Template Method Test Pattern

A. Bijlsma[a], H.J.M. Passier[a,*], H.J. Pootjes[a], S. Stuurman[a]

[a] *Faculty of Management, Science and Technology, Department of Computer Science, Open Universiteit, Valkenburgerweg 177, Heerlen, The Netherlands*

Abstract

How to test abstract classes is an area of testing that is not paid much attention to. Abstract classes cannot be instantiated and therefore standard execution-based test strategies are not applicable.

In this paper, we consider a special case of an abstract class, namely one produced by the Template Method pattern. This pattern results in an abstract class, with a concrete template method and one or more abstract primitive operations, and one or more subclasses, implementing the primitive operations specifically for each subclass.

How should we test an instance of the Template Method pattern? Testing the concrete template method by testing the abstract class is impossible. Testing an instance of the Template Method pattern by testing the template method in all of the subclasses individually is error-prone in the long run.

This paper presents a structured approach to test instances of the Template Method pattern in an elegant way using the Abstract Factory pattern. Furthermore, we introduce the new concept *semi-abstract method* to reason about concrete methods that depend on abstract operations. We formalize the pattern and demonstrate the approach at work using an example program.

Keywords: Software testing, Software design and implementation, Abstract classes, Template Method

2010 MSC: 00-01, 99-00

*Corresponding author

Email address: harrie.passier@ou.nl (H.J.M. Passier)

1. Introduction

The problem we consider in this article is that classes produced by the *Template Method Pattern* [1] cannot be tested in a straightforward way. This pattern is applicable if subclasses implement algorithms that contain similar steps in the same order, while the steps themselves are different. Applying the Template Method moves the algorithm structure and identical steps to a superclass and leaves the implementation of the different steps in the subclasses.

To illustrate the problem, we use a simple example throughout this paper. Listing 1 shows the situation before applying the Template Method pattern.

Listing 1: Example program

```
public class A {
  public String sayHello() {
    return "Hello";
  }
}

public class B extends A {
  public ArrayList<Integer> addSquares(ArrayList<Integer> base) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    for (int val : base) {
      res.add(val * val);
    }
    return res;
  }
}

public class C extends A {
    public ArrayList<Integer> addPosVals(ArrayList<Integer> base) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    for (int val : base) {
      if (val > 0) res.add(val);
    }
```

```
    return res;
  }
}
```

Superclass *A* has a method *sayHello* and has two subclasses *B* and *C*. Both subclasses inherit method *sayHello* and both have their own method (*addSquares* and *addPosVals*). The bodies of methods *addSquares* and *addPosVals* show similar steps in the same order.

Throughout this paper the word 'testing' refers to black-box testing, in other words, checking that the result of all operations conforms to their specification, regardless of implementation details.

Method *addSquares* of *B* returns an array containing the squares of the values in *base*, whereas method *addPosVals* of *C* returns an array containing only the positive values of *base*. In functional programming terms, they are a *map* and a *filter*, respectively. There exist several compact notations for such operations, e.g. the *Bird-Meertens formalism* [2].

Testing these classes is simple. We implement three test classes: test class *ATest* testing method *sayHello*, test class *BTest* testing method *addSquares* and test class *CTest* testing method *addPosVals*.

After application of the Template Method pattern, we get the program shown in listing 2. Figure 1 shows the corresponding class structure.

Listing 2: Template Method pattern applied

```
public abstract class A {
  public ArrayList<Integer> process(ArrayList<Integer> base) {
    ArrayList<Integer> res = new ArrayList<Integer>();
    for (int val : base) {
      use(res, val);
    }
    return res;
  }

  protected abstract void use(ArrayList<Integer> list, int element);
```

```
65
    public String sayHello() {
      return "Hello";
    }
  }
70
  public class B extends A {
    public void use(ArrayList<Integer> list, int element) {
      list.add(element * element);
    }
75  }
  public class C extends A {
     public void use(ArrayList<Integer> list, int element) {
      if (element > 0) list.add(element);
    }
80  }
```

