Università degli studi di Firenze

SCUOLA DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

Testing of chosen Design Patterns with JUnit and Mockito

Docente
Prof. Enrico Vicario

Relazione a cura di
Niccolo' Fabbri
Francesco Santoni

Firenze,
24 Agosto 2016

Contents

Chapter 1

Introduction

We identify 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[1] plug-in for Eclipse and the Mockito[**?**] framework. EclEmma [**?**] is used to provide a code coverage measure.

Chapter 2

Design Patterns

In software engineering, a software design pattern is a general reusable template to solve a commonly occurring problem within a given context.

We will study:

1. Structural patterns: Adapter(both in his Class and Object variants), Proxy, Decorator, Composite.
2. Behavioural patterns: Observer, State, Visitor.

2.1 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, produce aggregated results or completely new functionality can be added.

Class Diagram

- **Adaptee**: legacy class with specific methods and fields
- **Target**: desired interface
- **ClassAdapter**: inherits from both Adaptee and Target, adapts the legacy methods to the desired interface

Fault Model

Given that the pattern focuses on allowing access to legacy methods through a new interface, failures are found in the following situations:

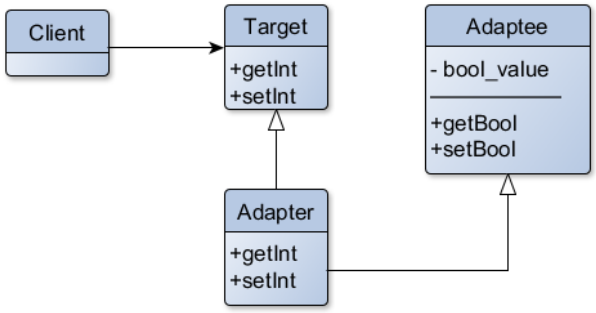

Figure 2.1: Class Adapter: Class Diagram

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

2.1.1 Testing

The two sources of failure both depend on the inability of the Client to reach the Adaptee methods through the Adapter. We reasoned that it is thus sufficient to test the ways in which the variable *bool_value* interacts and is modified by the methods.

As long as the variable changes in an unexpected way the connection between Client and Adaptee is in fact cut off.
We have thus represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

We then decided to test the interaction between the variable and the methods following the *all-uses* criterion, we in fact deemed the *all-def* criterion not useful to test this pattern because we value mainly the transmission of the right value of the field and thus all its *uses*.

Data Flow Graph The Client is the only external class interacting with the public methods of the Adapter. The Data Flow Graph can be seen in Figure 2.2.

Tests

We generated a test suite capable of testing all the *all-uses* paths:

- Adapter() getInt()

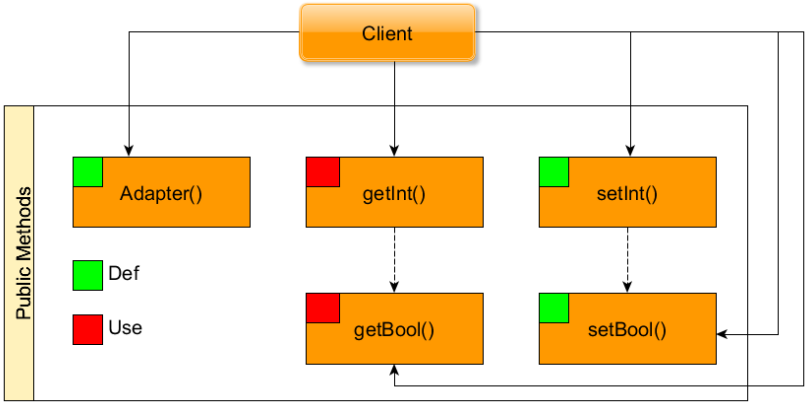

Figure 2.2: Data Flow Graph: *bool_value*

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

The special case of

- Adapter() setBool() getBool()

is not tested because all the methods are directly inherited from the legacy class(and thus are supposedly already tested) and no side-effects, which could have invalidated some invariants, are introduced in the adapter.

In this particular pattern the presence of a lone object other than the Client makes Unit and Integration tests un-distinguishable.

Code Coverage

The code coverage measure obtained from EclEmma Java plug-in is: 100%.

2.2 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, produce aggregated results or completely new functionality can be added.

The composition adds the possibility of dynamically switching the adapted legacy class (not contemplated in our implementation) and the possibility of overridden methods with altered functionality.

If the adaptee class is declared final, the increased complexity from overridden methods is eliminated but the extra functionality is removed as well.

Class Diagram

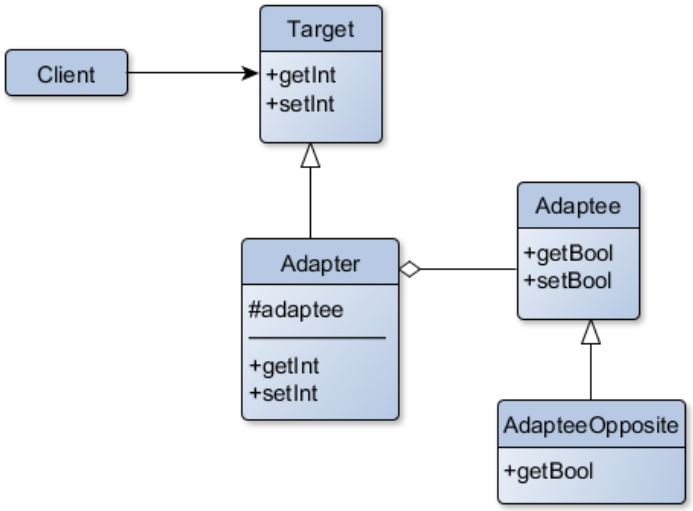

Figure 2.3: Object Adapter: Class Diagram

- **Adaptee**: legacy class with specific methods and fields
- **Target**: desired interface
- **ObjectAdapter**: inherits from Target, adapts the legacy methods to the desired interface through delegation.

