I402A Software Architecture
and Quality Assessment

Session 2

Object-Oriented Design Pattern

Sébastien Combéfis

Fall 2019

Objectives

- Presentation of object-oriented design patterns
  - Definition and characterisation of a design pattern
  - Presentation of the Gang of Four (GoF) classification
- Use examples of several GoF design patterns
  - **Creational**: Builder
  - **Structural**: Facade, Adapter
  - **Behavioural**: Template Method, Observer, Memento

Design Pattern (1)

- A design pattern is a solution to a common problem

  Repeatable solution to apply when designing software

- It is a model which describes how to solve the problem

  It is not a code that is just meant to be imported

- Speed up software development

  Tested solution proved to be adapted to each problem

Design Pattern (2)

- Reusable model to be used to generate something

  A code, a package, a framework, an architecture, a UI design, etc.

- Four elements are required to describe a design pattern
  - Its **name**
  - A **description of the problem** for which it is applicable
  - The **solution** as a description of its application
  - The **consequences** of applying it

Gang of Four

Software Design Pattern

- Software design patterns by Gang of Four (GoF) in 1994

  Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

- Three main categories and 23 patterns

  1. **Creational** patterns

     Class instanciation

  2. **Structural** patterns

     Class and object composition

  3. **Behavioural** patterns

     Communication between objects

GoF Design Pattern

- The 23 design patterns defined by the GoF

Creational	Structural	Behavioural
Abstract factory	Adapter	Chain of responsibility
Builder	Bridge	Command
Factory method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template method
		Visitor

Singleton Design Pattern (1)

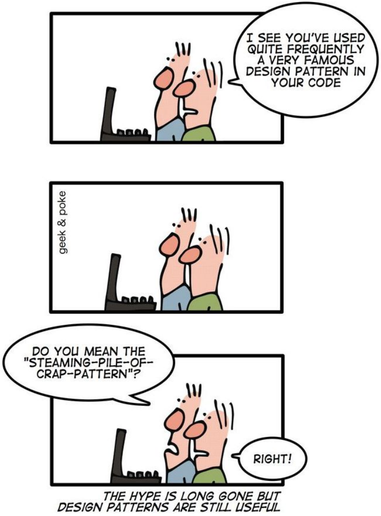
- Designing a class that can be instantiated at most once
  - This unique instance must be accessible
  - Ensure that new instances cannot be created
- **Solution**
  - Only private constructors
  - The class itself creates its own unique instance
  - Method to retrieve this unique instance

Singleton Design Pattern (2)

- Two other important details related to the Java
  - Avoid simultaneously threads access by with `synchronized`
  - Avoid addition of constructors by not allowing subclasses

```
public final class Singleton
{
   private static Singleton instance;    // Unique instance

   private Singleton(){}                 // Private constructor

   // Class method to retrieve the instance
   public synchronized static Singleton getInstance()
   {
      if (instance == null)
      {
         instance = new Singleton();
      }
      return instance;
   }
}
```

Design patterns simplify your life!



No Silver Bullet!

- Design patterns are not silver bullet solution to all problem

  Source of inspiration to a set of well-known common problems

- Bad things may happen if you try to force a design pattern
  - Avoid to overthink and forcing a design to fit a design pattern
  - Solution to problems, not solution finding problem
  - Privilege the saviour design pattern, avoid a possible mess

Multi-Pattern Architecture

- Possible to combine design patterns for a given problem

  Each design pattern has its own purpose and application context

- Each pattern must be used for the correct purpose
  - In accordance to its category: creation, structure or behaviour
  - Correct actors must be well identified
  - Consequences of application must be well balanced

Six Examples

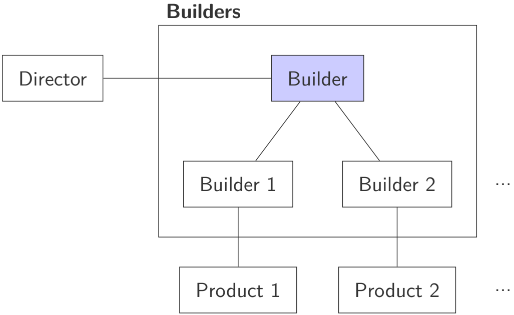
- Creational design patterns
  1. `Builder`: build complex objects
- Structural design patterns
  1. `Facade`: interface with subsystems
  2. `Adapter`: adapts an interface to another one
- Behavioural design patterns
  1. `Template method`: define algorithm skeleton
  2. `Observer`: notify observers of events
  3. `Memento`: save and restore things (state, actions, etc.)

