Specification-Driven Design with Eiffel and Agents for Teaching Lightweight Formal Methods

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

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

Abstract. We report on our experiences in teaching lightweight formal methods with Eiffel. In particular, we discuss how we introduce formal methods via Eiffel's design-by-contract and *agent* technologies, and how we integrate these techniques with test-driven development, in an approach called *specification-driven design*. This approach demonstrates how formal methods techniques fit with industrial software engineering practice.

1 Introduction

For a number of years we have been teaching formal methods (FMs) in a variety of different ways, including traditional program verification (via weakest preconditions as well as refinement calculi), to Computer Science and Engineering undergraduates with a range of backgrounds. Our focus in the past few years, though, has been on teaching formal methods as an integrated part of software engineering. Specifically, we want to teach FMs in such a way so that students – especially those who are maths-phobic – do not get the impression that they are an eccentric, specialised technology. Instead, we want students to obtain the view that they are part of the software engineer's toolkit, and are supported by mainstream, industrially applicable tools.

The approach that we have taken, developed over a number of years of experiment and careful consideration, is based upon the use of Eiffel [11]. Eiffel is an important and substantial language for teaching formal methods, for a number of reasons:

- It is both a specification and programming language and as such is compatible with the ideas of refinement [15].
- It is one of the best object-oriented languages available, in terms of its generality and expressiveness, its support for building reliable systems, static type checking, reusable libraries, and ease of use.
- It has industrially proven tools that support checking of specifications against code, while also providing important features such as incremental compilation, debugging, and GUI construction.

The first point is a critical one for teaching FM. It is not well-known that Eiffel can be easily used as a formal specification language, and that tools can be used to check the correctness of Eiffel specifications. The specification elements of the language go as far as supporting quantifiers in boolean expressions, via Eiffel's agent technology (discussed in the sequel). The ability to use Eiffel both as a specification and programming language allows broader us of the language over one or more courses that emphasise different elements, such as GUI design, real-time and embedded systems programming, etc.

There are of course substantial challenges to teaching Eiffel as a lightweight formal method. In particular, students are sometimes hesitant to learn Eiffel, because it is not perceived as a mainstream OO language – i.e., they want to learn Java or C++. As well, students sometimes think that Eiffel tools are inferior to tools for, e.g., Java, and that Eiffel libraries are less comprehensive and expressive than similar ones for C++ and Java. The second point is one that can best be addressed by experience with Eiffel tools and libraries; after some experience the students tend to change their assessment in favour of Eiffel. The first point can be addressed by teaching Eiffel in a software engineering context, and by emphasising that a programming language is not specifically being taught. To this end, we introduce Eiffel and its FM techniques in a very specific way, which we discuss shortly.

In this paper, we describe our experiences and approach to teaching lightweight FM with Eiffel. We start with an overview of Eiffel, dwelling on its support for formal specification, and describe its agent technology briefly. We then describe three important definitions: specifications, requirements, and programs – and explain why it is critical to ensure that students know these before starting to program. The definitions are critical to understanding when teaching and using a wide-spectrum language such as Eiffel. We then describe our approach to teaching with Eiffel, including how we introduce agents to students, and how we integrate Eiffel techniques with software engineering practices such as testing. This approach, termed *specification-driven design*, encompasses elements from Extreme Programming and formal methods, and is highly suited (though not strictly dependent) on teaching with Eiffel. We then briefly outline how a typical course using this approach could be structured, and conclude with some observations.

2 Eiffel, Design-by-Contract, and Agents

Eiffel is an object-oriented programming language and method [12]; it provides constructs typical of the object-oriented paradigm, including classes, objects, inheritance, associations, composite ("expanded") types, generic (parameterised) types, polymorphism and dynamic binding, and automatic memory management. It has a comprehensive set of libraries – including data structures, GUI widgets, and database management system bindings – and the language is integrated with .NET.

A short example of an Eiffel class is shown in Fig. 1. The class $CITIZEN$ inherits from $PERSON$ (thus defining a subtyping relationship). It provides several attributes, e.g., $spouse$, $children$ which are of reference type (in other words, $spouse$ refers to an object of type $CITIZEN$); these features are publicly accessible (i.e., are exported to ANY client). Attributes are by default of reference type; a reference attribute either points at an object on the heap, or is $Void$. The class provides one expanded attribute, $blood\_type$. Expanded attributes are also known as composite attributes; they are not references, and memory is allocated for expanded attributes when memory is allocated for the enclosing object.