Fault Model

Given that the pattern focuses on allowing access to legacy methods through a new interface, failures are found in the following situations:

- the adapter did not inherit from the legacy class or the new interface
- the adapter for some reason 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

2.2.1 Testing

The sources of failure depend on the inability of the Client to reach the Adaptee methods through the Adapter or in the inability of the Adapter in foreseeing the possible ways in which the Adaptee methods can be overridden: we reasoned that it is sufficient to test the ways in which the variable *bool_value* interacts and is modified by the methods, considering all the possible alternative implementations. As long as the variable changes in an unexpected way the connection between Client and Adaptee is in fact cut off.

The problem of testing thus expands in two different orthogonal dimensions: data flow, the series of calls that modify or use the variable *bool_value*, and topology, the different ways the classes are ordered in a hierarchy at runtime.

We decided to explore both of them separately in specific focused tests.

Data Flow: field *adaptee* We have represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

Data Flow Graph Like the ClassAdapter, only the Client interacts with the Adapter methods. The Data Flow Graph can be seen in Figure 2.4.

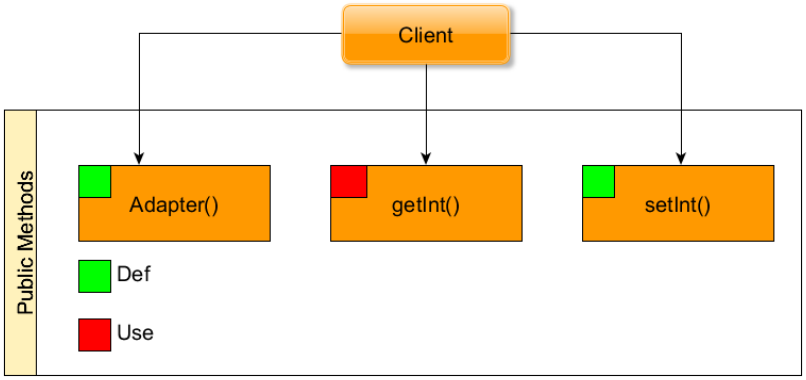

Figure 2.4: Data Flow Graph: *bool_value*

We then decided to test the interaction between the variable and the methods following the *all-uses* criterion, we in fact deemed the *all-def* crite-

rion not useful to test this pattern because we value mainly the transmission of the right value of the field and thus all its *uses*.

Overridden methods We have tested separately the overridden variants of the methods through simple tests.

We reasoned that such a separation would allow us to sufficiently cover the code while maintaining a low number of tests.

Tests

We generated a test suite capable of testing all the *all-uses* paths:

1. Adapter() getInt()
2. Adapter() setInt() getInt()

To these are also added the topology tests:

1. Adapter(Adaptee) getInt()
2. Adapter(AdapteeOpposite) getInt()

Of these the first, being already tested in the first suite, is not repeated.

Since in this pattern the tested class's methods (Adapter's) interacted with external classes (Adaptee) we differentiated between Unit and Integration tests.

We utilized Mockito's *mock* function to isolate errors with an origin in external classes from interfering with the Adapter class's code.

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 98.6%

- Adapter: 97.1% (UnitTest: 97.1%)
- Adaptee: 100%
- AdapteeOpposite: 100%

The single remaining untested branch is a setter method which was not called with all the possible inputs equivalence classes.

Since the Unit Tests and the Integration Tests are identical, with the only difference that the Unit Tests utilize Mockito's help to isolate from external classes, the coverage of the Adapter class's code is the same.

2.3 Proxy

The Proxy pattern is constituted by a class functioning as an interface to something else, usually a complex or heavy object.

It is called by the client to access the real serving object behind the scenes, it either provides a cached result or transmits the request to the actual object.

Class Diagram

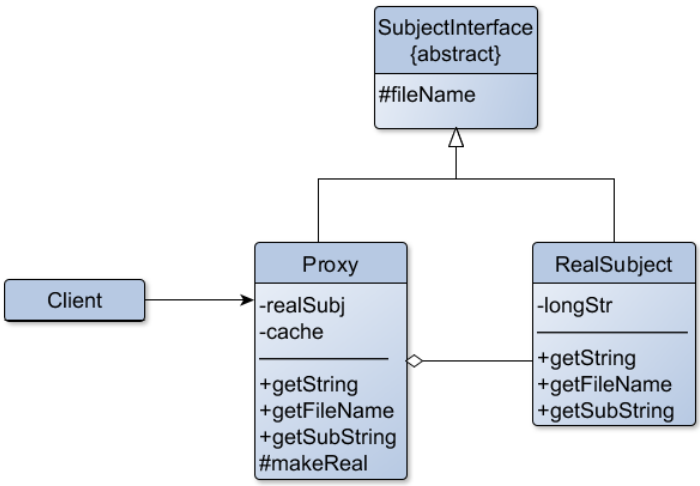

Figure 2.5: Proxy: Class Diagram

- **SubjectInterface**: defines the interface used by the Client to access the RealSubject.
- **Proxy**: allows access to the RealSubject, either by providing a cached response or by delegating the requests to the RealSubject itself.
- **RealSubject**: defines the real complex or heavy object wrapped by the proxy.

