Testing object Interactions

Grüner, A.

Chapter 1

Introduction

On Wednesday, the 30th September 2009, at 3.46am a data transmission problem, caused by a routine software update, resulted in a crash of an airline company's software system [58]. Specifically, the check-in systems of the airline and of some partner companies at more than 200 airports around the world were affected. The ground staff had to fall back on an older system of the 1970's – which basically consisted of writing passenger lists, boarding passes, and luggage tags, manually.

This system crash example joins a long list of more or less well-known software failures. Although the recent crash does not represent an overly spectacular software failure, yet it demonstrates that, on the one hand, developing complex software systems is still error-prone and, on the other hand, that the economy highly and increasingly depends on such software systems. Indeed, already in the late 1960's, software developers and scientists were aware of the discrepancy between the need for complex software systems and the difficulty of writing correct and reliable computer programs. It was F. L. Bauer who coined the term "*software crisis*" at the first NATO Software Engineering Conference in 1968 [52], in order to refer to the above mentioned software development dilemma. And in [23] Edsger W. Dijkstra stated in 1972:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

Since then various software development approaches, methodologies, and processes have been developed to come out of the crisis. Indeed, although the "silver bullet" has not been found yet, some progress has been made. The development of *high-level programming languages*, for instance, is based on the idea that language constructs abstracting from the details of the computer hardware allow a more efficient and less erroneous software development process. In particular, *object-*

oriented programming languages enjoy great popularity, as the object-oriented approach also facilitates the re-use of software components due to encapsulation and information hiding.

Moreover, to cope with the software dilemma, several standardized *software development process models* have emerged promising more predictable and more successful software projects. To achieve this, they subdivide the project into several smaller tasks and activities by the help of management and engineering techniques.

Besides efficiency, one key aspect of the software development processes in general is *quality assurance*. Although there has been much progress in recent years, proving the correctness of a software system by means of formal verification is still not feasible for most of the real world software products due to its high complexity. Thus, *testing* is still of prime importance in assuring the quality of software. In contrast to exhaustive methods for system verification and validation, testing aims at detecting faults, thus increasing confidence in the system under test [55, 71]. However, testing a complex product en bloc at the end of the development process, as it has been done in the early days of software testing, is not feasible anymore, either. In fact, to manage the complexity of modern software, testing has become a systematic operation, conducted on different levels and integrated into the software development process [33].

This thesis deals with testing at its lowest level, i.e., *unit testing*, of object-oriented software. In the remainder of this chapter, we first introduce the context of this work. Specifically, in the following sections, we will say some preliminary words regarding object-oriented languages, software development process models, and software testing. In particular, we will have a close look at some existing unit testing approaches for object-oriented languages. Finally, we will give a short overview of our testing approach and of the thesis.

1.1 Object-oriented programming languages

In object-oriented programming languages, a key concept is to *encapsulate* implementation details within so-called *objects* which represent an association of data with its operations. Specifically, the operations provide an *interface* between the object's data and its user rendering it unnecessary for the user to access the data directly. Due to this *information hiding*, the developer of the object, on the one hand, is free in choosing (or even in changing) the appropriate implementation for the data as well as the operations and, on the other hand, the user does not have to rely on a specific implementation but only on the interface. This idea can be traced back to the early 1960's. One of the first and probably most influential object-oriented languages was *Simula* [22], developed by Ole-Johan Dahl and Kristen Nygaard. *Simula* did not only introduce the concept of objects but also the notion of *classes*, used as "blue-prints" from which new objects can be instantiated. Moreover, the language supported *sub-classing* and *overriding*. That is, a class can inherit the data types and operations of another class and, beyond that,

it may re-define such an inherited operation by providing a new implementation.

Inspired by *Simula*, Alan Kay led the development of *Smalltalk* in the 1970's [30]. With *Smalltalk* Kay introduced the term *object-oriented programming* to express the pervasive use of objects and messages passing. Indeed, in *Smalltalk* everything is an object, including classes which can be created and modified dynamically.

As for the mainstream software application development, the object-oriented programming approach had its break-through in the early 1990's largely due to C^{++}, developed by Bjarne Stroustrup [67]. The programming language C^{++}, originally named "C with Classes", can be considered as an extension of the language *C* by object-orientation. Stroustrup developed C^{++} with the intention to make *Simula*'s object-oriented features available for real word software applications, since *Simula* was too slow for practical use. In fact, *C* regarded as a middle-level language, was and still is one of the most popular programming languages due to its execution speed.

The high performance of nowadays computer hardware, however, allows to use more high level computer languages also for most of the mainstream software applications. As a consequence, lots of high level programming language and scripting language with support for object-orientation have been developed. In the following, we will discuss two widely used representatives in more detail, namely *Java* and C♯.

