Design Patterns: Observer and Decorator

Mikael Svahnberg[*]

2023-09-15

Contents

1	Introduction	2
2	A Quick Recap of the Strategy Pattern	2
3	Responsibilities and Coupling	3
4	Connecting an Application	3
5	Pushing Data	4
6	Pulling Data	4
7	Program to an Interface	5
8	Program to an Interface: DataPublisher	5
9	Program to an Interface: Observer	5
10	Design Pattern: Observer	6
11	Finding the Subject	7
12	Summary of the Observer Pattern	7
13	Composition over Inheritance	7
14	Listening to Multiple Publishers	8
15	Observer and Strategy	9
16	When the Behaviour is Not Enough	9
17	Inheritance Hell	10
18	Design Pattern: Decorator	10
19	Tea, Earl Grey, Hot	11

[*]Mikael.Svahnberg@bth.se

20 Summary of Decorator **12**

21 Wrapper? **13**

22 Tea, Earl Grey, Hot with Strategy **14**

23 Tea, Earl Grey, Hot with Observer **14**

24 Summary **14**

25 Next Lecture: Constructors, Destructors, Pointers and References **15**

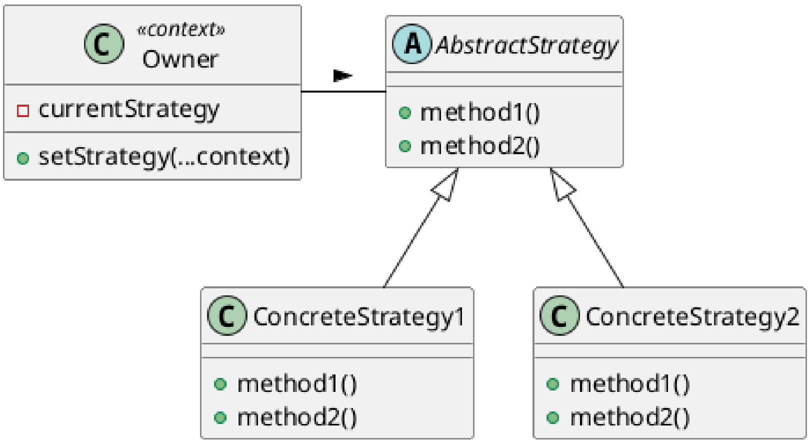
1 Introduction

- Freeman & Robson, Chapter 2: The Observer Pattern
- Freeman & Robson, Chapter 3: The Decorator Pattern
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*
- Design Principle: *Low Coupling*
- Design Principle: *Composition over Inheritance*
- Design Principle: *Open-Closed Principle*
- Design Pattern: *Observer*
- Design Pattern: *Decorator*

2 A Quick Recap of the Strategy Pattern

Responsibility Driven Design and the Strategy Pattern

- Each point of variation is *encapsulated* in a strategy pattern
  - An abstract strategy class defines the public interface
  - Each strategy is *encapsulated* in a separate class
  - Each strategy class only deal with one specific variation, i.e. *high cohesion*
  - Editing a strategy is done in one single class, i.e. *localised change*
- The «context» class is *loosely connected* to the current strategy, via *indirection*



3 Responsibilities and Coupling

Clean responsibilities means more classes and more connections between classes.
However, each connection:

- focus on only one thing: the single responsibility of the other class
- can only access the public interface: the rest is encapsulated and private
- may be further loosened by connecting to an *interface* rather than an *implementation*

$\sum$ "low coupling" does not mean "no coupling"

4 Connecting an Application

Assume:

- One object in the application is responsible for some data, e.g.
  - a table in a database
  - a data structure
  - a sensor reading, e.g. the current temperature
  - a part of the user interface, e.g. whether a specific button is being pressed or not.
- Another part of the application would like to act when this data changes, e.g.
  - Update a display
  - Update a prognosis for when something will be done
  - Apply business rules to ensure that data is within contractually specified parameters
  - Act on the pressed button in the GUI

We can use a *push* or a *pull* solution for this:

- The data responsible can *push* via method calls when the data changes
- The consumers can *pull* from the data responsible by regularly checking in on the data responsible

5 Pushing Data

```
public class DataPublisher {
// ...
  public void dataChanged() {
    DisplayConsumer dc = DisplayConsumer.getDisplayByName("disp0001");
    dc.update(myData);

    ArrayList <ActionConsumer> aclist = ActionConsumerManager.getActionConsumers("myDataTy
    for(ActionConsumer ac : aclist) {
      ac.dataChanged(myData);
    }

    PrognosisGenerator pg = new PrognosisGenerator();
    pg.fetchOldData();
    pg.addData(myData);
    pg.updatePrognosis();
    pg.storeUpdatedPrognosis();
  }
}
```