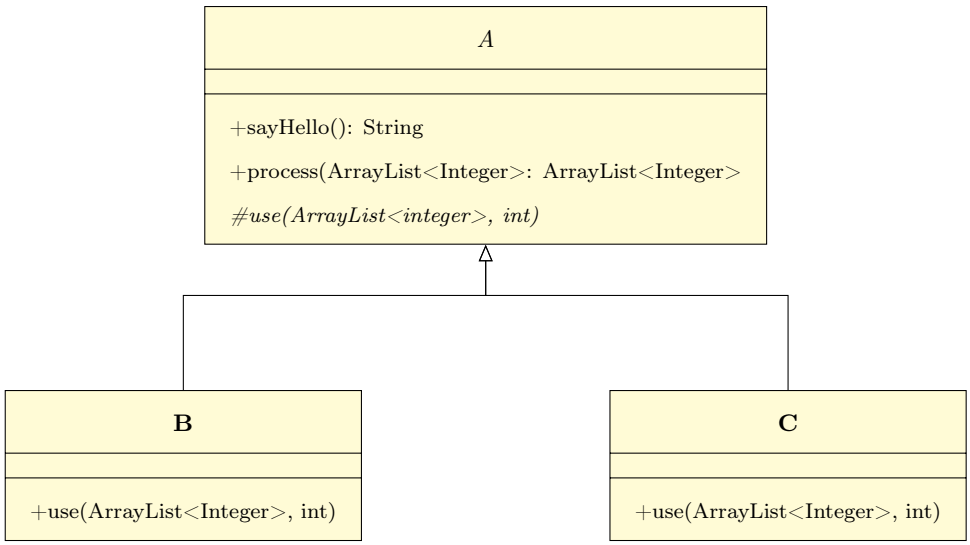

Figure 1: The class structure after refactoring

The generic structure of the *Template Method* pattern consists of an abstract

class and one or more concrete subclasses [1, page 360]. The *abstract class* (*A* in our example) defines a template method (*process*) that defines the skeleton of that algorithm. The template method makes use of other abstract operations(*use*) in order to run the algorithm. Concrete subclasses implement these operations.

1.1. Problem

The question we consider is: 'How to test an instance of the Template Method pattern in an elegant and safe way?'.

When we refactor to this pattern, as above, we should, in theory, create tests for all new concrete methods [3]. In a Template Method pattern, this is not possible: although the *process* method is concrete, its body refers to abstract methods. Therefore, we cannot test the template method (*process*) by unit testing its class (*A*).

Remember that (here) testing means checking that the result conforms to its specification. Method *process* has no fixed postcondition because its workings depend on the choices of *use*, which are undetermined. If we would include white-box properties, for instance a demand that *process* should call *use* on each of the elements of *base*, these could be tested by means of an appropriate stub, but that is a different problem.

Testing an instance of the Template Method pattern by testing one subclass is not enough, because the expected return value of *process* differs for each subclass. Concrete methods that depend on abstract methods should be tested by testing all subclasses that implement these abstract methods individually [4]. However, that is an undesirable approach: each time we add a subclass, we have to remind ourselves to test this concrete method of superclass *A* too, which is error-prone in the long run: because the template method in the abstract class is concrete, it is easy to forget testing this method in subclasses.

1.2. Contributions

First, we show how an instance of the Template Method pattern can be tested in an elegant and safe way using the Abstract Factory pattern [1] resulting in what

we call a *Three Parallel Architecture for Class Testing* (3PACT). Secondly, we introduce the concept *semi-abstract method* to reason about concrete methods depending on abstract operations. This concept helps in recognizing when the test pattern is applicable and in describing the structure of the class hierarchies.

2. The Template Method Test Pattern

The abstract class in the Template Method pattern implements a (concrete) template method which uses some (abstract) primitive operations. Subclasses implement these primitive operations to carry out subclass-specific steps. Given an object instance, the concrete template method calls the concrete primitive operations of one specific subclass.

2.1. Semi-abstract method

To be able to reason about this situation, we introduce the concept of a semi-abstract method: A *semi-abstract* method is a concrete method that depends, directly or indirectly, on one or more abstract operations, or calls a method of an abstract class defined in a class hierarchy elsewhere.

The Template Method pattern uses a specific form of a semi-abstract method: the concrete method in the abstract class depends on one or more abstract methods in the abstract class, which must be implemented in the concrete subclasses. In contrast, most occasions of semi-abstractedness in code will concern method bodies that refer to an object through an interface or abstract class, because objects should be known, preferably, by their interface or abstract class, and not by their concrete class.