1.1.1 Java

Java is an object-oriented class-based general-purpose programming language which was developed at Sun Microsystems by a team headed by James Gosling. It was first released in 1996 [21]. Aiming at embedded systems, *Java*'s predecessor, *Oak*, was considered to be derived from C^{++}. Due to the lack of portability, however, the team decided to design a completely new language. Though, the syntax of *Java* is still inspired by *C* and C^{++}.

In contrast to the lower-level language C^{++}, *Java* does not allow pointer arithmetics. Specifically, a reference to an object is not represented by a pointer to a specific memory cell. Moreover, the language supports automatic garbage collection, i.e., the programmer needs not to allocate or de-allocate memory for objects, explicitly. *Java* is not fully object-oriented as it supports base types for integer or boolean values, for instance. However, for each base type there exists a corresponding class in *Java*, as well.

Java class definitions can be bundled to so-called *packages* which facilitate the re-use of class libraries [66]. In particular, the *Java* runtime environment comes with a huge class library including, among other things, thread classes allowing for a concurrent flow of control.

A *Java* program is compiled to *Java bytecode* which is executed by the *Java virtual machine* (JVM). *Java* bytecode is generally platform-independent. There exist JVM implementations for many computers and devices. For instance, today almost every cell phone is equipped with a JVM.

1.1.2 C♯

The programming language C♯ [25], first released in July 2000, can be considered as Microsoft's answer to *Java*. Developed by Anders Hejlsberg, the author of Turbo Pascal and chief designer of Delphi, C♯ is also an object-oriented class-based general-purpose programming language whose syntax likewise resembles that of C^{++}. Beyond that, it shares many other features with *Java*, like automatic garbage collection and the support for multi-threading. A C♯ program, too, is compiled to bytecode, called *Common Intermediate Language* (CIL), which is to be executed by the *Common Language Runtime* (CLR).

Apart from many similarities, C♯ provides some additional features which do not exist in *Java*. In contrast to *Java*, for instance, C♯ does support memory address pointers in order to increase execution speed in time critical applications. However, to prevent pointers from becoming a general security leakage, they may only be used within blocks which are to be marked as unsafe; unsafe blocks, in turn, need appropriate permissions to run. In this context, C♯ developers distinguish code which exclusively relies on automated garbage collection from code which includes user-allocated memory usage by the terms *managed* and *unmanaged code*.

Furthermore, C♯ introduces the concept of *delegates*. A delegate is a reference to an object's method which, in particular, can be passed around via method call parameters and return values. Consequently, a delegate may be invoked like a conventional method, although the caller need not to know the object of the method. However, the invoked delegate itself may access the object's fields and other methods.

Though, summarizing, there certainly exist some differences between *Java* and C♯, currently their similarities prevail. Aiming at object-oriented language more generally, in this thesis, we want to abstract from specific, distinguishing features but concentrate on the common characteristics of both languages. To this end, we will define and use a small object-oriented language intended to capture the object-oriented concepts that both languages have in common.

1.2 Testing in the software development life-cycle

It has been said, that several models regarding the software development process have been developed. Let us quickly discuss the basic idea of these models where we are specifically interested in the involved testing activities. Generally, the goal is to find repeatable and predictable processes that improve productivity and quality. In particular, to get a grip on the complexity of such a project, it is divided into smaller tasks. To this end, most models distinguish roughly the following phases:

- planning phase

  Usually, a software development project starts with a planning phase. The most important task within this phase is the *requirements analysis* where

the customer's needs and requests are gathered in a systematic way. This may lead to feasibility studies and first estimations regarding the effort, costs, and time needed.

- design phase

  Within the design phase, the overall *architecture* of the software system is to be determined. A hierarchy of subsystems and components is identified, such that the development processes can be divided into smaller manageable parts. The results includes a *specification* for each of the system's component capturing its requirements and its collaboration with other components.

- implementation and testing phase

  Based on the specification results of the design phase, the components are implemented. Furthermore, the specification of a component should be used as reference for a *component or unit test* where the component is tested in isolation. Following the hierarchy of the architecture, components are integrated resulting into larger components which in turn have to be tested by means of *integration test* activities. The idea of integration testing is to check whether the integrated components *interact* with each other as specified. The final integration test is called *system test* where the complete system is integrated.

- deployment and maintenance phase

  After completing and integration-testing the system, it is subject to an *acceptance test* with the customer. By this test, the customer checks whether the software meets the original requirements. Finally, if the acceptance test was successful, the software has to be integrated into the customer's production environment. However, in general this is still not the end of the software life cycle. For, often problems or improvement suggestions arise only during the daily operational use resulting into bug tracking or further software enhancement tasks.

A good example of a software development model demonstrating the relation between the actual development acitivities and the correpsonding testing activities is represented by the *V-model* (also: VEE model). The origins of the V-model can be traced back to the early 1980's [19]. Compared to its predecessor, the waterfall model, it has an emphasize on quality assurance aspects. Specifically, for each development phase it introduces a testing phase in which the results of the corresponding development phase are tested. As can be seen in Figure1.1, the V-model's course of action is often graphically represented in form of a V, hence, the model's name. The horizontal dashed lines indicate that test cases of a specific development phase should be formulated during the development phase itself, already.