- Each consumer may have their own interface
  - We are adding responsibilities to the data publisher: to know the interface of every consumer
- Adding a new consumer require us to edit the `DataPublisher`
  - implying we need access to the source code
- *We are programming to implementations, not interfaces* → Higher Coupling
- We have less encapsulation, since each new consumer is free to expose their own way of getting the data.
- Adding more data attributes means changing the interface on every consumer

6 Pulling Data

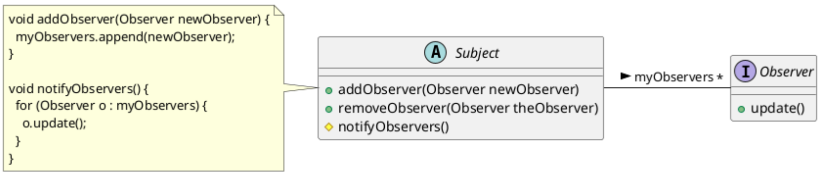
```
public class DisplayConsumer {
  private DataPublisher myDataResponsible;

  public void update() {
```

```
    setDisplay( myDataResponsible.getData() );
  }
}
```

A few questions remain:

- Who calls the `update()` method? Why? How often?
- How did the `DisplayConsumer` find the correct `DataPublisher` object?
- What if the `DataPublisher` has more data attributes?

7 Program to an Interface



8 Program to an Interface: DataPublisher

```
public class DataPublisher extends Subject {
  public void dataChanged() {
    this.notifyObservers();
  }
}
```

- Everything about being a `Subject` is encapsulated.
- `DataPublisher` can focus on its real responsibility without knowing anything about `Observers`.
- New `Observers` can be created without acccess to the source code for `Subject` or `DataPublisher`
- *We program to an interface*
  - `Subject` present the same interface for all data publishers
  - `Observer` present the same interface for all data consumers
  - The specific behaviour of each data consumer is *encapsulated*
- Low coupling: `Subject` only knows about the `update()` method in `Observer`

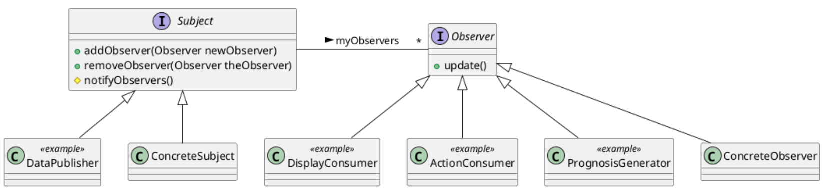
9 Program to an Interface: Observer

```
public class DisplayConsumer implements Observer {

  public void update() {
    DataPublisher theDataRespobsible;
```

```
  // In order to get the updated data, we do need access to the DataPublisher
  // The question is... How do we find it?

  setDisplay( theDataResponsible.getData() );
}

public DisplayConsumer(DataPublisher theDataRespobsible) {
  // It could, for example, be given to us in the constructor
  // Or, we create an object in our constructor
  // Or, we fetch it from somewhere else.
  theDataResponsible.addObserver(this);

  // Either way, we may wish to store it
  myDataResponsible = theDataResponsible;
}
```

- We still need to find the concrete data publisher.
  - The `Subject` class is not enough since it does not know what type of data is being published
  - However, we only need it inside the `update()` method.
  - We may store a reference as a private attribute in our object but it is only being *used* in the `update()` method.
- *We program to an interface*
  - `Observer` present the same interface for all data consumers
  - The specific behaviour of each data consumer is *encapsulated*

10 Design Pattern: Observer



- `Subject` is often known as `Observable`
- `Subject` can be implemented as a class or defined as an interface
  - When `Subject` is a class we "use up" our one chance to `extend` the sub-classes.
  - When `Subject` is an interface, we can still `extend` and `implement` other classes and interfaces
  - The implementation for the methods in `Subject` are easy enough anyway...