Fault Model

The pattern focuses on optimizing or controlling the access to the heavy subject. We have failures in the following situations:

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

2.3.1 Testing

The sources of failure depend on the inability of the Client to reach the RealSubject fields through the Proxy or in the inability of the Proxy of providing correct cached versions: we reasoned that it is sufficient to test the ways in which the variable *realSubj* interacts and is modified by the methods, by slight modification of the tests we can automatically verify the cached versions validity.

Data Flow: field *price* We have represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

Data Flow Graph Only the Client interacts with the Proxy methods. The Data Flow Graph can be seen in Figure 2.6.

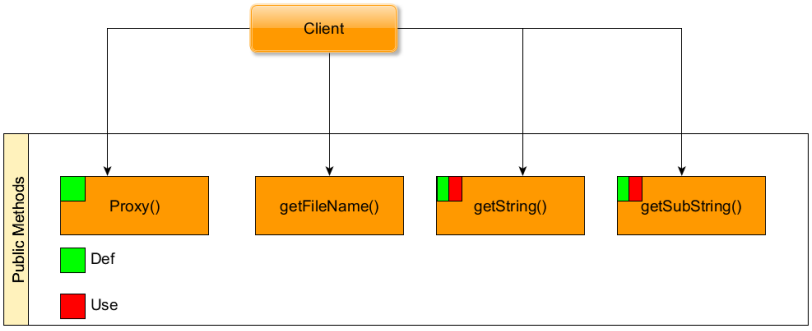

Figure 2.6: Data Flow Graph: *realSubj*

We decided to test the interaction between the variable and the methods following the *all-uses* criterion, we in fact desired to verify through the tests not only the correct interactions of the *realSubj* field with the methods but also the *cache*'s, we determined that a more extensive coverage could thus be desired.

Tests

Data Flow: field *price* We generated a test suite capable of testing all the *all-uses* paths:

1. Proxy() getString()
2. Proxy() getString() getString()

3. Proxy() getSubString()
4. Proxy() getString() getSubString()
5. Proxy() getSubString() getSubString()
6. Proxy() getSubString() getString()

In particular *Proxy() getSubString() getSubString()* was altered to test the returned cached version after each method call.

Since in this pattern the tested class's methods (Proxy's) interacted with external classes (RealSubject) we differentiated between Unit and Integration tests.

The Proxy class, by having to provide a cached version or to instantiate a RealSubject which will process the heavy data and return it to the Proxy, poses problems to the injection of a mocked RealSubject class.

Two possibilities were considered: create a RealSubjectFactory to pass during the construction of the Proxy and utilize it during the instantiation of the inner RealSubject or separate the responsibility of instantiation in a method void of logic.

In the first possibility one would have to use Mockito's *mock* method to mock the Factory so that it would return a mocked RealSubject, while in the second case one would have to use Mockito's *spy* method to modify the Proxy itself so that the logic-less method would return a mocked RealSubject.

Due to not having to create an unnecessary extra class we preferred and thus implemented the second option.

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 93.8%

- Proxy: 95.4% (UnitTest 95.4%)
- RealSubject: 88%
- SubjectInterface: 100%

The remaining untested branches correspond to the *getFilename()* methods, which where not tested due to absence of logic.

Since the Unit Tests and the Integration Tests are identical, with the only difference that the Unit Tests utilize Mockito's help to isolate from external classes, the coverage of the Proxy class's code is the same.

2.4 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.

Class Diagram

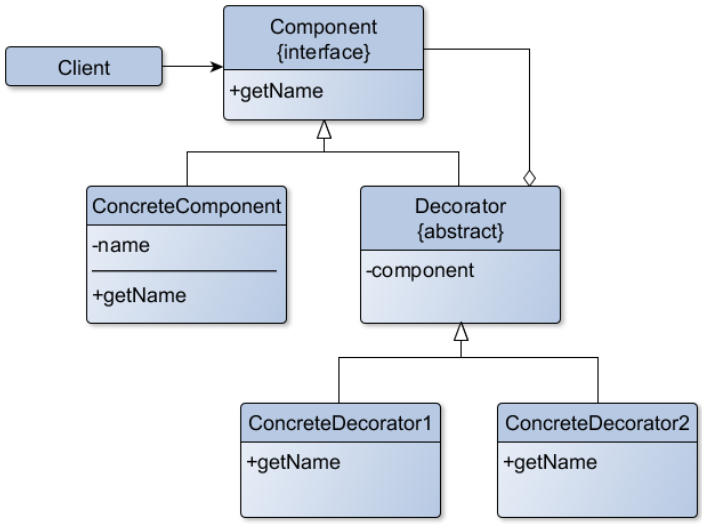

Figure 2.7: Decorator: Class Diagram

- **Component**: common interface to the concrete component and its wrappers.
- **ConcreteComponent**: implements the object to which responsibilities can be added.
- **Decorator**: interface to concrete additive responsibilities, maintains a reference to a Component.
- **ConcreteDecorator**: implements an additive responsibility.