2.2. The test pattern

Testing an instance of the Template Method pattern implies testing the template method as well as instances of the abstract operations. Because the functionality of the template method is determined by the specific subclass chosen, we have to test all object instances individually.

Creating an object of a specific subclass can be straightforward (like in our example), but it might also involve creating other objects that should be given as parameters to the constructor. It is undesirable, therefore, to have the test class create these objects. That would be a responsibility that does not belong to the test class.

We propose, therefore, to use the Abstract Factory pattern to create instances of the subclasses to be tested, and to create a hierarchy of test classes mirroring the hierarchy of the classes forming the Template pattern, see Figure 2.

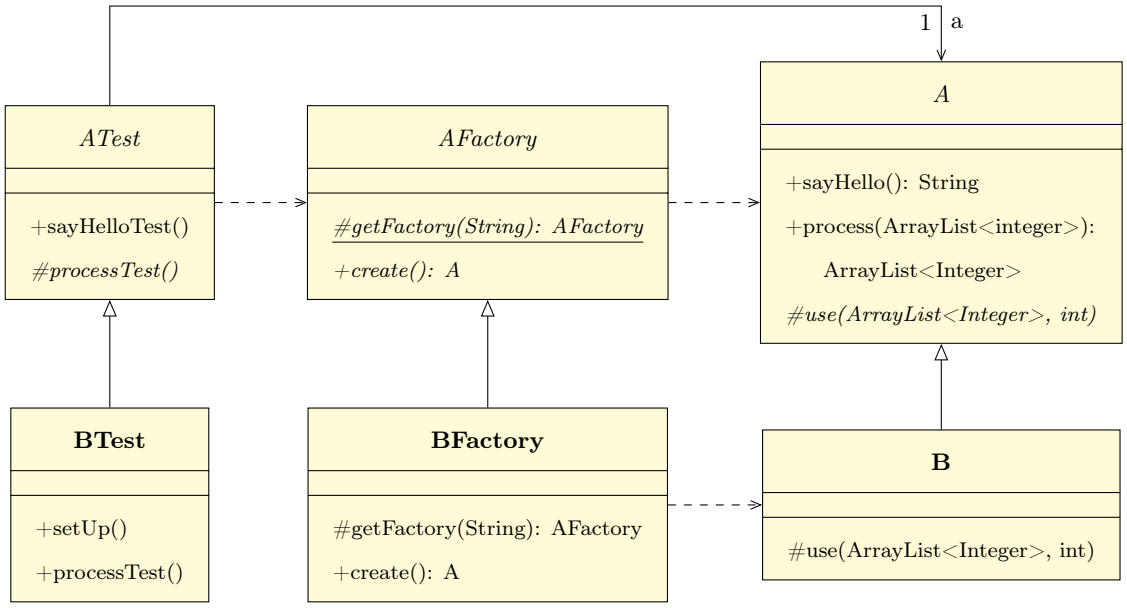

Figure 2: The 3 Parallel Architecture of Class Testing (3PAC)

Listing 3 shows the abstract factory. The abstract class *AFactory* declares an abstract operation *create* which returns an instance of a concrete subclass part of the class hierarchy under class *A*. The abstract class *AFactory* also has a class method *getFactory* that returns an object of class *AFactory*, based on a string that is passed to it by the calling object. When this string is `"B"`, this method returns an object of class *BFactory*. Class *AFactory* defers the creation of objects to one of its concrete factories, in this case *BFactory*. Each concrete factory redefines this create operation to create objects of the concrete class under test, in our case objects of type *A*, of class *B*.

Listing 3: The factory hierachy

```
public abstract class AFactory {
```

```
  static AFactory getFactory(String testClass) {
    AFactory factory = null;
    //based on the value of testClass, determine adequate factory
    return factory;
  }
  public abstract A create();
}

public class BFactory extends AFactory{
  public A create(){
    return new B();
  }
}
```

The client of the factory (a test class) only knows the interface declared by the abstract factory and the name of the class it should test. It receives a concrete factory, that it can use to call the operation *create*, to get an instance of the class under test. In our example, creating an instance of the class under test would be simple, but in many cases, creating such an object is more complicated: often, the constructor should be called with parameters, and the values of these parameters are sometimes objects that should be created as well. By using the Abstract Factory method, we relieve the test classes of the responsibility to create these objects.

