Testing of chosen Design Patterns with JUnit and Mockito

Niccolò Fabbri
Francesco Santoni

Università degli Studi di Firenze
Master of Science in Information Engineering

24 August 2016

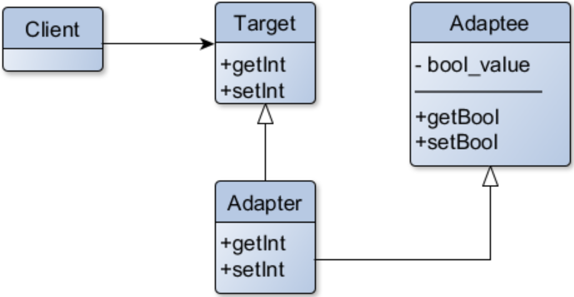
Instructor: Prof. Enrico Vicario

Introduction

- We have identified a collection of **structural** and **behavioral design patterns**: Class Adapter, Object Adapter, Proxy, Decorator, Composite, Observer, State, Visitor.
- For each pattern we realize an implementation in Java and we develop a **reasoned test suite** based on a realistic **fault model** and on chosen **coverage criteria**.
- We realize the tests through the *JUnit* plug-in for Eclipse and the *Mockito* framework. *EclEmma* is used to provide a code coverage measure(bytecode branch coverage).

Class Adapter

Adapts a pre-existent class to a new interface through inheritance.
Through the new interface the old methods can be directly presented, modified or produce aggregated results.



Class Adapter - Fault Model

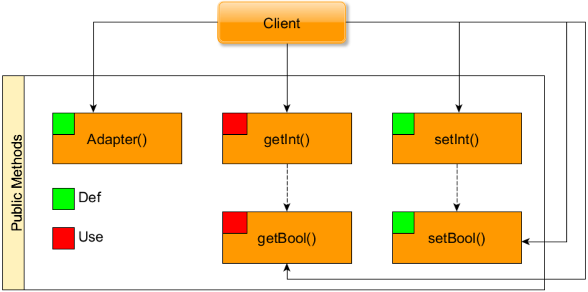
The pattern focuses on allowing access to legacy methods through a new interface.

- The adapter did not inherit from the legacy class or the new interface
- The adapter cannot interact with the legacy methods

Solutions

- Test the ways in which the variable *bool_value* interacts and is modified by the methods

Class Adapter - Data Flow Graph

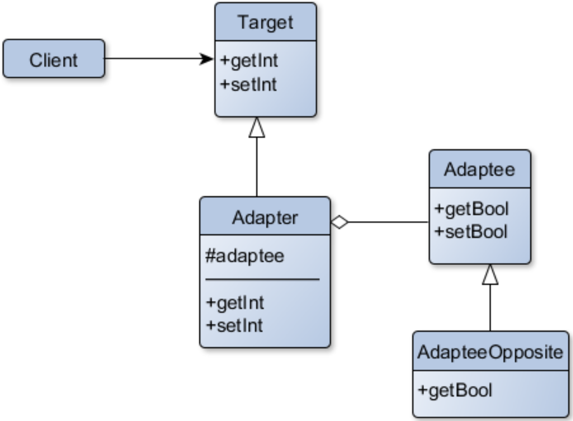


Almost all the ***all-uses* paths**:

- Adapter() getInt()
- Adapter() getBool()
- Adapter() setInt() getInt()
- Adapter() setInt() getBool()
- Adapter() setBool() getInt()

Object Adapter

Adapts a pre-existent class to a new interface through class composition. Through the new interface the old methods can be directly presented, modified or produce aggregated results.



Object Adapter - Fault Model

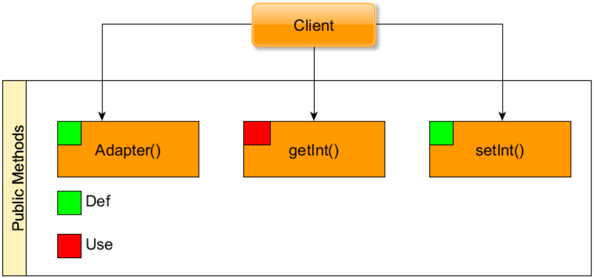
The pattern focuses on allowing access to legacy methods through a new interface.

