A Pattern to Test Instances of the Template Method

A. Bijlsma∗, H.J. Pootjes† and H.J.M. Passier‡

1Open Universiteit, Faculty of Management, Science and Technology, Department of Computer Science, Postbus 2960, 6401 DL Heerlen, The Netherlands

July 21, 2017

∗lex.bijlsma@ou.nl
†harold.pootjes@ou.nl
‡harrie.passier@ou.nl
1 Introduction

Context. Refactoring is the process of improving the structure of software without altering its external behavior [3]. The goal of refactoring is to improve the design of existing code for better understandability and ease of maintenance and evolution.

Software systems can become very complex and developing such a system in one go is mostly unfeasible. Instead, software is often developed in an iterative and incremental way using agile software development methods such as the Unified Process (UP) [7] or eXtreme Programming (XP) [1]. An important principle of practice of these methods is that code is refactored continuously as soon as possible [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 [3]. For Java we use JUnit as a testing framework. A unit test tests the external behavior of code, which should be unchanged after a refactoring has been performed. But refactoring can break the API and hence the unit test. For example, renaming a public method requires that the corresponding test method is renamed in the same way and calls the new named method. As a result, after a refactoring is applied the alignment between the structure of source code and corresponding unit tests can be broken. Maintaining consistency between refactored program code and unit tests is an unsolved problem [9].

The refactoring considered in this report is the Form Template Method [3, page 345], based on the Template Method Pattern [4]. This refactoring is applicable if subclasses implement algorithms that contain similar steps in the same order, yet the steps are different. Applying the Template Method moves the algorithm structure and identical steps to a super class and leaves the implementation of the different steps in the subclasses. The simple program of listing 1 will be used for applying the Form Template Method in our report. Figure 1 shows the class structure of this program.

Listing 1: Example program

```java
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);
        }
    }
}
```
Superclass \texttt{A} has a method \texttt{sayHello} and has two subclasses \texttt{B} and \texttt{C}. Both subclasses inherit method \texttt{sayHello} and have an own method \texttt{process}. In class \texttt{B} method \texttt{addSquares} squares the values in list \texttt{base}, whereas method \texttt{addPosVals} in class \texttt{C} filters the positive values in list \texttt{base}. Methods \texttt{addSquares} and \texttt{addPosVals} have similar steps in the same order.

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

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

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{class_diagram.png}
\caption{The class structure of the example program}
\end{figure}

\begin{lstlisting}[language=Java]
Figure 1: The class structure of the example program

public abstract class A {
    public ArrayList<Integer> addPosVal(ArrayList<Integer> base) {
        ArrayList<Integer> res = new ArrayList<Integer>();
        for (int val : base) {
            if (val > 0) res.add(val);
        }
        return res;
    }
}

Listing 2: Template Method pattern applied

public abstract class A {
    // Method implementation
}
\end{lstlisting}
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);

public String sayHello() {
    return "Hello";
}

public class B extends A {
    protected void use(ArrayList<Integer> list, int element) {
        list.add(element * element);
    }
}

public class C extends A {
    protected void use(ArrayList<Integer> list, int element) {
        if (element > 0) list.add(element);
    }
}

Figure 2: The class structure after refactoring

Notice, that method process is a concrete method and depends on the abstract method use which is implemented in both subclasses of A.

The generic structure of the Template Method pattern consists of [4, page 360] an abstract class and one or more concrete subclasses. The abstract class
A in our example) defines abstract primitive operations (use) that concrete sub-
classes define to implement steps of an algorithm. Furthermore, the abstract
class implements a template method (process) that defines the skeleton of an
algorithm. This template method calls the primitive operations as well as pos-
sible other operations defined in the abstract class or those of other objects. The
concrete classes (B and C in our example) implement the primitive operations
to carry out subclass-specific steps of the algorithm.

**Problem.** The question we consider is: ‘How to test an instance of the Tem-
plate Method pattern in an elegant way?’. We cannot test the template method (process) by unit testing its class (A). Although the template method (process) in itself is a concrete method, its class (A) is abstract because of the abstract primitive operations (use) part of it.