As mentioned earlier, this thesis focuses on unit testing. For, a common statement is the later a software failure is observed during the software development

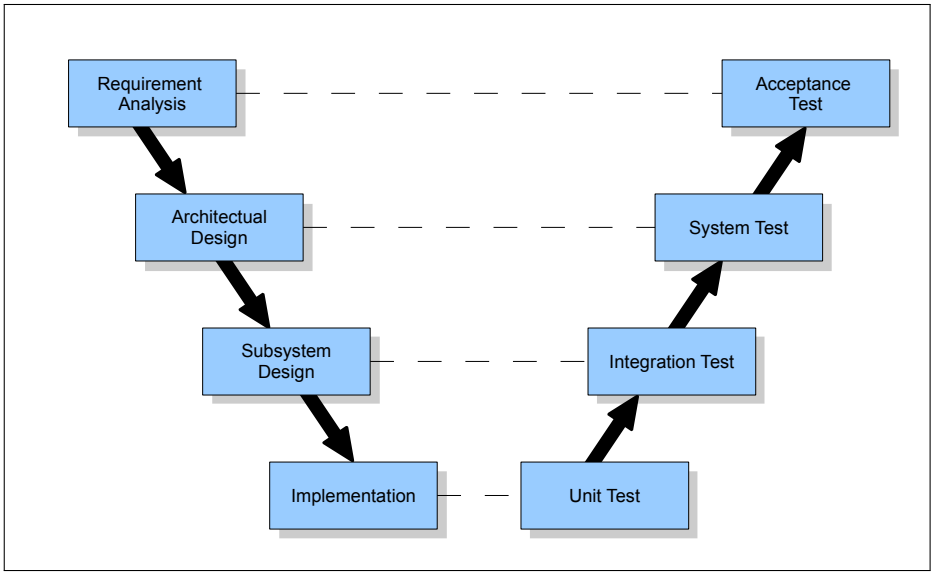

Figure 1.1: Software development process and testing levels

life cycle the more cost effect is the finding of the corresponding defect. For instance, B. Boehm and V. Basili state in [18] that finding and fixing a software problem after delivery is generally 100 times more expensive then finding and fixing it earlier. On the other hand, according to Jones, 85% of software failures are introduced during the design and the (low-level) implementation phase [39]. Therefore, low-level testing, i.e., unit testing, seems to have a key position for efficiency and quality in the software development process. This may be a reason why unit testing enjoys such a prominent role in agile software development processes like *extreme programming* [11]. For instance, concerning the extreme programming methodology, all code must have unit tests and all code must pass all unit tests before it can be released. In the following we will discuss three existing unit testing frameworks.

Before we discuss some exemplary unit testing approaches of the object-oriented world in the next section, however, let us first fix a terminology pertaining to testing that we will use in this thesis. Specifically, we will resort to corresponding definitions given in [61] (see also [63]).

Definition 1.2.1 (Errors, defects, and failures)**:** If a software developer makes an *error* (mistake), this results in a *defect* (fault, bug) in the code. If a defect in the code is executed, it may become observable in terms of a *failure*, i.e., the system may fail to do what it should do (or do something it should not do).

1.3 Unit testing object-oriented software

Unit testing in general represents the idea of automatically testing small program fragments, i.e., a unit or component, by executing a *test program* which incorporates the unit under test [62]. In the context of procedural or functional programming languages, a unit is often considered to be a procedure or, respectively, a function. Concerning object-oriented languages the smallest unit is used to mean an object or a class (depending on whether one wants to refer to dynamical, or respectively, static aspects). To test a unit, a test program *interacts* with the unit and possibly investigates the resulting change of the program state. While in a procedural or functional language these interactions are usually carried out in terms of function or a procedure calls, an object-oriented test program investigates the unit by means of *method calls*. A central aspect of object-oriented programming, however, is that objects heavily rely on the interaction with other objects. Thus, most often, an object-oriented unit test program has to ensure the existence of several *collaborator* objects that are required to cooperate as assumed by the unit in order to enable the unit to fulfill its tasks. Due to this fact, it is generally accepted that, regarding object-oriented systems, unit testing coincides with integration testing or, at least, that the dividing line between the two testing activities is blurred (cf. [6] and [16], for instance). As a consequence, writing unit tests in an object-oriented setting is usually considerably more complex. However, a couple of testing frameworks exist that aim at unit testing object-oriented components. We will have a closer look at three of these frameworks. The first two frameworks are widely used, specifically by the extreme programming community. Despite its usefulness, the third framework is not so commonly accepted. All three frameworks aim at the *Java* programming language. However, similar approaches do exist also for C♯ and other related object-oriented programming languages.