Fault Model

The pattern focuses on allowing an extension of functionality in objects. Grave errors are found in the following situations:

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

2.4.1 Testing

The sources of failure depend on the presence of errors in the sequence of additive behavior to the *operation (getName)*. We reasoned that to produce a satisfying test suite it is sufficient to test the correctness of the sequence of method calls in different hierarchies of classes.

Topology: method *operation* We generated a Class Dependency Graph to identify all the possible relations between classes.

Class Dependency Graph In the Figure 2.8 are made explicit the relations between classes: in particular since Decorator is an abstract class it and its inheriting classes depend on each other.

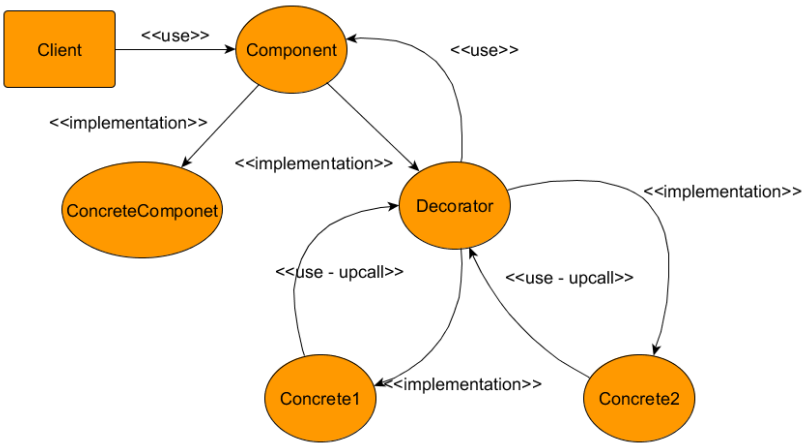

Figure 2.8: Class Dependency Graph: *getName()*

We decided to generate a test suite dependent on the *all-edges* criterion.

Tests

testo topologia, potrei fare con un unica singola copertura degli edges, decido di fare "pi simile a all uses" es. se ConcComp lo vedo come use lo devo raggiungere dai pi modi possibili ConcreteComp DecoratorComp DecorDecorComp

non si testa su null poich quello un errore di chi ha creato (factory) e gli si delega tale responsabilit

Topology: method *operation* We identified the 3 cases of:

- ConcreteComponent
- Decorator ConcreteComponent
- Decorator Decorator ConcreteComponent

as representative of the entire set of possible hierarchies.

We then tested the *getName()* method on the identified cases.

Since the tested class's methods (ConcreteComponent and Decorator) interacted with external classes (an other Component) we differentiated between Unit and Integration tests.

We utilized Mockito's *mock* function to isolate errors with an origin in external classes from interfering with the main class's code.

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 100% (Unit: 100%)

Since the Unit Tests and the Integration Tests are identical, with the only difference that the Unit Tests utilize Mockito's help to isolate from external classes, the coverage of the classes' code is the same.

2.5 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.

Class Diagram

- **Component**: common interface to the individual and compound objects.
- **Composite**: composition of individual objects and other compositions.
- **Leaf**: individual object.
- **LeafException**: exception thrown when a composite-specific method is called on a Leaf object.

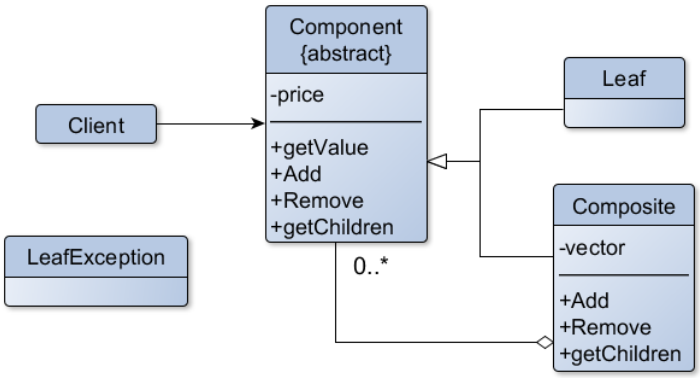

Figure 2.9: Composite: Class Diagram

Fault Model

The pattern focuses on treating uniformly individual and compound objects. Grave errors are found in the following situations:

- the common *operation* works differently than expected
- the composite-specific methods produce unexpected effects when called on a Leaf object

2.5.1 Testing

The sources of failure depend on the inability of the specific components (individual or composite parts) to be interacted in an uniform way. We reasoned that to produce a satisfying test suite we must test two different things: the way *operation* works under the possible hierarchies at runtime and the way the different objects behave under calls from composite-specific methods.

As long as outcomes different from the expectation are observed we can reasonably predict that no uniform treatments of the components is in place.

The problem of testing thus expands in two different orthogonal dimensions: data flow, the series of calls that modify or use the variable*price*, and topology, the different hierarchies of objects with respect to the method *operation*.

We decided to explore both of them separately in specific focused tests.