11 Finding the Subject

- The concrete `DataPublisher` object could be given to us as a parameter to the constructor
- Or, we create an object
- Or, we fetch it from somewhere else
- we should store the `DataPublisher` object in a private attribute
  - In case we need access to the object in order to read the updated data
  - In case we want to remove ourselves as Observers.
- We could also change the `update()` method:

```
public interface Observer {
  public void update(EventData theData);
}

public interface EventData {
  // Maybe no methods here, it is just a "placeholder" to inherit from
  // which we cast to the right type before accessing the
  // specific data attributes for one specific type of event.
}
```

12 Summary of the Observer Pattern

- Loose Coupling: `Subject` only knows about `Observer` and `Observer::update()`
- Encapsulation: Anything else is encapsulated in sub-classes to `Observer`

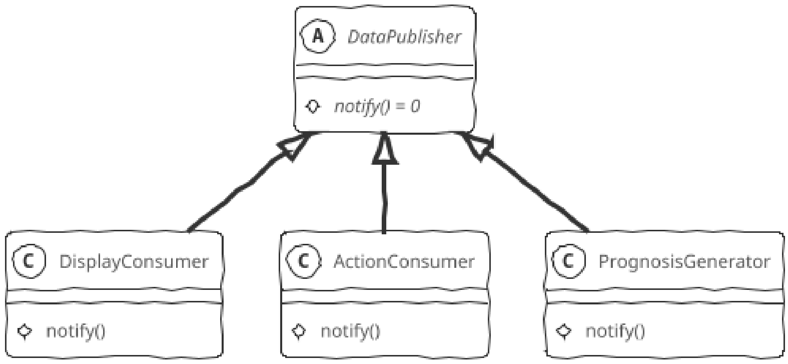
Discuss How does the Observer pattern address:

- Separate aspects that vary from what stays the same
- Program to an Interface, not an Implementation
- Favour Composition over Inheritance

13 Composition over Inheritance

- We *could* have created one sub-class for every observer
  - Requires the `notify()` method to exist and be abstract in the base class
  - If we want to listen to more publishers, we need multiple inheritance (→ not *low Coupling*)
  - A single datum can now only be watched by a single type of consumer (→ not *low Coupling*)

- It is decided at compile-time which consumer to use (→ not *low Coupling*)
- Each consumer is now both a publisher and a consumer (→ not *High Cohesion*)



- By instead using composition, e.g. with the *Observer* pattern
  - we can add new consumers at runtime (→ *low Coupling*)
  - each class is responsible for one and only one thing (→ *High Cohesion*)
  - each consumer can listen to many data publishers (→ *low Coupling*)

14 Listening to Multiple Publishers

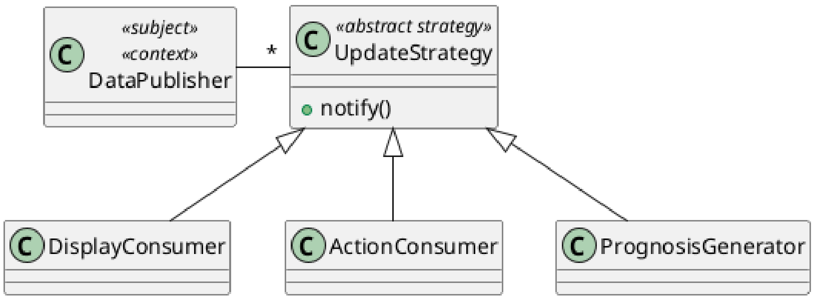
- What if we want to listen to two or more publishers?
  - All of them will call the same `update()` method.
  - How do we know which one that was called?
  - This is typical if you have several GUI buttons in your java application
    * each button will want to call `actionperformed()`
- One solution: Pass along an event object `update(Event theEvent)`
- Another solution: Inner classes
- A third solution: Lambda functions

```
public class MultipleoBserver {
  public MultipleObserver(Subject firstPublisher, Subject secondPublisher) {
    firstPublisher.addObserver(new FirstListener());
    secondPublisher.addObserver(new SecondListener());
  }

  class FirstListener implements Observer { // Inner class, works as a private member
```

```
    public void update() {
      system.out.println("the first publisher just updated");
    }
  }

  class SecondListener implements Observer {
    public void update() {
      System.out.println("the second publisher just updated");
    }
  }
}
```

15 Observer and Strategy

- *Observer* uses the same mechanism as *Strategy*
- An `Observer` class is separate and a first class entity
  - It can be known by many
  - It can be used by other parts of the system
- A `Strategy` class is encapsulated by the «context» class
  - It is only known by its «context» class
  - It can only be used via its «context» class



16 When the Behaviour is Not Enough

When we want to extend the behaviour of a class we can either:

- Find the implementation and add our new behaviour
  - Bad idea: maybe other parts of the system depended on the first implementation
  - Do we even have access to the source code?
- Inherit to a new class and extend there
  - Better idea, the original class remains untouched

  - New code that requires the new behaviour will simply use the subclass instead
  - **Note**: The original *interface* of the class remains the same!
    * We are only changing the *implementation*.