- The adapter cannot interact with the legacy methods
- The instance contained in the adapter, which inherited the adaptee class, has overrode its methods in an unforeseen way

Solutions

- Test the ways in which the variable *adaptee* interacts with the methods, considering all the possible alternative implementations of the overrode methods.

Object Adapter - Data Flow Graph



All the ***all-uses*** **paths**:

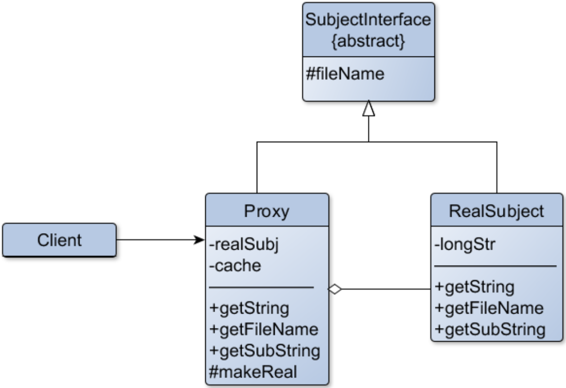
- Adapter() getInt()
- Adapter() setInt() getInt()

To these are also added the overridden method' tests:

- Adapter(Adaptee) getInt()
- Adapter(AdapteeOpposite) getInt()

Proxy

The Proxy pattern is constituted by a class functioning as an interface to something else, usually a complex or heavy object. It is used to access the real serving object behind the scenes, it either **provides a cached result or transmits the request to the actual object**



Proxy - Fault Model

The pattern focuses on optimizing or controlling the access to the heavy subject.

- The access to the RealSubject is impeded
- The cached copies provided by the Proxy differ from the actual source

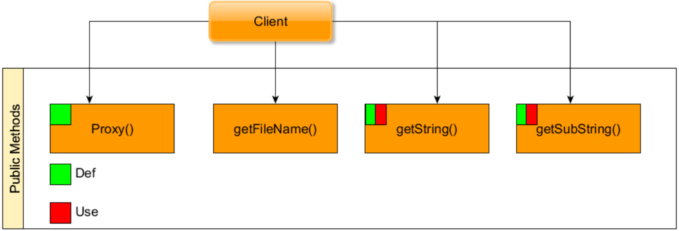
Solutions

- Test the ways in which the variable *realSubj* interacts and is modified by the methods

  $\rightarrow$ by slight modification of the tests we can automatically verify the cached versions validity

.

Proxy - Data Flow Graph

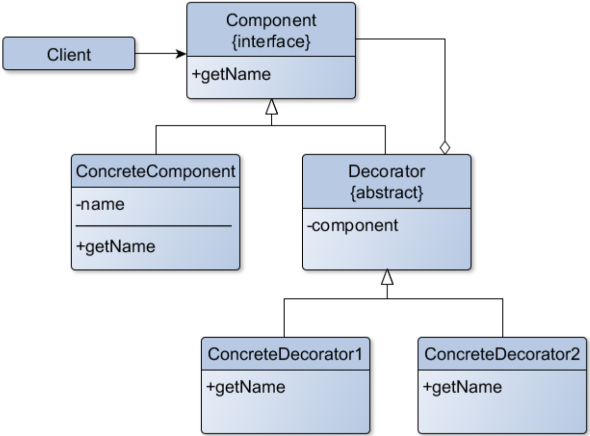


All the ***all-uses*** **paths**:

- Proxy() getString()
- Proxy() getString() getString()
- Proxy() getSubString()
- Proxy() getString() getSubString()
- Proxy() getSubString() getSubString()
- Proxy() getSubString() getString()

Decorator

The Decorator pattern **allows behavior to be added to an individual object**, either statically or dynamically, without affecting the behavior of other objects from the same class.



Decorator - Fault Model

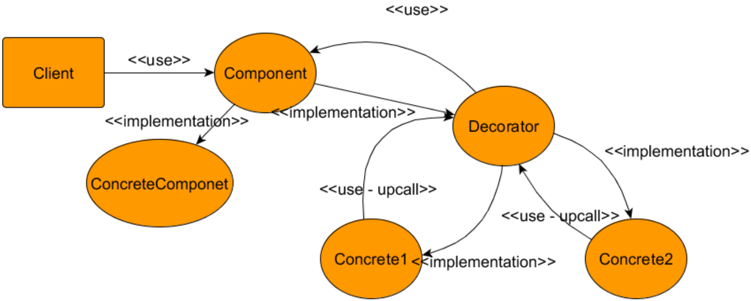
The pattern focuses on allowing an extension of functionality in objects.