Data Flow: field *price* We have represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

Data Flow Graph Both Client and other wrapping Composites interacts with the Component methods. The Data Flow Graph can be seen in Figure 2.10.

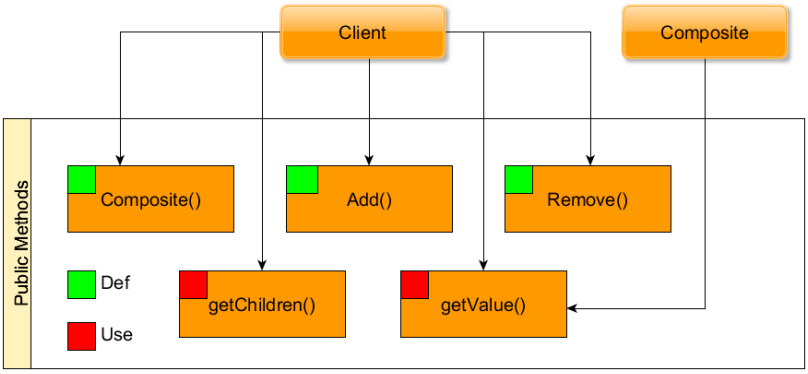

Figure 2.10: Data Flow Graph: *price*

We then decided to test the interaction between the variable and the methods following the *all-uses* criterion, we deemed the *all-def* criterion excessively restrictive to test this pattern.

The field was tested in the fixed hierarchy formed with a Composite element containing another Composite, which contained a Leaf. We reasoned that the chosen fixed hierarchy, while not allowing to test on each possible instance, is sufficiently complete, maintaining a balance between number of tests and efficiency.

Topology: method *operation* We generated a Class Dependency Graph to identify all the possible relations between classes.

Class Dependency Graph In the Figure 2.11 are made explicit the relations between classes: in particular since Component is an abstract class it and its inheriting classes depend on each other.

We decided to generate a test suite dependent on the *all-edges* criterion. *all-nodes* is in fact excessively relaxed and would not cover all cases of interests.

Tests

Data Flow: field *price* We generated a test suite capable of testing all the *all-uses* paths:

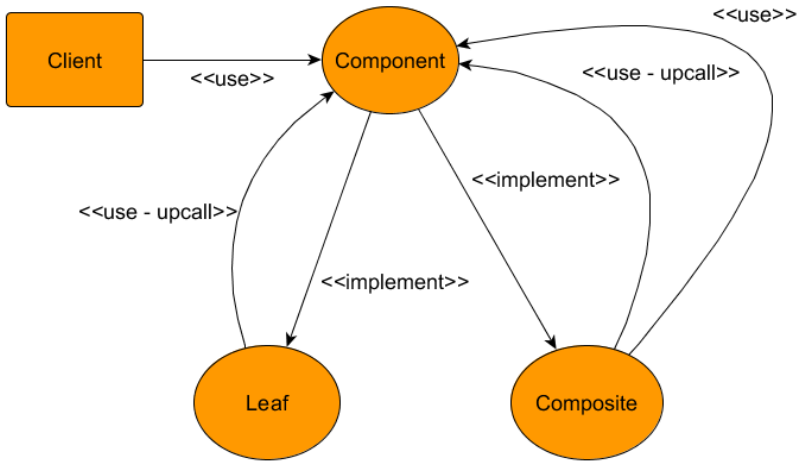

Figure 2.11: Class Dependency Graph: *operation()*

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

Topology: method *operation* We identified the 3 cases of:

- single Leaf
- Composite containing Leaf
- Composite containing Composite

as representative of the entire set of possible hierarchies.

We then tested all the composite-specific methods (add, remove, getChild) on the identified cases.

In both type of tests, since the tested class's methods (Component) interacted with external classes (Leaf or Composites) we differentiated between Unit and Integration tests.

We utilized Mockito's *mock* function to isolate errors with an origin in external classes from interfering with the main class's code.

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 97.8%

- Component: 92.3% (UnitTest 92.3%)
- Composite: 100%
- Leaf: 100%
- LeafException 100%

The remaining untested branches are found in the implementation of the composite-specific methods that Component provides for Leaf: all other classes (Composite) directly override those methods so their implementation when they are called not by a Leaf is never tested.

Since the Unit Tests and the Integration Tests are identical, with the only difference that the Unit Tests utilize Mockito's help to isolate from external classes, the coverage of the classes' code is the same.

2.6 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.

The observers must be explicitly associated and disassociated with the subjects they follow, possibly causing memory leaks.

Class Diagram

- **Subject**: provides methods to add or remove observers in his list of followers.
- **Observer**: specifies an interface for the notification of change on a subject.
- **ConcreteSubject**: contains a state that can be modified.
- **ConcreteObserver**: implements the update operation.