Class *ATest* (see listing 4) has an attribute *a* of type *A*. An instance of the class under test is assigned to this attribute by calling method *create* of the factory that is received by calling *getFactory* of class *AFactory*. Because method *sayHello* in class *A* is a concrete method, class *ATest* contains a concrete test for it. Notice the abstract definition of method *processTest*; this enforces a redefinition in *all* subclasses.

Class *BTest* extends abstract class *ATest*. In this class, we define a constant *TESTCLASS* with the name of the class under test. This can be used in the call to *getFactory*, to receive a factory that will provide the right value for attribute

a. Method *setup* prepares the test data according to the specific functionality of method *use*. Method *processTest* is redefined compulsory for doing the right test.

Listing 4: The test class hierarchy

```
public abstract class ATest {
  protected A a = null;

  @Test
  public void sayHelloTest() {
    String s = a.sayHello();
    assertEquals("Hello", s);
    }
  public abstract void processTest();
}

public class BTest extends ATest {
   private ArrayList<Integer> testlist = new ArrayList<>();
   private ArrayList<Integer> resultlist = new ArrayList<>();
   static final String TESTCLASS = "B";

   public BTest() {
     super();
     a = AFactory.getFactory(TESTCLASS).create();
  }

  @Before
  public void setUp() throws Exception {
    testlist.add(1); testlist.add(2); testlist.add(-3);
    resultlist.add(1); resultlist.add(4); resultlist.add(9);
  }
  @Test
  public void processTest() {
     assertEquals(resultlist, a.process(testlist));
```

```
  }
}
```

2.3. Discussion

Our pattern solves the problem that we have to remember to test the (concrete) template method of the abstract class in the Template Method pattern (in our example *process*). We simply need a concrete subclass of *ATest* for each concrete subclass of *A*. By delegating the responsibility to create objects of the classes to be tested to an abstract factory, the test classes are not cluttered with the creation of objects.

Thanks to the interface of class *ATest*, all tests can be run in a uniform way. Furthermore, by defining an abstract method *processTest* in class *ATest* we *enforce* that each subclass defines its own test for the method *process*. Now, we only have to remember to create a test class and a factory for each subclass of *A*. Our pattern ensures that the concrete method *process* is tested in each concrete subclass (thereby implicitly testing each *use* method).

The architecture presented, forms a so-called Parallel Architecture of Class Testing (PACT) [5]. In our case, we even have three parallel hierarchies: the test hierarchy, the factory hierarchy and the domain class hierarchy. We will call our hierarchy a 3 Parallel Architecture of Class Testing (3PACT). Notice that a concrete method in abstract class *A* (in this case *sayHello*) is tested by a semi-abstract test method (in this case *sayHelloTest*) in class *ATest*. Method *sayHelloTest* is semi-abstract because it depends on attribute *a* which value is determined by the abstract method *getFactory*. A semi-abstract method in class *A* (in this case, *process*) is tested by an abstract test method (in this case *processTest*) in class *ATest*. In other words, the methods of the abstract class under test are tested by methods that are one step more abstract.

This pattern is not specific to the example, but can be applied to any abstract class. A black-box unit test for an abstract class, should contain

- no test for an abstract method,

- an abstract test for a semi-abstract method,
- a semi-abstract test for a concrete method.

One might consider the hierarchy in 3PACT as overcomplicated. The code could be simplified by merging the functionality of the factory classes into the test classes resulting in a create method in each of the test classes. Although, this simplifies the code in cases the factory classes are simple, it has two important drawbacks. First, it violates the OO principle 'Separate the use of an object from its creation'. Secondly, mixing responsibilities results in low cohesion and as such increases the code's complexity and decreases the code's readability.

3. Related work

Testing abstract classes is not paid much attention to in the literature [4]. The inability to instantiate objects of an abstract class is mentioned as a reason, thereby preventing them from being executed at runtime [4].

Clarke et al. [4] gives general advice about the minimal set of test cases that should be used, for instance, that in the case of a concrete method that references abstract methods, there should be a test case for each class that implements such an abstract method. No attention is given to how one ensures that the right tests are developed.