Builder

Builder Design Pattern

- Construction/representation separation for objects

  Delegates the construction of an object to another class



Context and Application

- Complex object creation algorithm

  Independent of how the objects are assembled

- Generic construction process for a set of objects

  Based on an abstract class

- Avoid a constructor pollution of classes
  - Several “flavours” of the same object can be created
  - Object creation requires a lot of complex steps

Actors

- **Builder**

  Abstract class for the creation of parts of the product

- **ConcreteBuilder** (`Builder 1`, `Builder 2`, etc.)

  Build and assemble the parts of the product

- **Director**

  Build an object using the `Builder` abstract class

- **Product** (`Product 1`, `Product 2`, etc.)

  The complex object under construction

Builder Example (1)

```
public final class SebBurgerMenu
{
   private static enum Size {SMALL, MEDIUM, LARGE};
   private static enum Burger {CLASSIC, CHEESE, BACON};
   private static enum Drink {COCA, SPRITE, FANTA};
   private static enum Dessert {CHURROS, DONUT};

   private final Size size;
   private final Burger burger;
   private final Drink drink;
   private final Dessert dessert;

   public static final class Builder
   {
      // ...
   }

   private SebBurgerMenu (Builder builder)
   {
      size = builder.size;
      burger = builder.burger;
      drink = builder.drink;
      dessert = builder.dessert;
   }
}
```

Builder Example (2)

```java
public static final class Builder
{
   // Required
   private final Size size;
   private final Burger burger;
   private final Drink drink;
   // Optional
   private Dessert dessert;

   public Builder (Size size, Burger burger, Drink drink)
   {
      this.size = size;
      this.burger = burger;
      this.drink = drink;
   }

   public Builder dessert (Dessert dessert)
   {
      this.dessert = dessert;
      return this;
   }

   public SebBurgerMenu build()
   {
      return new SebBurgerMenu (this);
   }
}
```

Builder Example (3)

- Several ways to build a `SebBurgerMenu` through the builder

  Possible to have a menu with or without a dessert

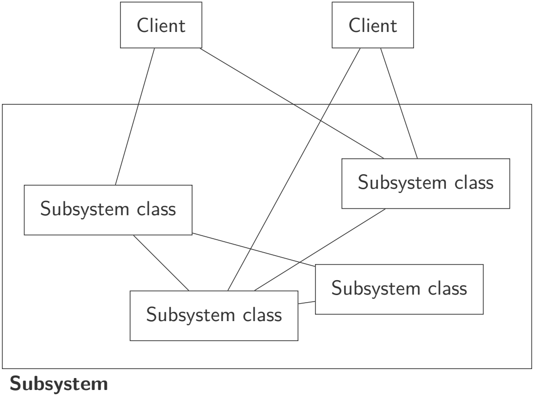
```java
public static void main (String[] args)
{
   // Simple menu avec frites, burger et boisson
   SebBurgerMenu menu = new SebBurgerMenu.Builder
                          (Size.SMALL, Burger.CHEESE, Drink.SPRITE).build();
   System.out.println (menu);

   // Menu avancé avec un dessert en plus
   SebBurgerMenu.Builder builder = new SebBurgerMenu.Builder
                                     (Size.LARGE, Burger.BACON, Drink.COCA);
   builder.dessert (Dessert.DONUT);
   System.out.println (builder.build());
}
```

Facade

Facade Design Pattern (1)