The remaining features of the class are routines, i.e., functions (like *single*, which returns *true* iff the citizen has no spouse) and procedures (like *divorce*, which changes the state of the object). These routines may have preconditions (**require** clauses) and postconditions (**ensure** clauses), but no implementations. Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, i.e., before any valid client call on the object. While we have used predicate logic in specifying the invariant of *CITIZEN*, it should be observed that Eiffel does not support this exact syntax. It does possess a notion of *agent* that can be used to simulate quantifiers like the ones used in the example; we discuss this in Section 2.2, and show how to rewrite the quantifiers given in Fig. 1 using agents there.

```
class CITIZEN inherit PERSON
feature {ANY}

  spouse: CITIZEN
  children, parents: SET[CITIZEN]
  blood_type: expanded BLOOD_TYPE

  single: BOOLEAN is
    do Result := (spouse=Void)
    ensure Result = (spouse=Void)
    end

feature {BIG_GOVERNMENT}

  marry is ...
  have_child is ...
  divorce is
    require not single
    do ...
    ensure single and (old spouse).single
    end

invariant
  single or spouse.spouse = Current;
  parents.count <= 2;
  for_all c member_of children it_holds
    c.parents.has(Current)
end -- CITIZEN
```

Fig. 1. Eiffel class interface

Other facilities offered by Eiffel, but not demonstrated here, include generic (parameterised) types, dynamic dispatch, multiple inheritance, and static typing. We refer the reader to [11] for full details on these and other features.

In teaching Eiffel, we give thorough coverage to the language, and consider all aspects of it, including agents, multiple inheritance (and its challenges), and covariant redefinition. We introduce language design principles that are supported or enforced by Eiffel, such as the query/command separation principle (which states that functional routines should be side-effect free). The discipline that these principles provide is generally appreciated and applied by the students in their projects and assignments.

2.1 Design-by-Contract

Design-by-Contract (DbC) is a mathematical description technique for engineering software systems with significant requirements for reliability and robustness. DbC is typically integrated with a programming language, providing formal annotations for interfaces of components and services. It differs from well-known formal methods such as B and Z in its cost: it can be selectively applied to those parts of the system associated with the highest risk; it integrates mathematical descriptions with code, ensuring consistency; and it is designed to be supported by tools that are comfortable and familiar to developers, e.g., compilers, debuggers, static checkers, and testing frameworks.

DbC recommends annotating classes with preconditions, postconditions, and class invariants. This was illustrated in the Eiffel example in Fig. 1. These assertions imply contracts that bind callers of class services with implementers of said services: callers guarantee to satisfy preconditions, while implementers guarantee to satisfy postconditions. This convention guarantees that conditions which may affect the correct operation of a class are checked only once. In Eiffel, these assertions are built in to the programming language, and the assorted Eiffel compilers and IDEs (e.g., ISE EStudio and GNU SmartEiffel) provide tools for managing and debugging assertions.

The benefits of using contracts and DbC are as follows.

– Contracts provide precise mathematical specifications of software and its services.
– Many views of the software can be automatically extracted. One view (automatically extracted) is the *contract view* that provides the client with the precise interface specifications. For example, a routine can only be invoked if the client satisfies the routine precondition.
– With assertion checking turned on, the contracts are checked every time the code is executed, and contract violations are immediately flagged. This test for consistency between code and specifications comes for free as opposed to the 3-fold cost mentioned earlier.
– Classes and components are self-documenting: the contracts are the documentation. There is no way that the documentation and code can become inconsistent, because the contracts are included within the code – the code would not execute if there are inconsistencies.
– Contracts provide design rules for maintaining and modifying the behaviour of components, cf., behavioural subtyping.
– Contracts provide a basis for formal verification. We discuss this further in the sequel when we suggest how Eiffel can form the basis for a formal verification course.

Contracts may of course be incorrect or incomplete, and thus need to be supplemented with a rigorous testing process. A key point to note then is that an Eiffel program annotated with assertions can result in errors due to incorrectly implemented functionality, or violation of contracts. As well, some conditions are extremely difficult to express using executable contracts – the paper [13] considers examples.

2.2 Agents

Agents in Eiffel are objects that represent operations. Agents can be passed to different software elements, which can use the object to execute the operation *whenever* they want. Agents thus provide a way of separating the definition of an operation from its execution. They also are a way of combining high-level functions (operations acting on other operations) with static typing in Eiffel. This should be contrasted with similar techniques, e.g., reflection in Java, which allows similar functionality at the cost of loss of static type checking.

Here is a simple example of an agent, using Eiffel's GUI library EiffelVision. Suppose you want to add the routine $eval\_state$ to the list of event handlers that will be executed when a mouse click occurs on the widget my_button. To carry this out, the following Eiffel statement would be executed.

$$my\_button.click\_actions.extend(\textbf{agent}\ eval\_state)$$

The operation being added to the button is indicated by the **agent** keyword. The keyword distinguishes an operation call to $eval\_state$ from a binding of the operation to the button. In general, the argument to $extend$ can be any agent expression. An agent expression will include an operation plus any context that the operation may need (e.g., arguments). The ability to supply context with an agent expression is essential. Suppose that you want to integrate the three-argument function

$$h(a : T1;\ x : REAL;\ b : T2) : REAL$$

over its second argument in the domain $[0, 1]$. Given a suitable integration scheme $integrator$ the following agent call will suffice.

$$integrator.integral(\textbf{agent}\ h(u, ?, v), 0.0, 1.0)$$

The question mark ? indicates an open argument (similar to a wild card, representing an element taken from the collection) that is provided by iterating through the range arguments provided.