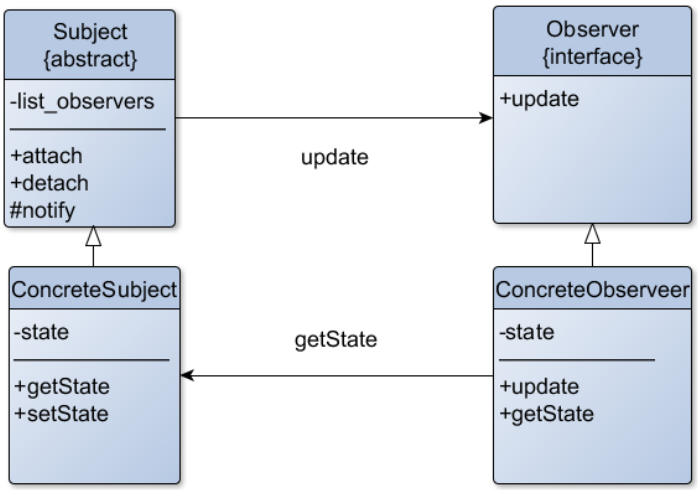

Figure 2.12: Observer: Class Diagram

Fault Model

The pattern focuses on maintaining updated objects that expressed the interest in a specific subject. Failures are found in the following situations:

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

2.6.1 Testing

The sources of failure depend on the presence of problems that impede the correct functioning of the inter-class methods and, in the lesser part, by the inability of the observer to correctly update. We considered the correct modification on the subject internal *state* as not important for our pattern. We reasoned that to produce a satisfying test suite we must test the way *list_observers* is modified after an inter-class method invocation and the way the *state* of the observer is modified after a notification.

As long as the outcomes differ from the expectation we can reasonably predict that the transmission of information upon notification encountered problems.

Data Flow: field *list_observers* We have represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

Data Flow Graph Only the Client interacts with the Subject methods, in particular the responsibility of attaching or detaching Observers from Subjects lies with the Client. The Data Flow Graph can be seen in Figure 2.13.

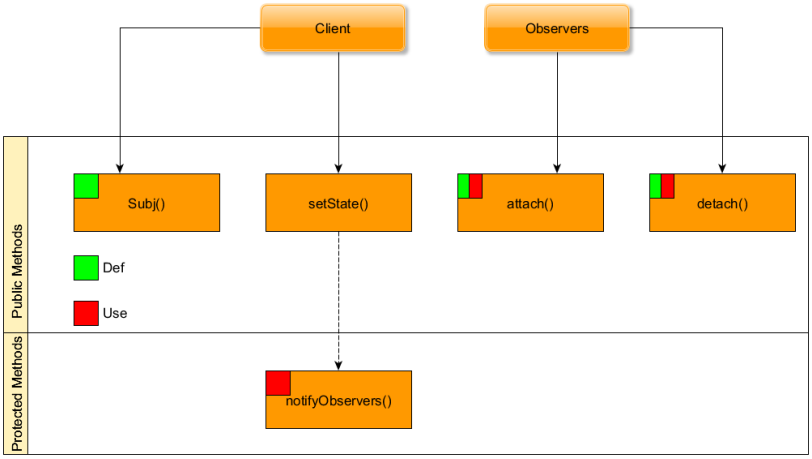

Figure 2.13: Data Flow Graph: *list_observers*

We decided to test the interaction between the variable and the methods following the *all-uses* criterion.

Data Flow: field *state* We have represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

Data Flow Graph Both Client and followed Subjects interacts with the Component methods. The Data Flow Graph can be seen in Figure 2.14.

We decided to test the interaction between the variable and the methods following the *all-uses* criterion, we deemed the *all-def* criterion excessively restrictive to test this pattern.

Tests

Data Flow: field *list_observer* We decided to test as separate and distinct cases for situations like *detach() detach()* where the methods are called

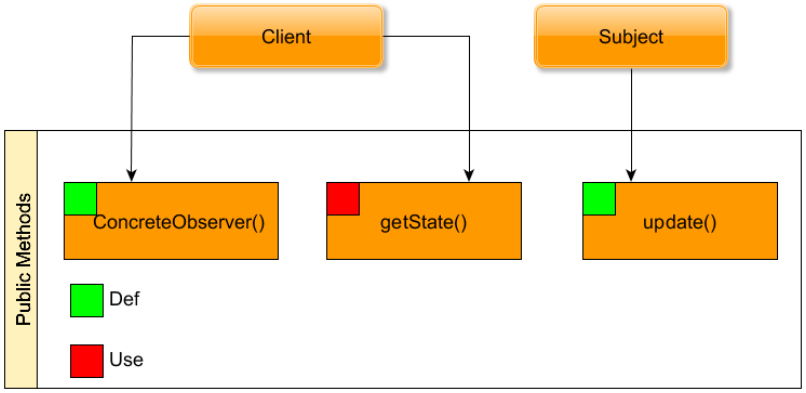

Figure 2.14: Data Flow Graph: *state*

with the same Observer or different ones. We generated a test suite capable of testing all the *all-uses* paths:

1. Subject() setState()[notify()]
2. Subject() detach()
3. Subject() attach()x3 detach()x2 (+setState()[notify()])
4. Subject() attach()x2 detach()x2 attach() detach() attach() (+setState()[notify()])

All these paths are interesting to study:

1. controls no errors are introduced if a *setState()* is called with empty list
2. a detach is called with empty list

3,4 by interspersing verifications on the *list_observer* we test all combinations of the different D-U paths:

   (a) attach() attach()
   (b) attach() detach()
   (c) detach() detach()
   (d) detach() notify()
   (e) Subject() attach()
   (f) detach() attach()
   (g) attach() notify()

To create an output of the *list_observer* we utilize a support method void of logic, which we decided did not need to be considered as an *use* during the tests.