- The call to the *operation (getName)* does not reach the Component or results in unexpected behavior

Solutions

- Test the correctness of the sequence of method calls in different hierarchies of classes

Decorator - Class Dependency Graph

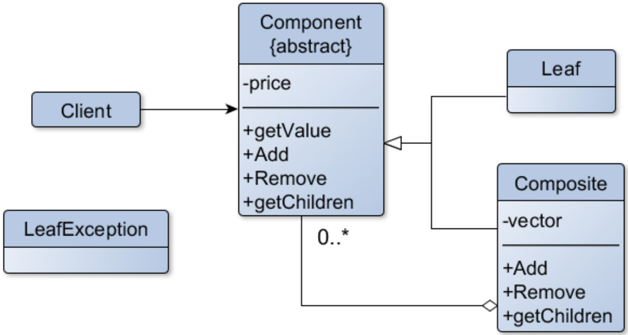


All-edges criterion: 3 cases

- ConcreteComponent
- ConcreteDecorator1 ConcreteComponent
- ConcreteDecorator1 ConcreteDecorator2 ConcreteComponent

Composite

The Composite pattern "composes" objects into tree structures to **represent part-whole hierarchies**. Implementing the composite pattern lets clients **treat individual objects and compositions uniformly**.



Composite - Fault Model

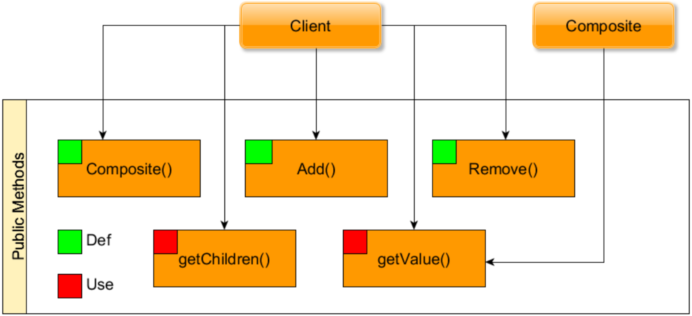
The pattern focuses on treating uniformly individual and compound objects.

- The common *operation (getValue)* works differently than expected
- The composite-specific methods produce unexpected effects

Solutions

- Test the way *operation* works under the possible hierarchies at runtime
- Test the way the different objects(*Leaf* and *Composite*) behave under calls from composite-specific methods

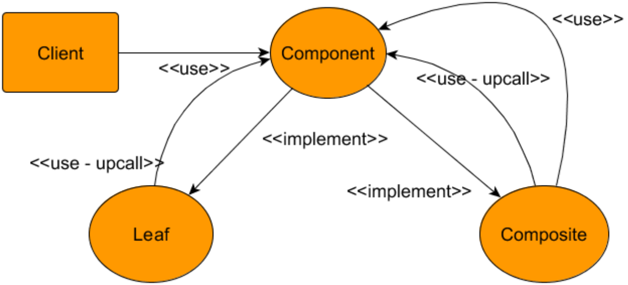
Composite - Data Flow Graph



Most ***all-uses*** **paths**:

- Component() getChild()
- Component() getValue()
- Component() add() getChild()
- Component() add() getValue()
- Component() add() add() remove() getChild()
- Component() add() add() remove() getValue()

Composite - Class Dependency Graph

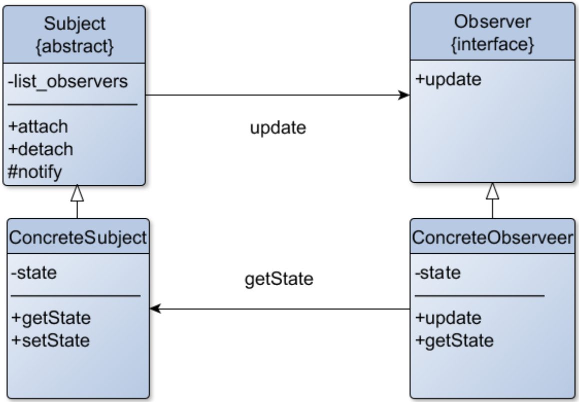


***All Edges* criterion**: 3 cases

- single Leaf
- Composite containing Leaf
- Composite containing Composite

Observer

In the Observer pattern an object, called the subject, maintains a list of its dependents, called observers, and **notifies them automatically of any state changes**, usually by calling one of their methods.