To support agents in Eiffel, it is necessary to introduce a number of classes, including ones to represent $FUNCTION$ and $PROCEDURE$ operations, $PREDICATE$ operations, and arguments. These meta-level classes provide introspection facilities.

Predicate agents are of significant use; they feature heavily in how we introduce formal methods to students, and how we carry out testing. Predicate agents apply boolean-valued operations to collections. For example:

$$intlist.for\_all(\textbf{agent}\ is\_positive(?)) \tag{1}$$

$$intlist.exists(\textbf{agent}\ perfect\_cube(?)) \tag{2}$$

The first example applies the boolean-valued function $is\_positive$ to elements of the integer list $intlist$, and conjoins together the result. Equation (2) applies the boolean-valued function $perfect\_cube$ to elements of the integer list $intlist$ and disjoins the result. The question mark indicates an open argument that is provided by the list interator. Using this approach, we could rewrite the third clause in the invariant of class $CITIZEN$ in Fig. 1 as follows.

```
children.for_all((c:CITIZEN):BOOLEAN
  do
    Result := c.parents.has(Current)
  end)
```

The above example illustrates anonymous operations (i.e., the argument passed to the iterator `for_all`). c, the bound variable, is an element taken from the collection $children$, to which the body of the anonymous operation (contained within the inner `do..end` block) is applied. Operations bound in agent expressions may make reference to attributes and routines of objects, since when the operation is finally invoked, the operation will have been bound to a target object.

3 Specifications and Requirements

The terms "requirements" and "specifications" are ambiguous, and often used interchangeably in the literature. The traditional understanding of requirements is that they say *what* the system will do and not *how*. Students learning formal methods – and other software engineering techniques – often struggle with the distinction between requirements and specifications. This is particularly the case when they are using a wide-spectrum language like Eiffel or B. It is important that they understand the distinction in order to make it easier to validate systems against requirements, to clarify whether they are modelling the physical world or the system itself, and to simplify system designs. We thus spend a small amount of time – aproximately half a lecture – making the definitions more precise, following [9,16], though with a slight change in nomenclature.

The computer under description (consisting of software and/or hardware) is called the *System*, and it operates in an *Environment*. There are some *phenomena* (states, signals, events and entities) that the System and the Environment do not share, but there are also some phenomena that they do share – these are called the *shared phenomena* as illustrated by the intersection of the two ellipses in Fig. 2, which uses a banking system as an example.

In a banking system, the bank and its customers are not interested in hashtables or sort routines. The bank is interested in customer satisfaction, and that customers can request withdrawals or make deposits. An example of a requirement, written in temporal logic, is

$$[R1] \qquad \Box(x < balance \wedge withdrawal\_request(c,x) \rightarrow \\ \Diamond(withdraw(c,x) \wedge teller\_gives(c, dollar - bill))\ )$$

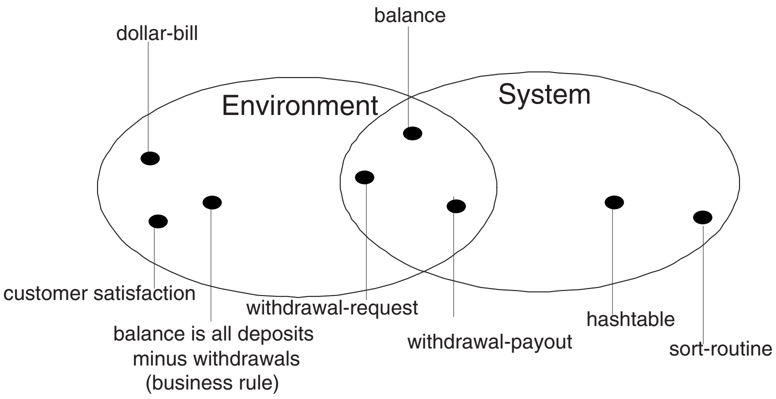

Fig. 2. The World: *System* and *Environment*

i.e., a withdrawal request of an amount x for customer c must lead to a withdrawal payout by the teller to the customer in *dollar-bill*s, provided that the requested amount does not exceed the current $balance$. The phenomenon *dollar-bill* is actual money in the bank teller's till (and is therefore not a System phenomenon), while $balance$ is a shared phenomenon. All the requirement phenomena are a subset of the of the Environment ($REQ \subset Environment$). Additional physical constraints or business rules (the application domain knowledge) will also need to be described.