Design Principle: *Open-Closed Principle*

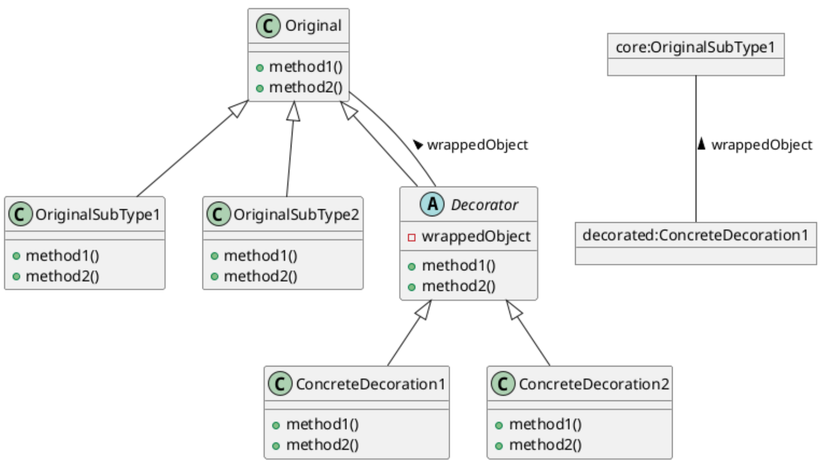
- A class should be *open* for extension
  - We can add or modify behaviour e.g. through inheritance
- A class should be *closed* for modification
  - modify the code → original tests no longer apply
  - modify the code → break other parts of the system

17 Inheritance Hell

- We use inheritance to *define type* as well as *reuse behaviour*
- When these two goals collide, the number of classes explode
  - Every combination of types gets their own class
  - Every new type means one new class for every existing combination
- We may break this up into separate inheritance hierarchies
  - One for each main type
  - One main class that composes the desired behaviour of each main type
  - $\sum$ One application of the *Strategy pattern* for each main type
  - Add a new type → update the strategy «context» class or its factory
  - Add a new type hierarchy → update the strategy «context» class.