Data Flow: field *state* We generated a test suite capable of testing all the *all-uses* paths:

1. ConcreteObserver() getState()
2. ConcreteObserver() update() getState()

In both type of tests, since the tested class's methods (ConcreteSubject and ConcreteObserver) interacted with external classes (Observer and Subject) we differentiated between Unit and Integration tests.

We utilized Mockito's *mock* function to isolate errors with an origin in external classes from interfering with the main class's code. To

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 100% (Unit: 100%)

2.7 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.

Class Diagram

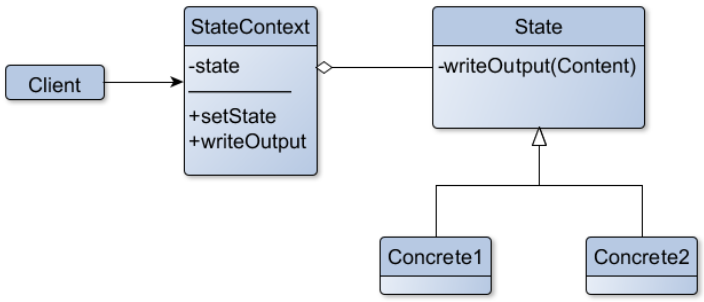

Figure 2.15: State: Class Diagram

- **StateContext**: maintains an instance of State to serve as its inner state.
- **State**: interface of classes with the specific behavior desired by StateContext.
- **ConcreteState**: implements StateContext's functions, possibly dependent on an inner state.

Fault Model

The pattern focuses on delegating the actual methods implementation to internal state classes. Grave errors are found in the following situations:

- internal state changes in an erroneous manner.
- the internal state methods produce unexpected side effects or results.
- (with less importance) the internal classes' inner state changes in an erroneous manner.

2.7.1 Testing

The sources of failure are errors found in the internal classes methods. We reasoned that to produce a satisfying test suite it is sufficient to test the way the *state* field interacts with the StateContext methods: by testing the field in fact we are also testing the way the instance's own methods are implemented.

We also decided to not rigorously test the way the ConcreteStates inner state is updated since we reasoned it is only a lesser error source.

Data Flow: field *state* We have represented the field's interaction with the class methods through a data flow graph in which the granularity was set such that basic blocks are represented by functions.

Data Flow Graph Both the Client and ConcreteStates methods interacts with the StateContext functions. The Data Flow Graph can be seen in Figure 2.16.

We decided to test the interaction between the variable and the methods following the *all-uses* criterion.

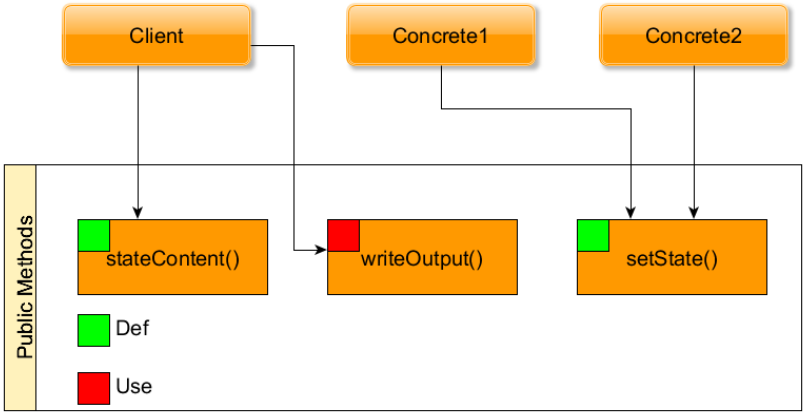

Figure 2.16: Data Flow Graph: *state*

Tests

Data Flow: field *state* We generated a test suite capable of testing all the *all-uses* paths:

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

The last test case is actually used to test the correctness of the way ConcreteStates update the field and does not cover a particular D-U path.

Since the tested class's methods (StateContext) interacted with external classes (ConcreteStates) we differentiated between Unit and Integration tests.

In particular the third integration test does not have a corresponding unit test case since it would be identical to the first unit test case.

We utilized Mockito's *mock* function to isolate errors with an origin in external classes from interfering with the main class's code.

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 100% (Unit: StateContext: 100%)

Since the Unit Tests and the Integration Tests are identical, with the only difference that the Unit Tests utilize Mockito's help to isolate from external classes, the coverage of the classes' code is the same.

2.8 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: the visitor class implements all of the appropriate specializations of the virtual function. The visitor takes the instance reference as input, and implements the goal through double dispatch.

Class Diagram

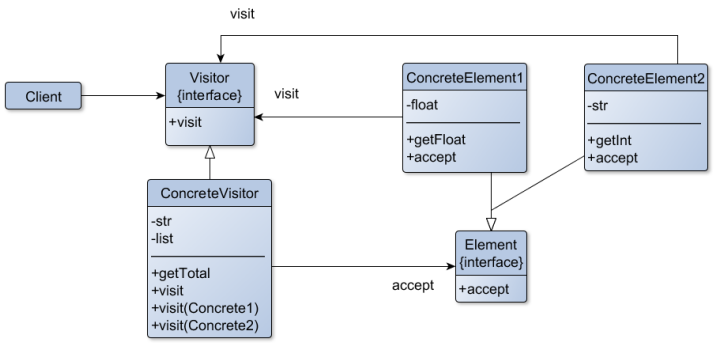

Figure 2.17: Visitor: Class Diagram

- **Visitor**: interface to the concrete implementations.
- **ConcreteVisitor**: implements the various specific way of visiting concrete elements, the actual way an element is visited is determined by the signature of the method called in the double dispatch.
- **Element**: interface of visitable element.
- **ConcreteElement**: contains specific fields and logic, is visitable.