The $balance$ is a shared phenomenon in $Environment \cap System$. It represents the debt owed by the bank to the customer. It may also occur as code in the System (e.g. see [S1] in Fig. 3).

Programs, in contrast to requirements, are concerned solely with the System phenomena. Programs may need to implement the requirements by using internal data structures and algorithms such as hash-tables and sort routines to do the job. All such program phenomena are a subset of the System.

The gap between requirements and programs is bridged by *specifications*. Specifications are concerned solely with the shared phenomena. Specifications are neither requirements nor programs. Specifications are unsatisfactory programs as they may not be executable. They are unsatisfactory requirements because requirements are not limited only to the coastline where the Environment and the System meet. But, specifications are useful as we transition from requirements to programs that will satisfy the requirements as specifications allow us to describe the black-box behaviour of the code at the observable input-output interface. For example, in order to satisfy requirement [R1], we might have a specification [S2] that specifies the behaviour of class $ACCOUNT$ as shown in Fig. 3. The specification in the figure is the contract view; hence, all the phenomena described in the figure are shared.

In the absence of implementation detail, the specification is not executable, but it can be compiled and automatically checked for type errors and the like by the compiler. Once implementation detail is provided, then the implemented code is automatically checked against the specification (i.e., the specifications now become "executable").

```
class ACCOUNT feature
    balance: INTEGER
    withdrawal(amount:REAL)
        require
            0 < amount and amount <= balance
        ensure
             balance = old balance - amount
 invariant
    balance >= 0
 end
```

Fig. 3. Specification [S2]

The perspective shown in Fig. 2 avoids the problem of implementation bias (when stating requirements) because no statements are made about the internals of the proposed System. Requirements and specifications are not descriptions of the state of the System, but rather a description of the state of the Environment. A specification might be compromised by a poor choice of designated phenomena or invalid domain knowledge, but it cannot overconstrain the implementation [16].

Eiffel, of course, can be used for capturing requirements, writing specifications, and implementation. The distinctions often escape students, especially early in their first software engineering course. We thus go over numerous case studies where we make the distinctions clear, and challenge them on this issue when they present reports, or participate in in-class discussions. We find that emphasising the distinctions as discussed above shows marked improvement in the clarity of assignment and project reports as the course proceeds.

4 Teaching Formal Methods with Eiffel

We have been teaching lightweight formal methods with Eiffel to third year Computer Science and Computer Engineering students for around six years now. Our approach has evolved over time, from simply using the Eiffel programming language to teach object-oriented techniques, to a whole-view approach for teaching software engineering, wherein formal techniques play a substantial role.

The course that we teach is one semester, and is project-focused, i.e., students (typically working in teams of 2-4) engineer working systems, based on requirements provided by the instructor. A substantial amount of instructor effort goes into the preparation of a comprehensive software engineering project that involves design (using UML or BON diagrams), implementation, testing, and documentation; we discuss this further later.

Students taking the course will already have some experience with object-oriented programming in Java. The typical student will have previously taken three one-semester courses that make use of Java as a programming language for introductory computer science, algorithms, and data structures. The students will also have taken two one-semester courses in logic and discrete mathematics and will have some experience with propositional and predicate logic, though their experience with using mathematics in programming (e.g., in introductory Computer Science courses that teach basic pre/postconditions and loop invariants) typically shows hesitancy in using these techniques in building systems.

After the usual preliminaries on software quality and engineering processes, the course leaps into a case study, designed to illustrate fundamentals of Eiffel and two typical approaches to building systems: plan-driven (in terms of modelling languages such as UML and BON) and test-driven. The case study proceeds by giving some informal requirements for a simple banking system. Use cases and scenarios are sketched very briefly, and from these a set of candidate classes is determined. The students are then posed the question: where should design commence?

This question is aimed at getting students to thinking about *acceptance tests* and testing in general. For the next stage is to introduce test-driven development [3] as (a) a general development technique, and (b) a way to introduce contracts (pre- and postconditions) and agents (for predicates) in a lightweight and indirect way. Our experience is that students who are less inclined towards mathematical techniques are more amenable to their study, use, and description if they are couched in terms of accepted engineering tasks, particularly testing, with which they have some experience. (We provide the students with an Eiffel testing framework, ETester, which supports unit tests and test suites, and also provides infrastructure for distinguishing between failures of the system versus failures in contracts. This is discussed elsewhere [10]. ETester has a similar design to JUnit for Java, but is targeted at languages that support contracts, where it is important during testing to distinguish failures in contracts from failures in code.)

Writing tests introduces the students to Eiffel's agent technology (used in ETester for automating testing) while at the same time serving to introduce them to the concept of writing formal specifications. Tests are a "backdoor" mechanism for introducing formal specifications and the usefulness of contracts. This is a critical point, which we now discuss.

4.1 Tests and Contracts Are Both Formal Specifications

The test-driven development (TDD) process described by Beck [3] is as follows.