Method process in class A can be tested by testing all of its subclasses in-
dividually. But that is an undesirable approach: each time we add a subclass,
we have to remember ourselves to test the concrete functionality of superclass
A too, which is error-prone in the long run.

In research literature, some solutions are described for testing features of
abstract classes. Clarke et al. [2] summarized some approaches for testing an ab-
stract class. Kong and Yin [6] described an approach using the Factory Method
pattern. All these approaches, however, do not solve the specific problem part
of the Template Method pattern, i.e. testing a concrete template method in
an abstract class whose functionality depends on one or more abstract methods
redefined in each of the subclasses of the abstract class. In section 3 we will
discuss the work of Clarke et al. and Kong and Yin in more detail.

**Contributions.** First we show how an instance of the Template Method pat-
tern, i.e. an abstract class containing a concrete template method depending on
one or more abstract primitive operations, can be tested in an elegant way using
the Abstract Factory pattern [4]. Applying this pattern results in three class
hierarchies showing identical hierarchical structures, which might be called, in
an analogy to the terminology of Kong and Yin [6], a 3PACT structure. Second,
y, we introduce a new concept semi-abstract to annotate concrete methods
depending on abstract operations. The concept helps in recognizing when the
test pattern is applicable and describing the structure of the class hierarchies.

**This report.** In section 2, we introduce the concept semi-abstractness and
describe and discuss the test pattern. In section 3, we describe related work.
Finally, in section 4 we draw conclusions. In this report, we assume the pro-
gramming language Java as is described by Gosling et al. [5] and JUnit version
4.10.
2  A pattern for testing an instance of the Template Method

The abstract class in the Template Method pattern implements a (concrete) template method which uses some (abstract) primitive operations. Subclasses implements 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-abstractness

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

2.2  The test pattern

Testing an instance of the Template Method 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.

By applying the Abstract Factory Pattern [4], we are able to create an object composed of the abstract class, defining the concrete template method, and a particular subclass, defining specific implementations of the primitive operations. Following our example in the introduction (Listing 1), we now describe the use of this pattern where we focus on superclass A and subclass B.

The abstract factory is showed in listing 3. The abstract class AFactor declared an abstract operation create which returns an instance of a concrete subclass part of the class hierarchy under class A. Class AFactor 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 composed of class A and subclass B.

Listing 3: The factory hierarchy

```java
public abstract class AFactory{
    public abstract A create();
}

public class BFactory extends AFactory{
    public A create(){
        return new B();
    }
}
```
The client of the factory knows only the interface declared by the abstract factory and receives, after instantiating a concrete factory and calling operation create, an instance of the class under test.

Class \textit{A}\textit{Test} (see listing 4) owns attribute \textit{a} of type \textit{A}. An instance of the class under test is assigned by calling method \textit{getFactory} which is redefined in all subclasses. On the returned object, the \textit{create} method is invoked. Because method \textit{sayHello} in class \textit{A} is a concrete method, class \textit{A}\textit{Test} contains a concrete test for it. Notice the abstract definition of method \textit{processTest} as well as method \textit{getFactory}; both enforce a redefinition of these methods in all of subclasses.

Class \textit{B}\textit{Test} extends abstract class \textit{A}\textit{Test}. Method \textit{getFactory} is redefined compulsory and instantiates the right factory instance which creates an object under test. Method \textit{setup} prepares the test data according to the specific functionality of method \textit{use}. Method \textit{processTest} is redefined compulsory for doing the right test.

\begin{verbatim}
Listing 4: The test class hierarchy

public abstract class ATest {
    protected A a = getFactory().create();

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

    public abstract void processTest();
}

public class BTest extends ATest {
    private ArrayList<Integer> testlist = new ArrayList<>();
    private ArrayList<Integer> resultlist = new ArrayList<>();

    protected AFactory getFactory() {
        return new BFactory();
    }

    @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(a.process(testlist), resultlist);
    }
}
\end{verbatim}
2.3 Discussion

Our pattern solves the problem mentioned in the introduction. By distributing the create methods over the concrete factory subclasses, a particular subclass (for instance $B$), with a specific implementation of primitive operations (method $\text{use}$), is combined with abstract class $A$ implementing the template method ($\text{process}$). Thanks to the interfaces of class $A\text{Test}$, all tests can be run in a uniform way. Furthermore, by defining an abstract method $\text{process}Test$ in class $A\text{Test}$ we enforce that each subclass defines its own test for method $\text{process}$.