Or, we can use the Design Pattern *Decorator*

18 Design Pattern: Decorator

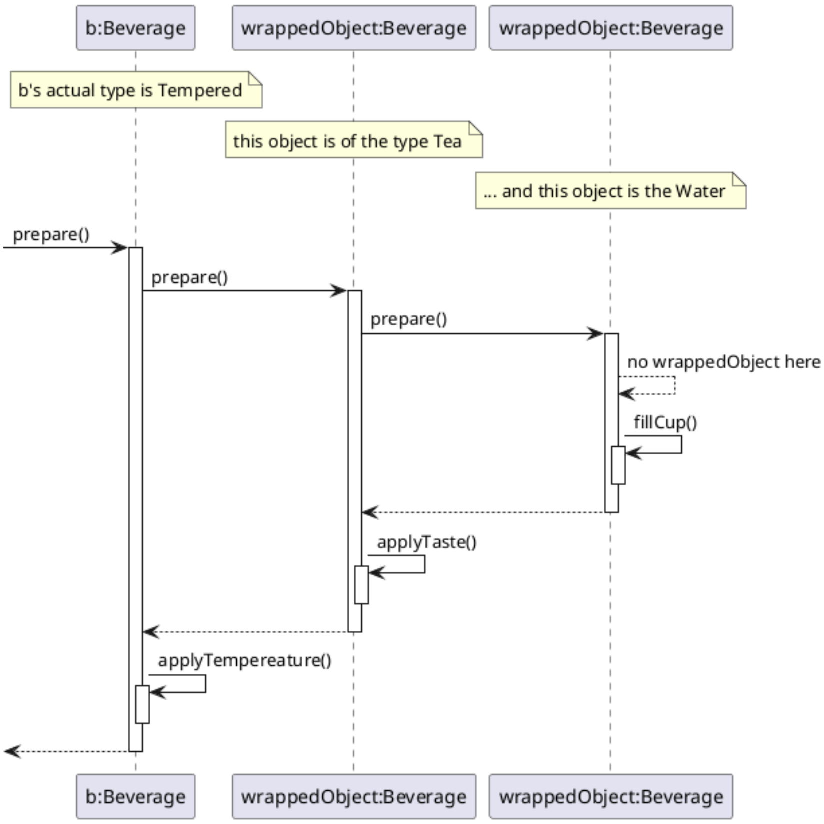


- `Original` can remain untouched, as can its original subclasses.
- We add the abstract class `Decorator` which inherits from `Original`
  - Inheritance → we can use Decorator objects *as if they were* of the type Original
- `Decorator::wrappedObject` is a reference back to an object of the type `Original`
  - Every method can implement their own behaviour
  - Every method *may* or even *should* call the same method on `wrappedObject`.

```
public class ConcreteDecoration1 {
  private wrappedObject;

  public ConcreteDecoration1(Original theWrappedObject) {
    this.wrappedObject = theWrappedObject;
  }

  public int method1() {
    // Do something unique here

    // Then pass control along to the wrapped object to
    // do their stuff. Compound this with our result and
    // pass upwards.
    return result + wrappedObject.method1();
  }
}
```

19 Tea, Earl Grey, Hot

```
Beverage b = new Water();
b = new Tea("Earl Grey", b);
b = new Tempered(98, b);

b.prepare();
```



- What would be the result of the following:

```
Beverage b = Tempered(98);
b = new Tea("Earl Grey", b);
b = new Water(b);
```

20 Summary of Decorator

- With decorator we *replace* our object reference
  - to an object of the same *supertype*
  - that contains the original object
  - that adds behaviour and calls the original object
    * or calls the original object and then adds behaviour
- We are *wrapping* the original behaviour.
  - Decorator can sometimes be called *Wrapper*, but this is not fully correct.
- Design Principle: *Encapsulation*
  - Each added layer of behaviour is encapsulated in its own class

    - Once created, no-one needs to know which layers an object is composed of.

- Design Principle: *Low Coupling*
    - The outside object reference is to the original top-level class
    - The `wrappedObject` object reference is *also* to the original top-level class
    - Any involved class only ever knows about the public interface of the original class.

- Design Principle: *Open-Closed*
    - Adding new behaviour is done through inheritance
    - The original classes are re-used without modification

21 Wrapper?

Wrapping one object inside another object is used in different ways:

Decorator add new layers of "flavour" to an object

- Keep the interface stable, add layers of new behaviour
- Decorations can be added at runtime, altering the behaviour of an object reference at runtime.

Adapter Replace the original interface with a new one

- Keep the behaviour stable, wrap it into a new interface
- Re-use an existing class in a new context, where a different interface is expected
- Adapter solves a compile-time problem

Facade Present a simple interface

- Hide complicated interfaces and behaviour, possibly composed of multiple objects.
- Facade solves an implementation-time problem

Proxy Replace a "heavy" object with a lighter one that manages the heavy object

- Keep interface stable, keep behaviour stable, manage the infrastructure
- e.g., the proxy object may call the real object on a remote server and translate local method calls to remote procedure calls
- Proxy solves a runtime problem, but is "fixed" at implementation-time.

22 Tea, Earl Grey, Hot with Strategy

```
public class Beverage {
  Liquid myLiquid;
  Flavour myFlavour;
  Temperature myTemperature;

  public Beverage() {
    myLiquid = new WaterStrategy();
    myFlavour = new TeaStrategy("Earl Grey");
    myTemperature = new HeatStrategy(98);
  }

  public void prepare() {
    // This order is hard coded at compile time
    // and must be updated if we would like to add something more.
    myTemperature.prepare(myLiquid.prepare(myFlavour.prepare()));
  }
}
```

23 Tea, Earl Grey, Hot with Observer

I started doing this, but it just got silly:

1. Start by creating objects of Water, Tea, and Tempering, and connect them as follows.
2. The Water (as Subject) will announce that it is poured
3. The Tea (as Observer of the Water) will add itself to the water.
4. The Tea (as Subject) would announce that it is added
5. The Tempering (as Observer of the Tea) would start heating the water

$\sum$ all objects must know about all other objects in their roles as

- Subject, so they can add themselves as observers
- Observers, so they can notify each other when they are ready
- Their main area of responsibility, so they can perform the right action

Conclusion: **Don't do this**

24 Summary

- Design Pattern: *Observer*
- Design Pattern: *Decorator*
- Overview of related Design Patterns: *Wrapper*, *Adapter*, *Facade* and *Proxy*

- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*
- Design Principle: *Low Coupling*
- Design Principle: *Composition over Inheritance*
- Design Principle: *Open-Closed Principle*

Also:

- Java inner classes
- UML Sequence Diagrams

25 Next Lecture: Constructors, Destructors, Pointers and References

- Constructors and Destructors
- Copy Constructors
- Pointers and References
- Identity and Equality
- Operator Overloading
- Constants