Observer - Fault Model

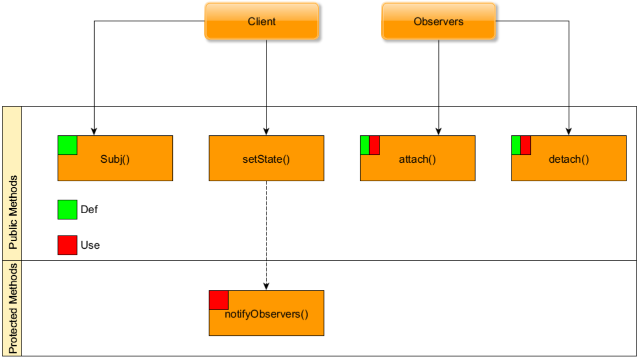
The pattern focuses on maintaining updated objects that expressed the interest in a specific subject.

- *Attach* and *detach* do not produce the expected results
- After a change of the subject state the observers are not *notified*
- The observer after being notified does not execute correctly the *update* method

Solutions

- Test the way *list_observers* is modified after an inter-class method invocation
- Test the way the *state* of the observer is modified after a notification

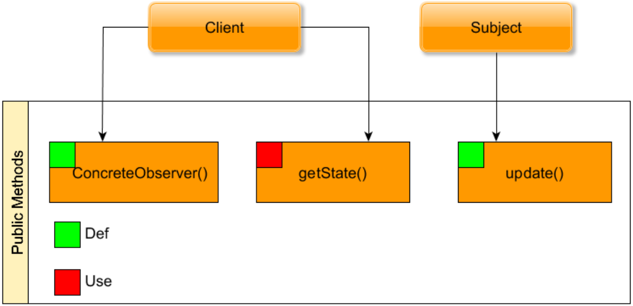
Observer - Data Flow Graph: field *list_observer*



All the ***all-uses* paths**:

- Subject() setState()[notify()]
- Subject() detach()
- Subject() attach()x3 detach()x2
- Subject() attach()x2 detach()x2 attach() detach() attach()

Observer - Data Flow Graph: field *state*

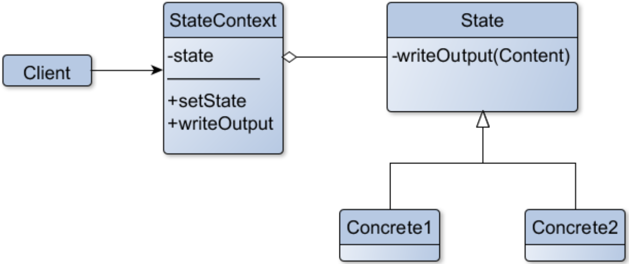


All the ***all-uses*** **paths**:

- ConcreteObserver() getState()
- ConcreteObserver() update() getState()

State

The State pattern implements a **state machine** by implementing **each individual state as a derived class** of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass.



State - Fault Model

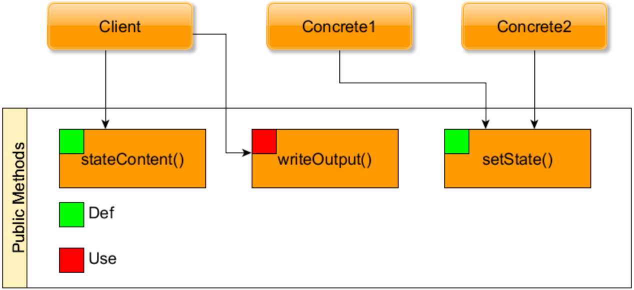
The pattern focuses on delegating the actual methods implementation to internal state classes.