- Several clients must access to a set of subsystems

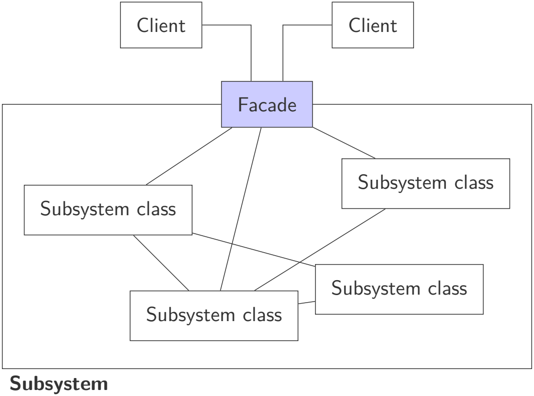
Each subsystems can be access by several clients



Facade Design Pattern (2)

- Unified entry point to a set of subsystems

  Access to all the functionalities offered by all the subsystems



Context and Application

- Simplified interface of a subsystem for some clients

  The subsystem remains completely accessible directly

- Implementation can change but the interface remains stable

  The facade makes the link with subsystem interfaces

- Decrease the coupling of the global system
  - Between clients and subsystems or between subsystems
  - But keep in mind that that facade can become a big class...
  - Several facades grouping logically related functions is possible

Actors

- **Facade**
  - Know the responsible classes for all the possible requests
  - Delegate the client requests to the appropriate objects
- **Subsystem classes**
  - Do not know that they are behind a facade
  - Manage the requests transmitted by the facade
  - Implement the functionalities of the subsystem

Facade Example (1)

- Server only allowing authenticated user to print documents

  First obtain credentials then use the printer to print

```java
public class Authentication
{
   public Credentials login (String username, String password) { /* ... */ }

   public void logout() { /* ... */ }
}

public class Printer
{
   public void turnOn() { /* ... */ }

   public void turnOff() { /* ... */ }

   public boolean isOn() { /* ... */ }

   public void printDocument (Credentials cred, Document doc) { /* ... */ }
}
```

Facade Example (2)

- Printing requires both `Authentication` and `Printer` classes

  The client code is complex and tightly coupled with two classes

```
public class Program
{
    public static void main (String[] args)
    {
        Authentication auth = new Authentication();
        Credentials cred = auth.login (/* ... */);
        if (cred != null)
        {
            Printer printer = new Printer();
            if (! printer.isOn())
            {
                printer.turnOn();
            }
            printer.printDocument (cred, /* ... */);
            auth.logout();
        }
    }
}
```

Facade Example (3)

- A facade can be designed with a method to print a document

  Encapsulate the authentication and the printing process

```
public class PrintingServer
{
   private String username, password;
   private Authentication auth;

   /* ... */

   public void printDocument (Document doc)
   {
      Credentials cred = auth.login (username, password);
      if (cred != null)
      {
         Printer printer = new Printer();
         if (! printer.isOn())
         {
            printer.turnOn();
         }
         printer.printDocument (cred, /* ... */);
         auth.logout();
      }
   }
}
```


ADAM elements
FUSE
Adapter

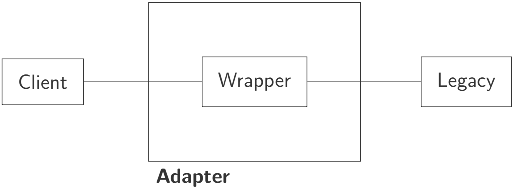
Adapter Design Pattern

- A client wants to use a legacy code but with a new interface

  Can be done if an adapter is provided, similar to plug adapters

- Wrapper transforms requests from client to request to legacy

  Makes compatible an initially incompatible object



Context and Application

- Convert an interface to another one expected by the client

  Can be seen as a wrapping of a class in another interface

- Typically used when willing to work with legacy code
  - Impedance match with an old component to a new system
  - Easier than completely rewriting the old component
  - Excellent opportunity to reuse code at lower cost

Actors

- **Legacy**
  - The interface or class to be used by the new client
  - Contain methods that cannot be directly called
- **Wrapper**
  - Contain methods that can be called by the new client
  - Wrap method calls to convert to calls in legacy code
  - Could implement the interface used by the client

Adapter Example (1)