The architecture forms a so-called Parallel Architecture of Class Testing (PACT), mentioned in [6]. 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). Figure 3 shows the architecture in UML class diagram notation.

![Diagram](attachment:image.png)

Figure 3: The 3 Parallel Architecture of Class Testing (3PACT)

Notice the symmetry in degree of abstractness between the related methods in this structure: The concrete method $\text{sayHello}$ in class $A$ is tested by a semi-abstract test method $\text{sayHelloTest}$ in class $A\text{Test}$. Method $\text{sayHelloTest}$ is semi-abstract because it depends on attribute $a$ which value is determined by the abstract method $\text{getFactory}$. The semi-abstract method $\text{process}$ in class $A$ is tested by the abstract test method $\text{process}Test$ in class $A\text{Test}$.

The problem of testing semi-abstract methods could be solved by means of an annotation, as for example $@\text{semiAbstract}$, informing the programmer like: ‘Caution, this is an (semi-)abstract method and should be tested by means of all of its subclasses’. Our approach offers a systematic solution that forces a programmer to define a test in each (new) subclass for this type of methods.
3 Related work

Testing abstract classes is not paid much attention to in the literature [2]. We have found very little literature about this topic too. The inability to instantiate objects of an abstract class is mentioned as a reason, thereby preventing them from being executed at runtime [2]. About testing the Template Method, we have not found any literature at all.

Clarke et al. [2] summarize three general, execution-based, approaches for testing the features of an abstract class which are mentioned in the literature: (1) defining a concrete class solely for the purpose of testing the features of the abstract class, (2) testing the features of an abstract class as part of testing the first concrete descendant, and (3) testing a minimal set of concrete descendants that do not redefine the methods in the abstract parent class. Thuy [11] presents three rules for testing abstract classes: (1) deferred methods must be overridden in a concrete class and can be tested in that class, (2) an inherited method can be tested in the framework of a concrete derived class that does not override it, and (3) an inherited method in a concrete class that calls an extended method in a derived class must be tested in the derived class. These approaches mentioned by Clarke et al. and Thuy are not suitable for testing an instance of the Template Method pattern, because these approaches 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 [6] described a method for testing abstract classes. Their testing approach makes use of the parallel architecture of class testing (PACT) [8] and uses a factory design model. For an abstract class $A$ and concrete descendant $C$, two tester classes are created in order to implement the PACT architecture. The $A\text{Tester}$ class, which is also abstract, contains methods to test the concrete features of $A$. The $A\text{Tester}$ class must declare two types of abstract methods compulsorily. The first type is a factory method which will return an instance of a concrete subclass inheriting the abstract class under test. The second type is a method that will return the anticipated behavior value of the concrete subclass. The $C\text{Tester}$ class inherits from the $A\text{Tester}$ class and implements the abstract methods from $A\text{Tester}$. The $C\text{Tester}$ implementation is then used to perform testing on $A\text{Tester}$, thereby testing $A$. Differences with our method are that: (1) they use a factory design model resulting in a parallel architecture, where we apply the Abstract Factory pattern explicitly resulting in a structure of three parallel class hierarchies, (2) our approach is specialized for abstract classes owning a semi-concrete method, i.e. instances of the Template Method pattern.
4 Conclusions

We presented a systematic test pattern for testing instances of the Template Method pattern, resulting in a three Parallel Architecture of Class Testing (3PACT). The pattern makes use of the Abstract Factory pattern. To recognize the specific situation to which the test pattern is applicable, we introduced the concept semi-abstractness.
References