To allow for comparison, each framework is illustrated by a simple example, realizing the test of a *voting system*. The voting system is a component that, when activated by an initiator, collects a vote from a group of external voter objects, compiles a report, and returns it to the initiator. It can be used, for example, to detect termination of a group of objects.[1] In our example, the voter system is implemented by means of a class *Census* defining a method *conductVoting* which realizes the above mentioned voting procedure. In particular, the method expects a list of *Voter* objects which, in turn, yield their vote in terms of a return value of a method *vote*. An exemplary implementation is sketched in Listing 1.1.

1.3.1 JUnit

JUnit [41] is a unit and regression testing framework written by Kent Beck and Erich Gamma. It has its origin in Kent Beck's *SUnit* [10], a unit testing framework for *Smalltalk*, and by now many adoptions to other languages exist. The collection of *JUnit* derivatives is often referred to as *xUnit*.

[1] We will, however, restrict our considerations on sequential programs until Part II of the thesis.

Listing 1.1: The voter system

```
1  class Voter {
2    public Boolean vote() {
3      ...
4      return(value)
5    }
6  }
7
8  class Census{
9    public Boolean conductVoting(List voters) {
10     Boolean result = true;
11     for (Voter v : List) {
12       result = result && v.vote();
13     }
14    return(result)
15   }
16
17 }
```

The intention behind *JUnit* is to encourage software developers to write and execute tests themselves instead of shifting the responsibility on to some other software tester [12]. Software testing small units of code should become part of the code writing process. To integrate unit testing into the code writing process, Beck and Gamma suggest a development paradigm which is often called *test-driven development* (*TDD*). *TDD* represents a cornerstone of *Extreme Programming* [11] and other agile development approaches. In TDD developers write code incrementally by extending the unit only by one small feature at a time. More specifically, first small test cases for the new feature are written and afterwards the corresponding code is implemented. This is followed by exercising the unit tests. Only if all tests terminate successfully, the developer goes on to extend the unit with the next feature. It is worthwhile to say that the unit should not only pass the new tests but also all the other test cases of previously implemented features are executed, i.e., after extending the unit with a new feature, the developer does *regression testing*.

To avoid obstruction of the developers flow of work, *xUnit* tests are written in the same programming language as the *production code*, hence, *JUnit* tests are written in *Java*. A test (case) in *JUnit* is basically a *Java* program which incorporates and executes the code of the unit under test to decide on success or failure. To this end, the *JUnit* framework consists of a small and simple set of *Java* classes which expedites and unifies the recurring tasks when writing test code. The recent transition from version 3.8.1 to 4.0 entailed some major changes concerning the implementation and the usage of *JUnit*. As the former version is still widely-used, we will sketch both versions in the following.

The developer writes tests in terms of *Java* methods. Typically she or he

implements

- an (optional) setup method which initialises the data needed for the test case (in *JUnit* terms: the *test fixture*),
- the actual test case methods consisting of the interaction with the unit under test and the test evaluation,
- and an (optional) tear down method to free resources which possibly have been reserved by the setup method.

In version 3.8.1 and former versions of *JUnit*, the above mentioned methods have to be implemented in a subclass of *org.junit.Test.TestCase*. The setup and the tear down method are realized by overwriting the methods *setUp()* and *tearDown()*. The test case methods are methods with an arbitrary name[2] and without any parameters. *JUnit* allows for defining several test case methods within a single test case class, which therefore share the same *tearDown()* and *setUp()* methods, i.e. all test case methods of one test case class will be executed against the same test fixture. However, every instance of a test case class always executes only one test case method. Thus, for each instance the developer has to designate the desired test case method. This can be done either statically or dynamically. In the first case, the developer has to overwrite the *TestCase*'s method *runTest()* which is expected to call the designated test case method. In the latter case the name of the desired method is passed to the test case instance via a parameter of the constructor. *JUnit* uses reflection to find and execute the corresponding method.

With *JUnit* it is possible to execute a batch of test case methods to realize automated regression testing. To this end, test case instances can be grouped to test suites (which again can be grouped to other test suites, allowing for a tree structure of test cases). Again, *JUnit* supports a static and a dynamic way to add test cases to a test suite. Either the developer adds a test case by passing a corresponding test case *instance* to an instance of *JUnit*'s *TestSuite* or he passes a test case *class* to a test suite which then will, again, use reflection to create and add instances of that class for every test case method within the class. In that case, however, *JUnit* uses a naming convention to find all test cases at runtime by name, i.e., all test case methods must start with *test*. Finally, one has to implement a static method *suite()* which returns a test suite that contains all test cases to be executed.

A test case method typically calls a method of the unit under test and checks afterwards the return value or the resulting side effect of the method call. For this purpose, *TestCase* provides a set of assertion methods with a boolean parameter which is used to decide on success or failure. The method *assertTrue*, for instance, expects a boolean expression that has to evaluate to true, otherwise the test is considered as failed.

[2]However, to be able to use some test automation provided by *JUnit*, methods names have to obey certain naming conventions discussed later.