- Using legacy code directly may require a lot of code

  Not easy to structure the code in a general way

```java
public class LegacyWorker
{
   public String compute();
}

public class Program
{
   public static void main (String[] args)
   {
      LegacyWorker worker = new LegacyWorker (/* ... */);
      String s = worker.compute();
      Json result = parseString (s);

      // This client needs a Json object...
   }

   private Json parseString (String s) { /* ... */ }
}
```

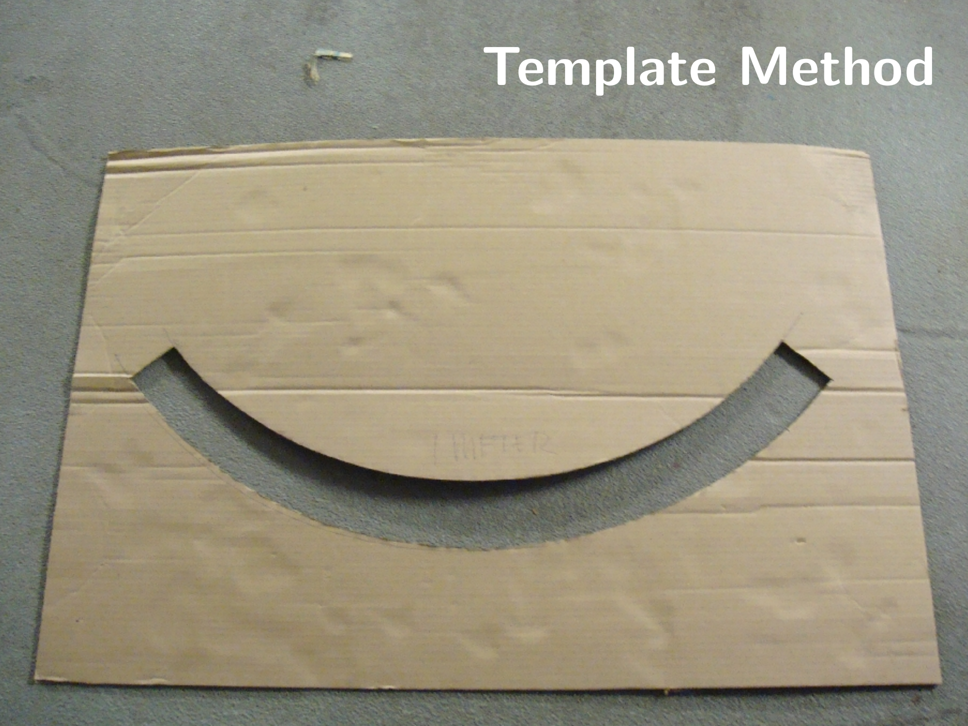
Adapter Example (2)

- Adapter pattern define a wrapper to the legacy code

  The client uses an interface representing its requirements

```
public interface Worker { public Json compute(); }

public class Adapter implements Worker
{
   private LegacyWorker lw;

   // ...

   public Json compute() { return parseString (lw.compute()); }

   private Json parseString (String s) { /* ... */ }
}

public class Program
{
   public static void main (String[] args)
   {
      Worker worker = new Adapter (/* ... */);
      Json result = worker.compute();

      // This client needs a Json object...
   }
}
```

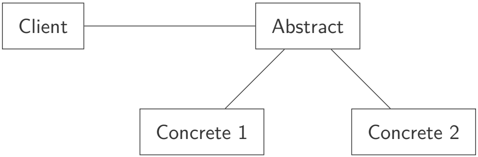
Template Method



Template Method Design Pattern

- Define the skeleton of an algorithm with holes to be filled

  Concrete operations are defined in the subclasses



Context and Application

- Several similar algorithms but with variable parts

  The same structure but some operations differ

- Factorisation of the common parts in a single superclass

  Specific parts are put in the subclasses

- Enforcing a control on the liberty for subclasses

  By defining precise "hooks" where code can be specialised

Actors

- **Abstract class**
  - Define abstract primitive operations
  - Define the skeleton of an algorithm based on the primitives
- **Concrete class**