Fault Model

The pattern focuses on treating uniformly objects of different types while operating on them with different specializations of the same function. A failure is produced by the following situation:

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

2.8.1 Testing

The sources of failure depend on the inability of the Visitor to identify the specialized implementation to execute. This depends mainly on the type of Element we are working with. We thus reasoned that to produce a satisfying test suite we must test the way *visit* works when applied to all possible hierarchies of Element types.

Topology: method *visit* We generated a Class Dependency Graph to identify all the possible relations between classes.

Class Dependency Graph In the Figure 2.18 are made explicit the relations between classes.

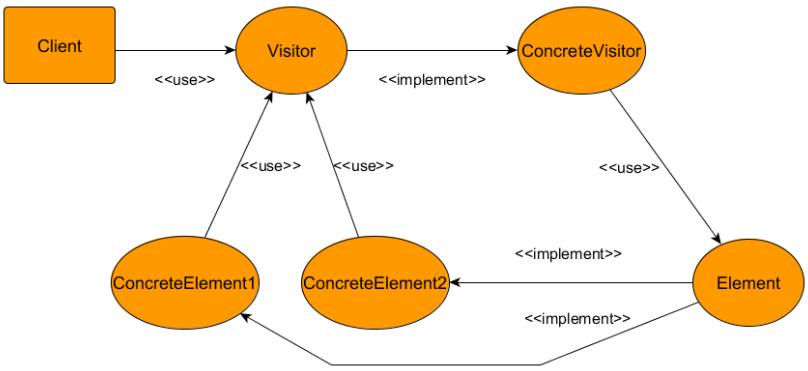

Figure 2.18: Class Dependency Graph: *visit()*

We decided to generate a test suite dependent on the *all-edges* criterion, in this case it is identical to the *all-nodes* criterion.

Tests

Topology: method *visit* We identified the 2 cases of:

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

as representative of the entire set of possible hierarchies.

Since in this pattern the tested class's methods (Visitor's) interacted with external classes (ConcreteElements) we differentiated between Unit and Integration tests.

We utilized Mockito's *mock* function to isolate errors with an origin in external classes from interfering with the main class's code.

In particular, we explored Mockito's functionalities and decided to utilize its Answer class to directly produce side-effects once the *accept()* method was called, rather than mechanically calling the generalized *visit()* and then the specialized version (or directly this second one).

Code Coverage

The code coverage measure obtained from EclEmma Java plugin is: 100% (Unit: Visitor: 100%)

Since the Unit Tests and the Integration Tests are identical, with the only difference that the Unit Tests utilize Mockito's help to isolate from external classes, the coverage of the classes' code is the same.

Chapter 3

JUnit & Mockito

3.1 JUnit

3.2 Mockito

answer serve per fare cose complesse es. particolare logica che decide che fare, qui lo ho usato per cose banali

EclEmma EclEmma is a plug-in which measures the branch coverage of the bytecode produced by the compiler. The branches which were not tested are then traced back to the code and highlighted as a warning.

In some cases such highlighting is confusing and can even be impossible to eliminate through testing: the Java compiler in fact sometimes creates additional bytecode that seems to have no relation to the source code (e.g. synthetic classes and methods).

In some other cases test execution is questionable or impossible by design: for example private, empty default constructors (assuming they receive no calls) or methods that contains no logic, like plain getters and setters, or even extra exception handlers installed to close resources from try/with statements.

Chapter 4

Conclusioni

In questo elaborato abbiamo mostrato un sistema in grado di riconoscere automaticamente i muri di una planimetria, indipendentemente dalla notazione grafica utilizzata. In questo programma, se i muri rientrano nelle tipologie citate, indipendente dagli altri standard e non necessita di informazioni aggiuntive per svolgere il proprio compito. Dalla bont dei risultati mostrati nel paragrafo precedente si pu notare che quello presentato un metodo valido per l'individuazione di muri in planimetrie. Le idee di base, in parte ispirate agli articoli [1] e [**?**] e in parte sviluppate ex novo, si sono dimostrate valide per raggiungere l'obiettivo prefissato. A sostegno di quanto appena detto viene ricordato quanto riportato nel capitolo precedente: ossia il fatto che i nostri risultati sono molto simili a quelli ottenuti da CVC ed in alcuni casi sono risultati anche migliori. Con l'utilizzo di queste idee appena esposte sarebbe possibile arrivare ad un riconoscimento al 100% indipendente dagli standard grafici utilizzati solo con un'aggiunta di codice che, riconosciuti i muri non pieni, si limiti a riempirli. Una volta raggiunto questo obiettivo un altro sviluppo futuro potrebbe essere il riconoscimento (gi effettuato da CVC) di altri elementi strutturali quali porte, finestre e stanze.

Bibliography

[1]

List of Figures