The biggest change that came with version 4.0 was the usage of *Java*'s annotations [36]. By using these annotations, developers need not to subclass *TestCase*, anymore. Instead, they mark a method as a test case, a tear-down, a setup, or a suite method by annotating them with a certain keyword. Moreover, additional keywords for new features have been introduced. For instance, apart from the tear-down and setup keywords which enables one to create and, resp., remove a fixture for every instance of a test case class, there exist new keywords which allow to create and remove parts of the fixture only once for all instances of a single class.

One criticism on *JUnit* 3.8 was the poor support for testing exceptions. If one wanted to ensure that a certain exception is raised in a certain situation, it was necessary to write a test case method which catches the corresponding exception if it has been raised. On the other hand, if the exception was not raised, one had to call the *fail* method of the *JUnit* framework manually to indicate that the test has failed. In *JUnit* 4 this is no longer necessary. Instead one can annotate a test method with the expectation of a certain exception.

Listing 1.2 and Listing 1.3 show the voter example for the test framework of *JUnit* 3.8.x and *JUnit* 4.x, respectively. The test fixture consists of three voters, instances of anonymous sub-classes of *Voter* to allow for different voting results, and one census object which is the actual unit under test. The sole test case method calls *conductVoting* of the census object and passes the above mentioned voters. After that it checks whether the result of the method call is as expected.

Essentially, the *JUnit* framework knows only one test pattern: call a method, wait until it returns and check the outcome. However, sometimes the developer wants to test not only the outcome at the end of a method call but also wants to ensure some features about the *interaction* in between the invocation and the return. For instance, in our example, we would like to ensure that *conductVoting* does not come to the right voting result only by chance, but that it indeed inquired the involved voting objects, i.e., we would like to test, whether *conductVoting* calls the method *vote* of the voter objects. This is not possible to test with *JUnit*[3].

There exist other unit testing frameworks in the style of *JUnit*. For instance, currently the strongest competitor of *JUnit* is most likely *TestNG* [68]. However, all these frameworks suffer from the lack of interaction test support.

1.3.2 jMock

The *Java* library *jMock* [38], developed by Nathaniel Pryce et alia, is also used for unit and regression testing of *Java* programs.

The *jMock* approach follows the idea that for testing object oriented systems it is more appropriate to test the interactions among objects rather than to test

[3]Actually, it certainly is possible to test this with *JUnit*, as a *JUnit* test is a normal *Java* program. Hence, every test that one can write in *Java* in general, can be embedded in the *JUnit* framework. But the test of interactions is not supported by *JUnit* directly, which means that one has to write additional code to realize this kind of tests

Listing 1.2: Voter example: *JUnit* 3.8.x

```
1  import org.junit.Test.TestCase;
2
3  class CensusTestCase extends TestCase {
4    Vector voters;
5    Census census;
6
7    protected void SetUp() {
8      voters = new Vector({
9        new Voter { Boolean vote() { return true } },
10       new Voter { Boolean vote() { return false } },
11       new Voter { Boolean vote() { return true } }
12     });
13     census = new Census();
14   }
15
16   public void testVoting() {
17     Boolean result = census.conductVoting(voters);
18
19     Assert.assertTrue(result == false);
20   }
21
22 }
```

the change of the program state caused by the interactions [28]. For, in object orientation it is often not possible to observe the state due to encapsulation. Moreover, even if an undesired state change has been observed then in many cases one has to identify the causing interaction, anyway.

For *interaction-based testing* [27], the developer has to identify the interaction partners of the unit under test and replace them by so-called *mock objects* [45] (regarding interaction-based testing, see also [57] and [8]) The task of these mock objects is to mimic the original environment objects of the unit and at the same time to verify assertions about the occurring interactions. Replacing the environment object by tester objects also makes sure that the unit is tested in isolation, i.e. the test is insulated from other possible failures caused outside of the component. Finally, this approach supports TDD, as even units can be tested whose final environment objects do not yet exist in the production code.

Usually *jMock* is applied on top of the *JUnit* testing framework which means that the developer writes *JUnit* test but utilizes the *jMock* library to formalize a behaviour-based testing with mock objects. Thus, one writes *JUnit* test case classes (if used with *JUnit* 3.8 or less) as described above with the following differences:

- Within the setup method, instead of setting up the test fixture by constructing the unit's environment by means of objects of the production code, one

Listing 1.3: Voter example: *JUnit* 4.x

```
1  import org.junit.Test.TestCase;
2
3  class CensusTestCase {
4    Vector voters;
5    Census census;
6
7    @Before
8    protected void createVotersAndCensus() {
9      voters = new Vector({
10       new Voter { Boolean vote() { return true } },
11       new Voter { Boolean vote() { return false } },
12       new Voter { Boolean vote() { return true } }
13     });
14     census = new Census();
15   }
16
17   @Test
18   public void conductVotingAndCheckResult() {
19     Boolean result = census.conductVoting(voters);
20
21     Assert.assertTrue(result == false);
22   }
23
24 }
```

creates mock objects correspondingly. However, as in common *JUnit* tests the object under test is certainly instantiated from a class of the production code.