  Implement the primitive operations, filling the hooks

Template Method Example (1)

```python
class Sorter:
   __metaclass__ = ABCMeta

   # Sort the tab array
   def sort(self, tab):
      while not self._isSorted(tab):
         for i in range(len(tab) - 1):
            if (self._compare(tab[i], tab[i + 1]) > 0):
               self._swap(tab, i, i + 1)

   # Swap values at index i and j in tab
   def _swap(self, tab, i, j):
      tab[i], tab[j] = tab[j], tab[i]

   # Test whether the tab array is sorted
   def _isSorted(self, tab):
      for i in range(len(tab) - 1):
         if self._compare(tab[i], tab[i + 1]) > 0:
            return False
      return True

   # Compare x and y
   # <0 if x is before y
   # >0 if x is after y
   # =0 otherwise
   @abstractmethod
   def _compare(self, x, y):
      pass
```

Template Method Example (2)

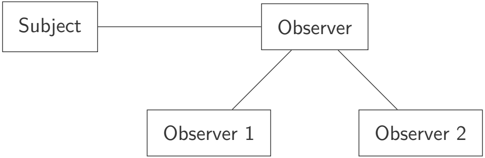
```python
# Ascending order sort
class AscSorter(Sorter):
    def _compare(self, x, y):
        return x - y

# Descending order sort
class DescSorter(Sorter):
    def _compare(self, x, y):
        return y - x

if __name__ == "__main__":
    tab = [7, 2, 9, -5]
    print(tab)

    sorter = AscSorter()
    sorter.sort(tab)
    print(tab)

    sorter = DescSorter()
    sorter.sort(tab)
    print(tab)
```


Observer

Observer Design Pattern

- Observers are notified of state changes of a subject

  Define a one-to-many dependency between objects



Context and Application

- Several objects depend on the value of another single object

  Must execute something as soon as main object changes

- Core component of the system encapsulated in the subject

  Variable component represented in an observers hierarchy

- Used as the view in the model-view-controller architecture

  Used to create less coupling and better modularity, evolution

Actors

- **Subject**
  - Object whose state changes should be monitored
  - Maintain a list of registered observers
  - Notifies the observers whenever a change occurred
- **Observer**
  - Monitor the changes in the state of a subject
  - Attach themselves to one or several subjects
- **Concrete observer**

  Changes their own states whenever a subject change is notified

Observer Example (1)

- The subject maintains a list of observers to notify

  They are represented by an `Observer` interface

```java
public class Sensor
{
   private List<Observer> observers;

   // ...

   public registerObserver (Observer obs) { observers.add (obs); }

   public void run()
   {
      while (true)
      {
         // ...
         for (Observer o : observers)
         {
            o.notify (value);
         }
         // ...
      }
   }
}
```

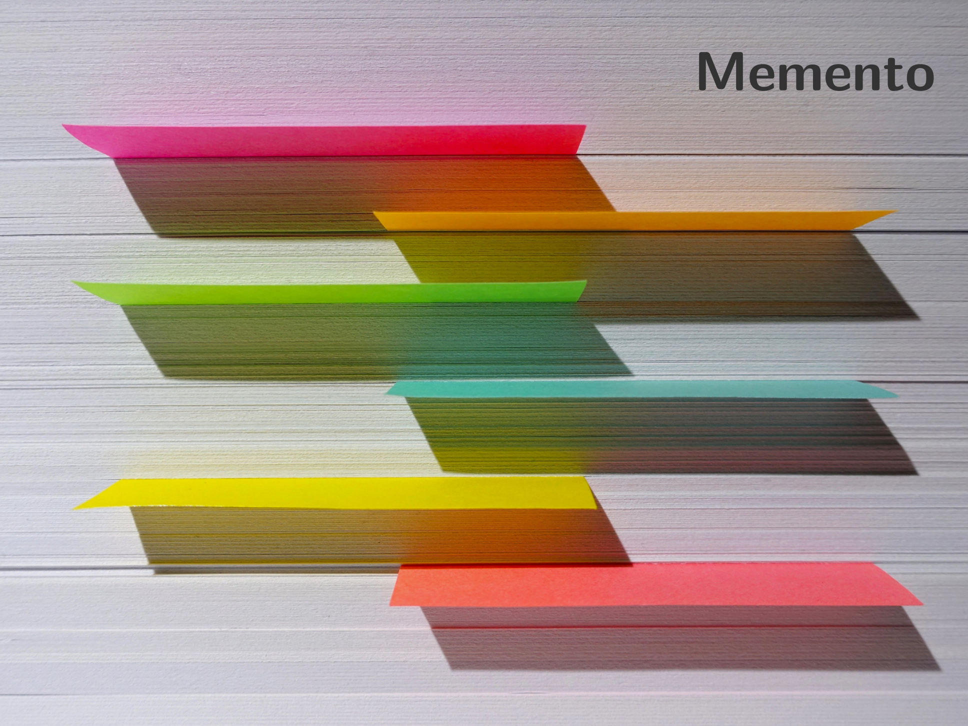
Observer Example (2)