Thuy [6] gives general, execution-based, approaches for testing the features of an abstract class. However, these approaches are not suitable for testing an instance of the Template Method pattern, because they pass over the fact that *each* concrete definition of a template method depends on the abstract class' template method as well as the overridden primitive operations in each subclass.

Kong and Ying [5] describe a method for testing abstract classes. Their testing approach makes use of the parallel architecture of class testing (PACT) [7] and uses a Factory Method design. They do not recognize the problem of concrete methods that refer to abstract methods, but their approach is usable for those instances. In their approach, test classes are responsible for the creation of objects of the right class; in our approach, test classes are uncluttered. Moreover, the approach of Kong and Ying ignores the problem of concrete methods that refer to abstract methods.

4. Conclusions

Almost always, software is developed iteratively, using development methods such as the Unified Process (UP) [8] or eXtreme Programming (XP) [9]. An important principle of practice of these methods is that code is refactored continuously [10]. As such, refactoring plays an important part in modern software development methods.

An essential precondition for refactoring code is to have solid tests in advance [11]. A unit test tests the external behavior of code, which should be unchanged after a refactoring has been performed.

However, refactoring code can break the API and hence the unit test. This means that after a refactoring, the unit test may have to be refactored as well. If the refactoring introduces a semi-abstract method, as is the case in the Template Method pattern, the solution described in this article should be applied.

Moreover, the Template Method pattern is often used in frameworks. Every application that uses such a framework can only be completely tested by taking into account the semi-abstract methods involved. Forcing subclass authors to do so would require framework authors to publish their unit tests, incorporating our pattern.

4.1. Future work

This pattern solves the problem that is caused by semi-abstractness within the Template Method pattern: in those cases in which a concrete method depends on abstract methods that are implemented in subclasses. A similar problem arises in semi-abstract methods where the body refers to abstract methods of other classes (for instance, in the Strategy pattern or the Bridge pattern). Determining how (adaptations of) this pattern can be useful in those circumstances is subject of future research.

In theory, it is possible to automatically detect semi-abstractedness in the form that we see in the Template pattern. Because the pattern that we described has a specific form that can be partially constructed from the classes under

315 test, it should be possible to automatically generate test classes and factory classes. The developer should implement the *setUp* and *processTest* methods of the concrete test subclasses, and the *create* method of the concrete factory classes. Everything else could be generated automatically. Creating such a tool is subject of future research.

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, Reading, MA, USA, 1995.

[2] J. Gibbons, An Introduction to the Bird-Meertens Formalism, in: S. Reeves (Ed.), Proceedings of the First New Zealand Formal Program Development Colloquium, Hamilton, 1994, pp. 1–12.

[3] H. Passier, L. Bijlsma, C. Bockisch, Maintaining unit tests during refactoring, in: Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ '16, ACM, New York, NY, USA, 2016, pp. 18:1–18:6. doi:10.1145/2972206.2972223.
URL http://doi.acm.org/10.1145/2972206.2972223

[4] P. J. Clarke, J. F. Power, D. Babich, T. M. King, A testing strategy for abstract classes, Software Testing, Verification and Reliability 22 (3) (2012) 147–169. doi:10.1002/stvr.429.
URL http://dx.doi.org/10.1002/stvr.429

[5] L. Kong, Z. Yin, The extension of the unit testing tool junit for special testings, in: Proceedings of First International Multi-Symposium on Computer and Computational Sciences, vol. 2, IMSCCS'06, IEEE Press, 2006, pp. 410âĂŞ–415.

[6] N. Thuy, Testability and unit tests in large object-oriented software, in: Fifth International Software Quality Week, Software Research Institute, 1992, pp. 1–9.

[7] J. McGregor, D. Sykes, A Practical Guide to Testing Object-Oriented Software, Addison-Wesley, 2001.

[8] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[9] K. Beck, Embracing change with extreme programming, Computer 32 (1999) 70–77. doi:doi.ieeecomputersociety.org/10.1109/2.796139.

[10] I. Sommerville, Software Engineering, 9th Edition, Addison-Wesley, Harlow, England, 2010.

[11] M. Fowler, S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, C. Poole, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.