allows us to compare any two elements in the array, e.g., the expresion `item(i) <= item(i+1)` is legal whether the array holds instances of integers or poeple, provided the instances are from classes that inherit from `COMPARABLE`.

Routine `sort` is specified via preconditions and postconditions. The preconditions state that there must be at least one non-void element to sort. The unit test did not specify this, nor is it generally easy for unit tests to specify preconditions. The postcondition states that the array must be sorted and is unchanged. This postcondition specifies this property for all possible arrays, holding elements of any type. Again, only an infinite number of unit tests could capture this property.

```
class      SORTABLE_ARRAY [G − > COMPARABLE]  inherit ARRAY[G]
feature      sort is
          require
               count_positive: count > 0
               elements_not_void: ∀ i | lower ≤ i ≤ upper • item(i) ≠ Void
          do
               ...
          ensure
               sorted: ∀ i | lower ≤ i ≤ upper • item(i) ≤ item(i + 1)
               count_unchanged: count = old count
          end
end
```

Fig. 3. Class SORTABLE_ARRAY

Since contracts and tests are both specifications (the contract being more general), they can both serve to drive development of the code.

Unit tests can be used to automatically check that the code satisfies its specification – just run the tests. Can code be checked against the contracts? One approach would be program verification which provides strong assurance but requires qualitatively more time and effort than testing. The simpler approach is to turn assertion checking on in the programming language. But, unit tests will now be required to execute the code so that contracts can be checked. However, there is a *test amplification* effect, which we discuss in the next section.

While contractual specifications are detailed and complete, they have disadvantages. Consider a class $STACK[G]$ with routines given by $push(x : G)$ and pop. While contracts can fully specify the effects of $push$ and pop individually, they cannot directly describe the last-in-first-out (LIFO) property of stacks which asserts that

$$\forall\, s : STACK, x : G \bullet pop(push(x, s)) = s$$

By contrast, the LIFO behaviour can easily be captured using test-based collaborative specifications.

4 Specification-Driven Development

Clearly there are benefits to plan-driven development based on DbC, and test-driven development. Choosing between the value offered by the approaches will equally clearly depend on the project at hand. There are surprising commonalities between TDD and DbC, particularly: both contracts and tests are specifications; both TDD and DbC seek to transform requirements to compilable constructs as soon as possible; both TDD and DbC are lightweight verification methods; both methods are incremental; and both emphasise quality first in terms of units of functionality. We claim that it is not necessary to choose between the two approaches *a priori*, and that there are substantial benefits to using TDD and DbC together in a project.

Specification-Driven Development (SDD) provides the ability to use TDD and DbC techniques in the same development. It assumes (a) the availability of a contract-aware programming language (e.g., Eiffel, or Java with a suitable pre-processor), and (b) a suitable testing framework (e.g., JUnit or ETester). The statechart of Fig. 4 describes the approach. It does not dictate where to start – it is the developer's choice whether to start with TDD or DbC based on project context. However, the emphasis is always on transforming customer requirements into compilable and executable code.

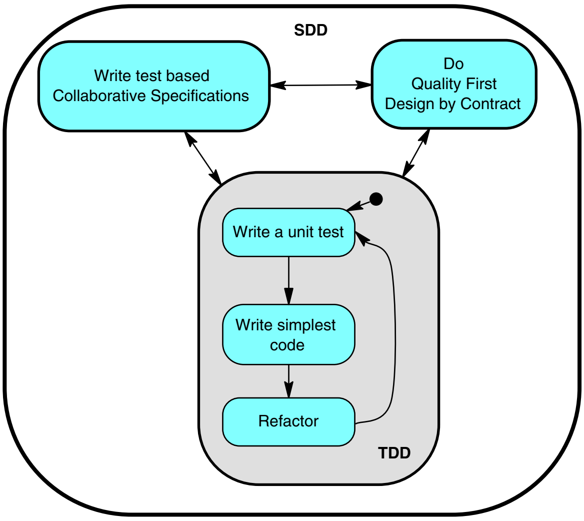
SDD
Write test based Collaborative Specifications
Do Quality First Design by Contract
Write a unit test
Write simplest code
Refactor
TDD

Fig. 4. SDD: Specification-Driven Development

SDD provides more than TDD or DbC individually, as it eliminates some of the limitations with each approach. But SDD is more than the sum of TDD and DbC, as there are synergies between the approaches. In particular, contracts act as *test amplifiers*. When writing a contract, it is easy to make mistakes, or write a contract that is simply too weak and which underconstrains the system. Some of these flaws will be caught by executing the system; but this is not sufficient in general. Writing tests to exercise the contracts (i.e., which validate and invalidate each pre- and postcondition) can help validate the tests, and can also help drive the production of tests.