- The concrete observer executes some action whenever notified

  Can contain information about the event that occurred

```
public interface Observer
{
   public void notify (int value);
}

public class WarningObserver implements Observer
{
   // ...

   public void notify (int value)
   {
      if (value > threshold)
      {
         System.out.println ("WARNING!");
      }
   }
}
```

Memento



Memento Design Pattern

- Capturing and saving externally the internal state of an object

  Typically to be able to restore the object's state

- Useful to propose an "undo/redo" capability

  States of an object are stacked (push to save, pop to restore)

Context and Application

- Used when needing to restore an object back to previous state
  - "undo/redo" for a desktop application
  - "commit/rollback" to manage database transaction
- Used to implement a checkpoints capability

  Need to define what state should be saved

Actors

- **Originator**
  - Object that knows how to save itself (its own state)
  - Manipulate memento objects to save/restore states
- **Memento**
  - The lock box in which states are stored
  - Written and read by the originator, shepherded by caretaker
- **Caretaker**

  Trigger the saving and restoring operations of states

Memento Example (1)

- Originator object saves/restores its own state with memento

  Use memento objects to keep track of the states

```
public class Editor
{
   private Object content;

   // ...

   public Memento save()
   {
      return new Memento (content);
   }

   public void restore (Memento memento)
   {
      content = memento.getContent();
   }
}
```

Memento Example (2)

```
public class Memento
{
   private Object content;

   public Memento (Object content)
   {
      this.content = content;
   }

   public void getContent()
   {
      return content;
   }
}

public class Program
{
   public static void main (String[] args)
   {
      Editor editor = new Editor();
      // ...
      Memento saved = editor.save();
      // ...
      editor.restore(saved);
   }
}
```

References

- Erich Gamma, Richard Helm, Ralph Johnson, & John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994. (ISBN: 978-0-201-63361-0)
- Kamran Ahmed, *Design Patterns for Humans!*, November 28, 2018. https://github.com/kamranahmedse/design-patterns-for-humans
- Eric Freeman, & Elisabeth Robson, *5 reasons to finally learn design patterns*, October 12, 2016. https://www.oreilly.com/ideas/5-reasons-to-finally-learn-design-patterns
- The Educative Team, *The 7 Most Important Software Design Patterns*, November 8, 2018. https://medium.com/educative/the-7-most-important-software-design-patterns-d60e546afb0e

Credits

- Book pictures from Amazon.
- tobym, November 8, 2006, https://www.flickr.com/photos/48089670@N00/293829728.
- Oliver Widder (Geek and Poke), http://geekandpoke.typepad.com/.a/6a00d8341d3df553ef0147e3cff536970b-800wi.
- clement127, December 29, 2014, https://www.flickr.com/photos/clement127/15979531229.
- Patrick Cain, March 15, 2012, https://www.flickr.com/photos/patrickcain/6858836140.
- Marco Verch, September 3, 2018, https://www.flickr.com/photos/149561324@N03/43749290834.
- BinaryTaskforce, August 25, 2009, https://www.flickr.com/photos/binarytaskforce/4479399780.
- Grant Hutchinson, June 14, 2013, https://www.flickr.com/photos/splorp/9046310594.
- Dean Hochman, March 2, 2019, https://www.flickr.com/photos/deanhochman/47210747322.