1. Write a little test which may not work initially (especially if code hasn't been written for a class).
2. Make the test work quickly, focusing on doing the simplest thing that works.
3. Refactor the design to eliminate duplication and improve the style and architecture in terms of reusability and maintainability.

Unit tests (from TDD) and contracts (DbC) are both forms of specifications associated with shared phenomena of the System and the Environment. Unit tests can also be used to do regression testing at the end of coding, but in TDD, unit tests are seen as formal specifications that drive the design [1, p38 and p51].

Unit tests are normally used to check small chunks of a design. JUnit and E-Tester are useful tools for writing unit tests. However, these tools can also be used to write tests that verify higher level behaviours such as system-level tests involving the shared

Testing the design – collaborative specification [S1]

```
test_teller_withdrawal_request: BOOLEAN is
    local
        a: ACCOUNT
        t: TELLER_TRANSACTION
    do
        -- setup: 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)                          -- message 1
        t.withdrawal_request                       -- message 2
        Result := a.balance = 400 and t.succeeded  -- message 3
    end
```

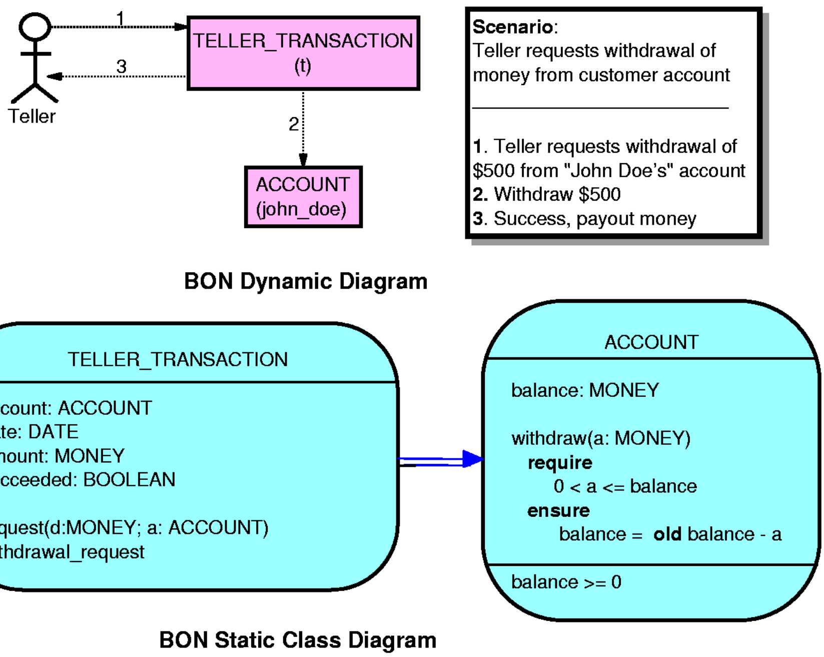

Fig. 4. Test and Design of a simple banking system

phenomena (Fig. 2). Such tests could verify parts of the collaborative behaviour of the kind mentioned in the small banking example involving requirement [R1]. We shall call these higher level tests *collaborative specifications*.

Collaborative specifications are related to UML interaction descriptions such as sequence or collaboration diagrams, and BON dynamic diagrams, especially where these diagrams describe a use case. The diagrams show a number of example objects and the messages that are passed between these objects within the use case. Such interaction diagrams provide an intuitive and visual way of describing (partial-order) executions of the system at the input-output interface, and therefore capture apsects of customer requirements useful in integration and acceptance tests.

Unfortunately, UML's interaction diagrams do not yet have a fully agreed-upon semantics, although various fomalizations have been proposed (but with conflicting semantic assumptions about the underlying communication system).

An equivalent to the interaction diagrams is the BON *dynamic diagram*. The BON dynamic and class diagrams for a simple banking system are shown in Fig. 4. These two diagrams describe the design of the system. At the top of Fig. 4 is a test that checks the dynamic diagram [14]. This test is an example of a collaborative specification (called [S1] in the figure). We note that [S1] is close to the temporal logic requirement [R1] described earlier in this section. It omits *dollar-bills* but otherwise mirrors all other phenomena described by [R1].

When we run the test [S1], we also exercise the detailed *contractual specification* [S2] for $ACCOUNT$ shown in Fig. 3. [S2] is repeated in the class diagram of Fig. 4.

The advantage of a test-based collaborative specification such as [S1] over the dynamic diagram (and the UML interaction diagrams) is that it is a formal artifact written in the same progamming language as the design. It can be compiled, type-checked and executed. If it fails, then there is a problem in the design, and if it succeeds then the expected behaviour is in place, at least for this execution. Finally, if the test succeeds then contracts such as those outlined in the design class diagram of Fig. 4 are also verified.