- The context state changes in an erroneous manner
- The internal state's methods produce unexpected side effects or results

Solutions

- Test the way the *state* field interacts with the StateContext methods

State - Data Flow Graph

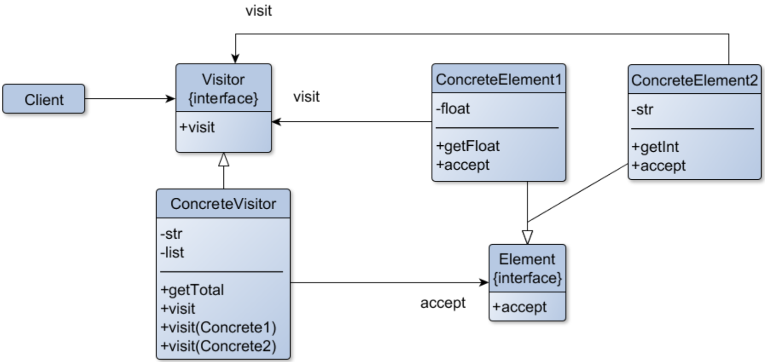


All the ***all-uses*** **paths**:

- StateContext() writeOutput()
- StateContext() setState() writeOutput()
- StateContext() writeOutput() writeOutput()

Visitor

The Visitor pattern is a way of **separating an algorithm from an object structure on which it operates**.
The pattern allows one to add new virtual functions to a family of classes without modifying the classes themselves.



Visitor - Fault Model

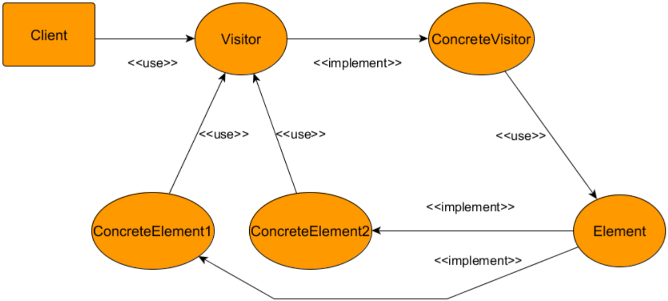
The pattern focuses on treating uniformly objects of different types while operating on them with different specializations of the same function.

- The wrong *visit()* is applied to an Element

Solutions

- Test the way *visit* works when applied to all possible hierarchies of Element types

Visitor - Class Dependency Graph



***All edges* criterion**: 2 cases

- Visitor ConcreteVisitor Element ConcreteElement2 Visitor
- Visitor ConcreteVisitor Element ConcreteElement1 Visitor

JUnit

JUnit is an open source **unit** (and integration) **testing framework** for the **Java** programming language. The framework allows the programmer to easily create *drivers* for the tests and the ability to verify the produced outputs.

- ***Annotation***s identify the test methods
- ***Assertion***s compare the produced output to expected results and produce a verdict on the test's outcome.

Main benefits of the framework are:

- JUnit tests can be run automatically, check their own results and provide immediate feedback without a need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.

Mockito

- Mockito is an open source testing framework for Java. The framework allows the **creation of test double objects** also called **mock objects**.
  - → Mock testing frameworks allow the faking of external dependencies so that the object being tested is **isolated from external behaviors**
  - → Ensuring that objects perform the way they are expected to would require the creation of tests that actually exercise each behavior and verify that it performs as expected. *With costs comparable to implementing the external dependencies*

Mockito

While utilizing the framework we identified some noteworthy details:

- in the Proxy class we utilized *spy* **on the very class we were testing** to allow injection of other mocked classes.
- in the Visitor class we returned an Answer() construct to produce side-effects when the mocked object's methods were called.
- Answer is not necessary if the side-effects are produced only on the very class under test due to the fact that one can independently produce them by simply calling the respective tested methods.

Conclusions

- We identified a collection of structural and behavioral design patterns. For each pattern, we:
  - produced an implementation
  - identified the most error-prone points
  - created a reasoned test suite based on the fault model and on a coverage criteria chosen pattern by pattern
- We realized both Unit and Integration tests through the JUnit plug - in for Eclipse and the Mockito framework.
  - → applied EclEmma to provide a code coverage measure(bytecode branch cov.f)
- We identified both the untested branches and the reason for which they were not covered.
- In the end most patterns presented a full code coverage.