- Within the test case method, one, firstly, formalizes the expected interactions between the component under test and the mock objects. Then, second, the unit's method to be tested is invoked and, finally, the expectations are verified.

The *jMock* library's basic idea is to support the creation of mock objects and the formalization of the expectations. However, the authors soon realized that in particular the design of the API for the formalization of the expected behavior has to be chosen carefully, as otherwise formalizations easily become tricky and error-prone. Thus, the design of the library is based on what the authors call an *embedded domain-specific language* [29] (EDSL). The idea is to provide developers a language for specifying interface behavior (hence, a domain-specific language) in terms of *Java* expressions. A key concept for this is *jMock*'s call chain syntax, where each method call to a *jMock* object yields another *jMock* object (or even the callee itself) such that another method invocation can be appended

without the need of prefixing the callee's name. A small example might clarify this. Suppose, one wants to formalize the expectation that the unit under test calls a method *buy* of object *mainframe* exactly once with parameter equal to the constant *QUANTITY*. Moreover, *buy* shall return the object *ticket*. Then one can write

```
mainframe.expects(once())
  .method("buy")
  .with(eq(QUANTITY))
  .will(returnValue(ticket));
```

Note, that the determination of the return value *ticket* does not represent an assertion on the behavior of the unit but stipulates the committed mock object's behavior towards the unit.

Listing 1.4 shows the voter example in terms of a behavior-based testing approach formalized with the help of the *jMock* library on top of the *JUnit* 4 framework. First, a *Mockery* object, *context*, is created which represents the entry point to the library. Moreover, an array of three mock objects of the *Voter* class is created. In the test case method *conductVotingAndCheckBehaviorAndResult* the expectations are formalized; the method *vote* of every voter (mock) object has to be called once. Additionally the return values are stipulated. After a call to the object under test *census* the result is checked.

With the help of the *jMock* library, the previous *JUnit* test example (Listing 1.3) has been improved in that now also the calls to the voter objects can be checked. However, although the authors put much effort into the design of the library, even this small example shows that there is still much "syntax noise", as Freeman and Pryce called it. In [29] they investigated the possibilities to create an EDSL for a more abstract description of the unit's behavior in context of a general purpose language like *Java* and concluded that "on the whole, it's too hard to extend conventional host languages, the syntax and the low-level operations get in the way".

Nevertheless, the mock object approach seems promising and by now several other implementations for *Java* exist. For instance, *EasyMock* [24] is also a well-known mock object library for *Java*. *EasyMock* tries to reduce the syntax noise by following the record-play idea. To formalize the expectations one calls first the mock objects methods as it is expected from the unit under test. After this "recording" step, the mode of the mock object is changed such that the mock object now expects (and realizes) the same interaction again. A drawback of this approach is that *EasyMock* by default only supports the generation of mock objects for interfaces. Moreover, this approach suffers from less expressiveness compared to the *jMock* approach.

Finally, although testing the observable interface behavior of a unit means also a higher, more abstract approach, by now no mock object implementations for *Java* support the use of the results for further analyses.

Listing 1.4: Voter example: *jMock*

```
1  import org.jmock.Expectations;
2  import org.jmock.Mockery;
3  import org.jmock.integration.junit4.JMock;
4  import org.jmock.integration.junit4.JUnit4Mockery;
5
6  @RunWith(JMock.class)
7  class CensusTestCase {
8    Mockery context = new JUnit4Mockery();
9    final Voter voters[] = {context.mock(Voter.class),
10     context.mock(Voter.class), context.mock(Voter.class)}
11
12   Census census;
13
14   @Before
15   protected void createVotersAndCensus() {
16     census = new Census();
17   }
18
19   @Test
20   public void conductVotingAndCheckBehaviorAndResult() {
21     context.checking(new Expectations() {{
22         one(voters[0]).vote(); will(returnValue(true));
23         one(voters[1]).vote(); will(returnValue(false));
24         one(voters[2]).vote(); will(returnValue(true));
25     }};
26
27     result = census.conductVoting(voters);
28     Assert.assertTrue(result == false);
29   }
30
31 }
```

1.3.3 JMLUnit

By using the *Java Modeling Language* (*JML*) [44], the unit testing tool *JMLUnit* [37] allows for specifying unit tests on a higher and more abstract level than *JUnit* or *jMock* do. In particular, developers need not to write the test code on their own but it is generated by *JMLUnit*.