The key point to note with this approach is that it emphasises the use of different kinds of *formal specifications*, and that these specifications are introduced via testing. There is substantial value to introducing (particularly maths-phobic) students to formal specifications that can be executed, as they can get immediate feedback as to their quality and can immediately see their value in catching mistakes and clarifying assumptions.

The other point to note is that different syntaxes can be used for writing formal specifications. On one hand, they can be written using precise boolean logic, in terms of pre- and postconditions and class invariants. Or they can be written as executable unit tests or test suites. Or they can be written as dynamic diagrams. These are all useful syntaxes to know and apply.

4.1.1 Collaborative vs. Contractual Specifications.

Test-based collaborative specifications are incomplete, especially in contrast to the details that can be supplied by formal specifications in the form of contracts. Consider the following unit test.

```
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 specifies that array `<<4, 1, 3>>` must be sorted. But what about tests for all the other (possibly infinite) arrays of integers. Furthermore, the test does not check arrays of `REAL`, or arrays of `PERSON` (say by age). After all, the class `SORTABLE_ARRAY[G]` has a generic parameter `G`. Also, 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.

By contrast, the contract-annotated specification in Fig. 5 is a precise and detailed specification of the sorted array (the *count* attribute used in the figure is inherited from *ARRAY*). The quantifiers have been specified in Eiffel using the *agent* construct, as students would be required to do.