4.1 Some Observations

SDD can start with writing tests (as illustrated by the left-most state in the statechart), or with writing contracts. However, there are two reasons to prefer writing unit tests before contracts:

Closure: A unit test provides a precise description of a unit of functionality and hence also a clear stopping point – you write just enough clean code to get the test to pass. Contracts do not provide clear stopping points for units of functionality in quite the same way, thus allowing for the possibility of unnecessary design.

Collaborative specification friendly: tests can formalize instances of collaborative specifications more easily than contracts, as illustrated by the last-in-first-out property of stacks.

Contracts, of course, can provide precise documentation of the complete behaviour of code in a way that tests cannot (as illustrated in Fig. 3). Contracts also provide preconditions; tests cannot document or check for preconditions. Finally, contracts can supply a qualitative level of assurance for code beyond that of testing in the case of program verification, and can act as an automatic test amplifier in the case that assertion checking is turned on.

In summary:

1. Contracts are good for fleshing out the design while making underlying assumptions explicit.
2. Contracts spell out the logical assumptions underlying a design more completely and concisely than unit tests.
3. Tests are good for writing collaborative specifications; as such, they are likely to be more appropriate early in the development process when scenarios are being refined to executable constructs. Contracts are good for constraining the design to meet the requirements.

Table 1. SDD synergies – $SDD > max(TDD, DbC)$

TDD lacks:	**Quality First DbC has:**
Good Design Documentation	✓ Self-Documenting Design (automated using seamless and reversible BON)
Detailed interface specifications of normal and abnormal behaviours	✓ Contracts and contractual specifications
TDD has:	**Quality First DbC lacks:**
✓ Collaborative specifications	Units of functionality for Quality First
✓ Automated Tests (JUnit/ETester)	Systematic regression tools for exercising contracts
Synergies: ✓ Contracts are test amplifiers ✓ Contractual and collaborative specifications provide lightweight verification of the design	

5 Conclusions

We have investigated the compatibility and complementarity of TDD and DbC, in producing a new agile approach called Specification-Driven Development. Our conclusion is that TDD and DbC are complementary technologies and can be used synergistically, but also to supplement limitations: contracts make design decisions explicit that may only be implicit in tests; and tests can better capture requirements (such as the LIFO property on stacks) than contracts.

We are providing tool support for the Eiffel language that allows TDD and DbC to be used together. This support comes via the ETester framework, documented elsewhere

[8]. ETester is specifically designed to make it easy to write unit tests and tests involving contracts. Additional work on an Eiffel plug-in for Eclipse will also make use of ETester.

Our work has similarities to that of Feldman [4]; his work focused particularly on the relationship between contracts and refactoring, whereas we have focused on the assistance that contracts provide to the TDD process. Feldman in particular makes the point that using contracts can reduce the amount of tests that need to be written because contracts cover the correctness of methods. We disagree on this point as tests must still be written to exercise the contracts, and to particularly deal with contracts that underspecify behaviour. However, we do agree with Feldman's findings that contracts work synergistically with refactoring.

Table 1 summarises our conclusions.

References

1. Beck, K. *Test-driven Development: by example*, Addison-Wesley, 2003.
2. Beck, K., A. Cockburn, R. Jeffries, and J. Highsmith. Agile Manifesto www.agilemanifesto.org/history.html. 2001.
3. Berry, D.M. Formal methods: the very idea — Some thoughts about why they work when they work. *Science of Computer Programming*, 42(1): p11–27, 2002.
4. Feldman, Y. Extreme Design by Contract. In *Proc. XP 2003*, LNCS, Springer-Verlag, 2003.
5. Fowler, M. and K. Beck. *Refactoring*, Addison-Wesley, 1999.
6. Gamma, E. and K. Beck. JUnit: A cook's tour. Java Report, p27-38, 1999.
7. Leavens, G.T., K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, ACM, 2000.
8. Makalsky, D. ETester Unit Testing Framework. Available at www.cs.yorku.ca/eiffel/etester, 2004.
9. Martin, R.C. *Agile software development*, Pearson Education, 2003.
10. Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1997.
11. Meyer, B. Practice to Perfect: the Quality-First Model. *IEEE Computer* 30(5), 1997.
12. Meyer, B. Towards practical proofs of class correctness. In *Proc. ZB 2003*, Springer-Verlag, LNCS 2651, p359-387, 2003.
13. Meyer, B. Personal communication, June 2003.
14. Paige, R. and J.S. Ostroff. The Single Model Principle. *Journal of Object Oriented Technology*, 1(5): 2002.
15. Williams, L. and A. Cockburn. Agile Software Development: It's about Feedback. *Computer*, 36(6): p39-43, 2003.