JML is a specification language for *Java* programs which is used to formally specify the interface behavior of a *Java* module. It is based on the design-by-contract [46] (*DBC*) approach in the style of the *Eiffel* programming language [26]. *Eiffel* provides language constructs for defining contracts between method callers and method callees. These constracts consist of program code stating pre- and postconditions of methods and invariants of classes. Formalizing contracts in terms

of executable code, firstly lowers the burden on the programmer who does not need to learn an additional specification language, and secondly, it means that violations of the contracts can be detected at runtime. Since *Java* does not support *DBC* originally, *JML* specifications are stated in terms of special *Java* annotations embedded in the unit under test. The *JMLUnit* tool, in turn, extracts these annotations in order to generate code for *JUnit* test cases.

To allow for more formal specifications than it is possible in *Eiffel*, *JML* additionally builds on ideas from model-based specification languages like VDM-SL [40] (see also [7]) and the Larch family [32]. In particular, *JML* enables one to define abstract models of classes and objects by declaring model (and ghost) variables and methods. These variables and methods are not accessible by the actual unit code but can only be referred to within the *JML* annotations. Usage of these abstract models within the contract definitions leads to more abstract, more formal specifications.

Typically, a method's *JML* specification precedes the actual method declaration; class invariants precede the field declarations of a class. All *JML* specifications are *Java* comments which start with the at sign (@). Essentially, a method specification consists of a precondition and a postcondition. Preconditions are boolean predicates that must hold before the method is called; postconditions must hold after the execution of the method call. This means, the responsibility for establishing the precondition lies with the caller of the method – the responsibility for establishing the postcondition lies with the method itself. In *JML* preconditions and postconditions consist of the *JML* keyword *requires*, respectively, *ensures* followed by a boolean expression. Boolean expressions in *JML* are similar to normal *Java* boolean expressions. However, within a boolean expression one may only call methods that are declared as *pure* methods, i.e. methods that have no side effects. Moreover, *JML* provides additional construct which allow for more abstract specifications. For instance, one can quantify expressions by using *\forall* or *\exist*.

Listing 1.5 shows the voter unit code annotated with *JML* specifications. Line 5-6 formalize the required postcondition: method *conductVoting* may only yield `true` if, and only if, all voter objects likewise yield `true`. We have to assume, however, that *vote* is a pure method. Moreover, note that we cannot express the requirement that *conductVoting* must call the *vote* method of each voter object. Nevertheless, compared to *JUnit*, the example test specification is rather clear and concise. So the question may arise why *JUnit* is much more often in use than *JML*. One fact which might prevent *JML* from gaining more acceptance is its use of mathematical expressions. That is, although embedded into the *Java* code, the requirements are formalized in terms of mathematical formulas. Thus, despite the advantages of a more abstract specification, software developers seem to become reluctant, when likewise rather formal expressions come into play.

Listing 1.5: Voter example: *JMLUnit*

```
1  class Census {
2
3    /*@ public behavior
4      @ requires voters != null && voters.length() != null;
5      @ ensures \result == true <==>
6      @ (\forall int i; 0<= i && i < voters.length();
7                    voters[i].vote() == true)
8      @*/
9    public void conductVoting( Voters[] voters) {
10     result = true;
11
12     for (int i=0; i++, i <= voters.length()) {
13       result = result && voters[i].vote();
14     }
15   }
16 }
```

1.4 Testing approach in this thesis

This thesis proposes a novel approach for unit testing object-oriented components. The idea is to combine the benefits of the aforementioned existing testing frameworks. In particular, similar to the *JUnit* framework, the new approach should be accessible for software developers, it should allow for behavior-based testing like the *jMock* framework, and, finally, similar to the *JML* framework it shall allow for more abstract, hence, clear and concise, formalizations of the test cases where the underlying framework is based on a formal background.

However, we neither want to define an EDSL nor do we want to embed a formal language by means of annotations into the programming language. Instead, we embark on a *language extension* strategy. That is, we define a new *test specification language* by extending the original programming language of the production code with additional specification constructs. The intention of these tailor-made specification constructs it to provide the possibility for specifying a desired behavior of the unit under test in an abstract way. Moreover, in order to get an executable program that realizes the corresponding unit test, the test framework proposal comes with a *test code generation* algorithm that automatically generates programming language code from a specification of the test specification language. The testing approach is sketched in Figure 1.2.

The new specification constructs should not allow to specify aspects of the unit's behavior that has no impact on the unit's environment anyway. In other words, the constructs must aim at the *observable* behavior (cf. [49] and [53]) of the unit, only. To investigate the observable aspects of a unit's behavior in general, we provide a formally defined object-oriented programming language that is derived from *Java* and C♯. Specifically, the language will serve as the formal bedrock