```
class SORTABLE_ARRAY [G -> COMPARABLE] inherit
  ARRAY[G]
feature
  sort is
    require
        count_positive: count > 0
        elements_not_void:
           Current.for_all((i:INTEGER): BOOLEAN
             do
               Result := (lower <= i <= upper) implies item(i) /= Void
             end)
    do
          ...
    ensure
     sorted:
        Current.for_all((i:INTEGER): BOOLEAN)
          do
            Result := (lower <= i <= upper) implies item(i) <= item(i+1)
          end)
          count_unchanged: count = old count
    end
end
```

Fig. 5. Contractual Specification

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 simple 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 (integers, people, etc.). Again, only an infinite number of unit tests could capture this property.

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

$$\forall\, s : STACK, x : G \bullet\ pop(push(x, s)) = s \tag{3}$$

Of course, given the complete contracts and an appropriate refinement calculus [15], the LIFO property can be expressed in the calculus as

$$push(x);\ pop \rightarrow skip \tag{4}$$

so that the LIFO behaviour indirectly *emerges* from the contractual specifications. The calculus can also be used to calculate if the implementation indeed satisfies the LIFO property. However, neither the description nor the calculation can be directly expressed as contracts.

By contrast, the emergent LIFO behaviour is easy to describe using collaborative specifications.

4.2 SDD

The approach that we teach students, which integrates elements of TDD and design-by-contract, is called *specification-driven design*; a full description of this approach is in [13]. It is an agile method tailored for Eiffel. It is perhaps surprising that agile techniques can be integrated successfully with formal methods. It is less surprising when one considers the commonalities between TDD and DbC.

- Both tests and contracts are formal specifications, each with their own limitations and advantages.
- Both TDD and DbC seek to transform requirements to compilable (high level) programming constructs as soon as possible. In TDD, tests describe designs. In DbC, contracts (and BON or UML diagrams) describe designs.
- Both TDD and DbC are automated (and hence easy-to-use) lightweight verification methods. Neither are complete, but both have simple-to-use tool support.
- Both are incremental development methods that intertwine analysis, design, coding and testing.
- Both TDD and DbC strive to obtain quality first for each unit of functionality, before proceeding to the next unit; this in effect aims to implement the Osmond Curve [2].

These commonalities indicate that we can combine TDD and DbC in a beneficial fashion. Boehm and Turner have argued in detail [6, p148] that (a) neither agile nor plan-driven methods provide a silver bullet; (b) each method has home ground where one clearly dominates the other; (c) future trends indicate that both will be needed; (d) it is better to build your method up than to tailor it down. Two case studies of actual

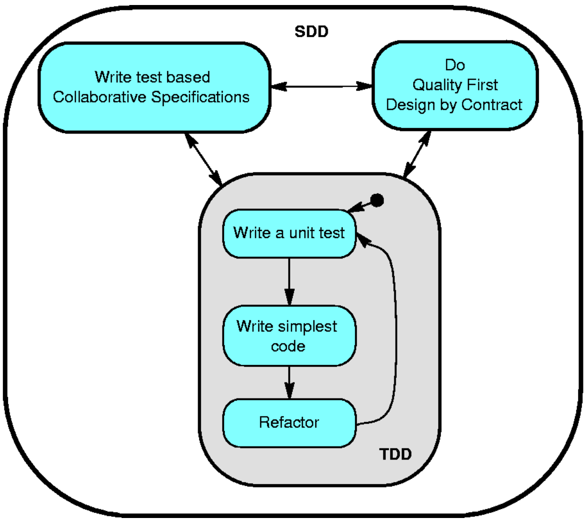
SDD
Write test based Collaborative Specifications
Do Quality First Design by Contract
Write a unit test
Write simplest code
Refactor
TDD

Fig. 6. SDD – Specification Driven Design

projects indicate how a balance can be achieved between agile and plan-driven method. The authors also detail the empirical evidence of how the costs and benefits of each method can be, and have been measured.

Conclusion (d) states that it is better to start small and elaborate one's development method over time. Boehm and Turner point out that plan-driven methodologies have a tradition of developing all-inclusive processes. Experts can can tailor the methodology down to fit a particular situation. However, non-experts tend to play it safe and use the whole thing often at considerable and unnecessary expense. Agilists offer a better approach of starting with minimal sets of practices, adding extras which can be justified by cost-benefits. This is extremely important for teaching: students may be turned off by presenting a large, complex methodology (or a large set of practices or principles), especially when they are first introduced to large-scale software engineering ideas.

In this spirit, the statechart of Fig. 6 describes *specification-driven design*. SDD is not complete but it does represent a core practices of a balanced method.

As described in Fig. 6, SDD does not dictate where to start – it is the developer's choice based on the context. However, whatever the starting point, the emphasis is on translating customer requirements to compilable and executable specifications, designs and tests. It might initially be possible to write a high level collaborative specification, or perhaps the developer wants to sketch out some class diagrams and contractual specifications.

Designs may need to be tested, or small chunks of design could be developed by TDD. In TDD mode, a developer might get stuck in an "infinite refactoring loop", indicating that more than an incremental design change is called for. The current chunk of design may need to be documented. This might be the right time to switch to DbC mode. In DbC mode, a developer might be stuck in the infamous "analysis paralysis", or the design may need to be tested and verified against the requirement. The trigger conditions on the statechart edges are thus not exhaustive.

Our experience in teaching with this approach is that captures the interest of students immediately, particularly because of the emphasis on executable specifications. Students see the value of formal specifications when they are couched in terms of testing. But at the same time, when testing is an emphasised part of development, they quickly see that tests as a form of specification are incomplete, and one needs to go further – and use contracts. We discuss our observations more in the next section.

4.3 Verification: Going Further

In current teaching with Eiffel we emphasise formal specification and lightweight verification and validation via testing. There is of course a desire to teach software engineering specialists more advanced techniques, in particular tool-supported formal verification, e.g., as with the B-Tool or with theorem provers like PVS. It is desirable to be able to teach verification using the same framework that students used for specification, design, and testing, so as to make use of students' existing expertise with notation and tools.

It is possible to teaching formal program verification using Eiffel. To this end, we have formulated the Eiffel Refinement Calculus, ERC. The full details are presented in [15]. ERC uses Eiffel as both a specification and programming language; Eiffel specifications consist of classes with contracts. A subset of the programming language – including reference types – is formalised using Hehner's predicative theory of programming [8], and verification rules are presented that show how to prove that a program satisfies a specification. The rules can be used for both after-the-fact verification, and for refinement. The rules are novel in that they apply directly to a subset of Eiffel, and allow introduction of object-oriented concepts such as method calls in refinement steps; translation to a non-OO language is not required.

More advanced projects can be defined to build on ERC, e.g., better tool support, case studies, addition of further constructs from Eiffel to the calculus (such as the novel **like** construct which introduces substantial challenges), and improving data refinement theories. This would be most suitable for an advanced post-graduate course on formal methods.