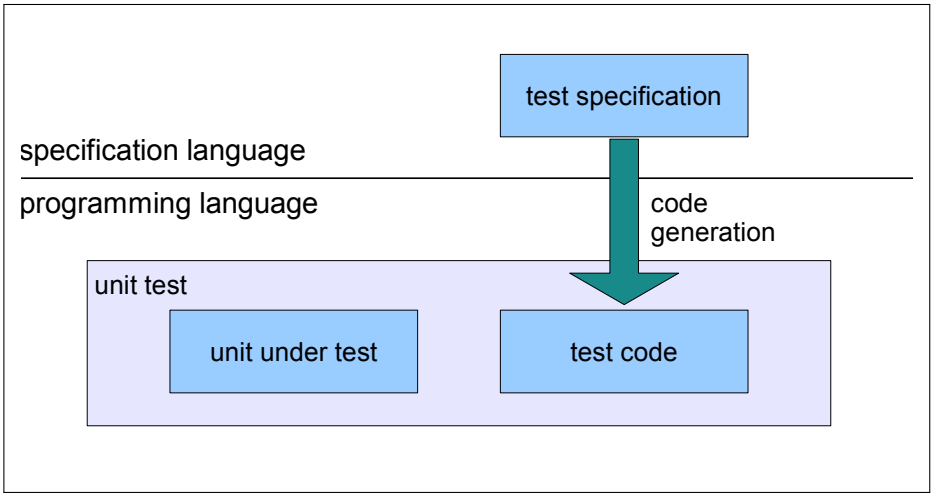

Figure 1.2: Novel testing approach

of our testing approach. For, apart from the features inspired from the above mentioned unit testing frameworks, we additionally want to support the unit testing of *concurrent* components which makes a formal context essential.

1.5 Structure of the thesis

The structure of this thesis is as follows. After this introductory chapter, the thesis consists of three parts. The first part deals with unit testing in context of a sequential object-oriented programming language. In particular, in Chapter 2 a formal definition for the *Java*-like object-oriented programming language *Japl* is developed. This is followed by the introduction of the test specification language for *Japl* in Chapter 3 and the code generation algorithm in Chapter 4. Finally, the first part concludes with the discussion about possible extensions of, both, the programming language and the test specification language.

The second part suggests a *concurrency* extension of the testing introduced in the first part. Specifically, Chapter 6 proposes an extension of the *Japl* programming language with *thread classes*. Correspondingly, Chapter 7 deals with an extension of the test specification language and, additionally, it sketches a suggestion on how to adapt the code generation algorithm of Chapter 4 in order to account for the concurrency extension. Finally Chapter 8 presents a conclusion of the thesis.

The third part of the thesis consists of the proofs. A central contribution of the thesis is the *correctness proof* of the code generation algorithm of Chapter 4. Although the *Java*-like language *Japl* covers only some basic aspects of typical object-oriented programming languages, still supported features like object-orientation and classes considerably increase the complexity of the proofs, already.

Thus, instead of embedding the proofs into the text they are presented separately in order to improve readability. In particular, it should be possible to understand and follow most of the ideas in this thesis without the need to understand all proofs in their details.

1.6 Relation to my previous scientific work

Many ideas of this thesis are based on or have been drawn from ideas related to my scientific work carried out and published during my Ph.D. studies.

In [2] a sequential class-based object calculus is introduced where programs consist of class definitions and a single thread definition. Considering components as sets of class definitions (and possibly a thread definition), the calculus serves as the mathematical vehicle for investigating the possible interaction *traces*, i.e., sequences of interactions that may take place between a component and its environment. The class-based setting makes instantiation a possible component-environment interaction which allows to create unconnected groups of objects, called *cliques*. Regarding a simple notion of *observability*, a notion of equivalence on these interaction traces is formalized which captures the uncertainty of observation caused by the fact that the observer may fall into separate cliques.

A similar class-based object calculus but additionally equipped with the support for *multi-threading* and re-entrant *monitors* has been proposed in [4]. The idea is to capture re-entrant monitor behavior, the basic synchronization and mutex-mechanism of, e.g., multi-threaded Java. A main result is that re-entrant monitors entail additional uncertainty of observation wrt. monitor operations at the interface which are captured by may- and must-approximations for potential, respectively, necessary lock ownership.

In [5] a class-based object calculus is introduced which allows dynamic thread instantiation by the support of *thread classes*. Similar to object instantiation, thread instantiation, occurring as a component-environment interaction, may lead to unconnected groups of objects which again increases the uncertainty of observation. The work formalizes a trace semantics with a notion of observable equivalence which accounts for the observational blur due to cliques.

In place of the thread-based concurrency model propagated by languages like *Java* and C♯, the work in [3] deals with object-oriented languages that introduces concurrency by means of asynchronous message passing. A corresponding object calculus is introduced capturing, furthermore, *futures* and *promises* which act as proxies for, or reference to, the delayed result from some piece of code. This allows to compare the concurrency model based on asynchronous message passing with the thread-based approach on a solid mathematical basis.

Based on the idea that the trace of interface interactions between a component and its environment may serve as a specification for the desired behavior of a component under test, in [20] an automated test driver generation for *Java* components is proposed. In particular, a specification language for specifying the desired behavior of a *Java* component is introduced. Moreover, the paper sketches an algorithm which allows to generate a *Java* test driver from such a specification.