5 Some Observations and Conclusions

We now summarise some of our observations in teaching specification-driven design in Eiffel.

1. *The learning curve with SDD is initially very steep.* The students have a great deal of material to assimilate and apply in a short period of time: they must get to grips with Eiffel and TDD; they must learn elements of UML/BON for modelling; and they must learn how to write tests and to test effectively. Requirements for evaluation and teaching mean that students are given an assignment shortly after the start of term and need to submit their solutions within 3-4 weeks.
   The curve is somewhat lessened by the previous experience students have had with object-oriented programming and, to a smaller extent, writing up programming

assignments. To lessen the curve further, the course starts with a detailed case study showing the development process and how to use the Eiffel IDE, E-Tester framework, how to write test cases, etc. Students also work in pairs (one of the practices of Extreme Programming) on their assignments, and can work in slightly large teams on their projects.

2. *Student feedback is that the course is challenging,* but not because of the level of mathematics required or applied. It is challenging because of the scale of design projects that the students must work on (previously they have worked only on small projects), the amount of information that must be learned and understood over the course of approximately 12 weeks, and the level of rigour required in assignment and project write-ups. The fact that the knowledge accrued in the course is to be applied in future software engineering courses is considered very helpful.
3. *Students learn the formal method gradually.* Their initial use of the formal method is by writing collaborative specifications, i.e., BON dynamic diagrams and test cases. This also introduces them to Eiffel's agent syntax. They quickly realise the value of pre- and postconditions for capturing collaborative specifications that are awkward, complicated, or inexpressible. Their use of preconditions invariably starts at a very simple level, e.g., that arguments to methods are non-*Void*; postconditions are not used much initially. Postconditions are applied more broadly once appreciation for the complexity of collaborative specifications is gained. There is a feedback loop here – once complex unit tests start to occur during assignments and projects, there is more of an appreciation for the expressiveness of postconditions. And once complex postconditions appear, there is more of a reliance on the use of unit tests.
4. *Collaborative specifications are a better way to introduce agents* than some of the alternatives, particularly as a way of introducing predicate quantifiers into contracts. There is resistance from many of the students to using quantifiers in contracts when they are not shown motivation via testing.
5. *Eiffel is underappreciated at first.* There is backlash against it from some students at first, in the sense that they desire to learn a more 'commercially relevant' language, e.g., C++ or Java. We deal with this in several ways: by introducing them to simple GUI techniques in Eiffel in the course of the initial case study; by presenting information on the industrial use of Eiffel; and by carrying out comparisons with competing technologies throughout the course. As well, the students have previous experience with Java and are generally aware of its limitations and abilities. When starting with Eiffel, the students find its support for generic classes and contracts very helpful.

We should not underplay the substantial time and effort that the instructor must put into making the course a success. Setting up suitable assignments and projects for this course is challenging, and requires much effort, especially in producing quality solutions. It is also difficult to obtain high-quality teaching assistants and laboratory demonstrators familiar with Eiffel and agile techniques. It is of course possible to teach such a course using Java (or C++) based technologies (e.g., by using a contract framework and testing framework for such a language), and we have done so, but in our experience the tools that support contracts in these languages are typically weaker than Eiffel's tools. While the iContract preprocessor for Java provides much the same functionality as Eiffel's contract mechanism, it does not have Eiffel's integrated tool support, e.g., a built-in debugger for tracing contract failures. As well, the fact that iContract is a preprocessor makes it more cumbersome to use for editing contracts, testing them, and keeping the code consistent with the contracts.

We have taught this course using both ISE EiffelStudio (a commercial tool) and GNU SmartEiffel, on a variety of development platforms. We have generally found it easier to make use of EiffelStudio, in part because of its complete integrated development environment (including debugger) and more comprehensive set of libraries. The SmartEiffel compiler and toolset is powerful and generally straightforward to use, but the lack of integrated debugging facilities is a weakness at this stage.

The general feedback that we have received on the course is predominantly positive: the course is perceived as challenging, hard work, practical, and insightful. A few students remain unconvinced by Eiffel as an industrially applicable language; some have been completely convinced and are now using Eiffel in industrial development [7]. Most have commented that they now think they have a better understanding of the principles of software engineering, object-oriented design, formal methods, and formal specification, which is in the end a good measure of success.

References

1. S. Ambler. Extreme Testing. *Software Development*, 11(5), June 2003.
2. S. Ambler. Agility for Executives. *Software Development*, September 2003.
3. K. Beck. *Test-driven Development: by example*, Addison-Wesley, 2003.
4. K. Beck, A. Cockburn, R. Jeffries, and J. Highsmith. Agile Manifesto www.agilemanifesto.org/history.html. 2001.
5. D. Berry. Formal methods: the very idea – some thoughts about why they work when they work. *Science of Computer Programming*, 42(1), 2002.
6. B.W. Boehm and R. Turner. *Balancing Agility and Discipline: a guide for the perplexed*, Addison-Wesley, 2003.
7. H. Cater. Strategic Command 2, web site at www.battlefront.com, last accessed June 2004.
8. E.C.R. Hehner. *A Practical Theory of Programming* (Second Edition), Prentice-Hall, 2003.
9. M. Jackson. *Software Requirements and Specifications*, ACM Press, 1995.
10. J.S. Ostroff, R.F. Paige, D. Makalsky, and P.J Brooke. *ETester: an Agent-Based Testing Framework for Eiffel*, submitted June 2004.
11. B. Meyer. *Eiffel: the Language* (Second Edition), Prentice-Hall, 1992.
12. B. Meyer. *Object-Oriented Software Construction (Second Edition)*, Prentice-Hall, 1997.
13. J.S. Ostroff, D. Makalsky, and R.F. Paige. Agile Specification-Driven Design. In *Proc. Extreme Programming 2004*, LNCS, Springer-Verlag, June 2004.
14. R.F. Paige, J.S. Ostroff, and P.J. Brooke. A Test-Based Agile Approach to Checking the Consistency of Class and Collaboration Diagrams. In *Proc. UK Software Testing Workshop*, University of York, September 2003.
15. R.F. Paige and J.S. Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel. *Formal Aspects of Computing* 16(1):51-79, Springer-Verlag, April 2004.
16. P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1), 1997.