T5 - Java Seminar

T-JAV-500

Day 08

Design Patterns

3.1

EPITECH.

Day 08

language: Java

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

In 1995, the *Gang of Four* (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) wrote a now very famous book called *'Design Patterns : Elements of Reusable Object-Oriented Software'*.
It introduced 23 **design patterns**, that are 23 ways to conveive and organize classes. Each design pattern is a reliable solution to a very common problem, a kind of good practise.
They are language-independent.

Nowadays, those design patterns are still used, and some more were added to them, and some variants appeared.
You are about to study a few of them in today's exercises in order to extract some common principles, helpful to design applications.

To represent classes' organization and content, UML diagrams will be used. You'll get a very brief introduction to these diagrams.

The use of design patterns shows many advantages:

- quality
  they are proven answers validated by experts,
- speed
  they are fast to implement and save time on conception brainstorming,
- reusability
  they can be reused in different applications without further development,
- ease
  they are fully documented and well-known
- readability
  they are a common vocabulary for many people who use and master them.

SUBDIVISION

Design patterns are subdivided into 3 categories:

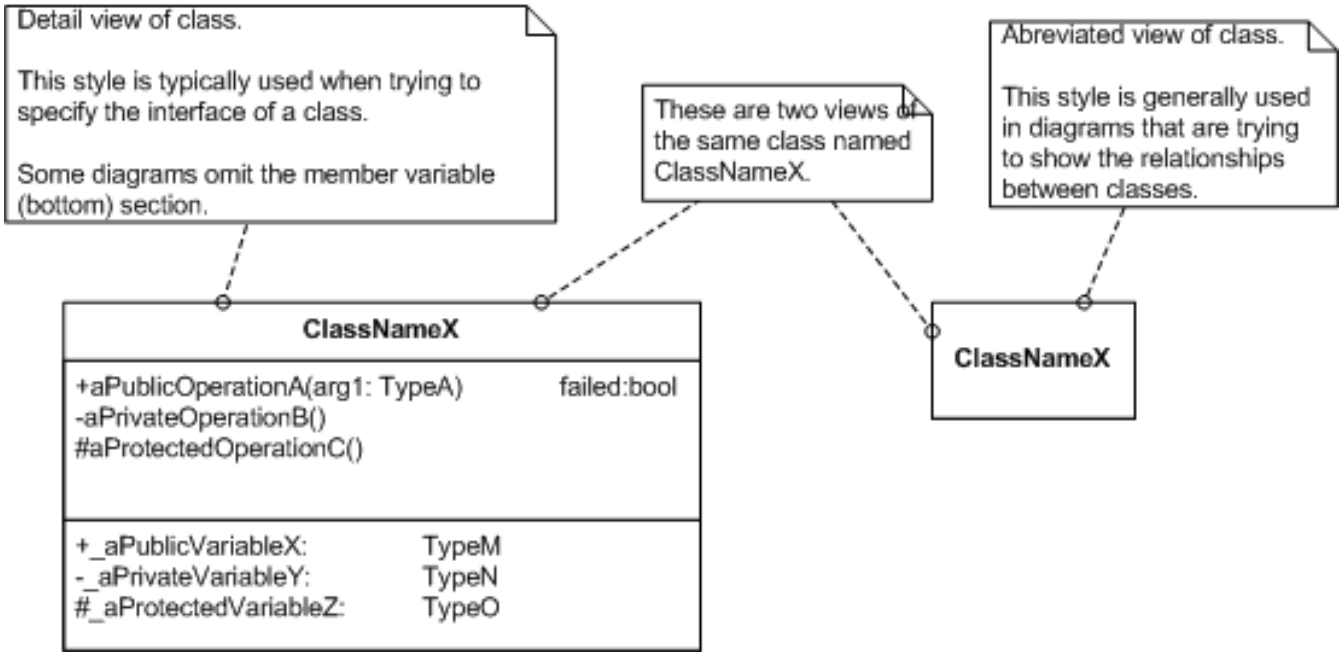
- **Creational**
  To instanciate, initialize and configure classes and objects.
  Factory, *AbstractFactory*, *Builder* and *Prototype*.
- **Structural**
  To organize and connect classes.
  Adapter, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* and *Proxy*.
- **Behavioral**
  To manage objects so that they can collaborate and interact.
  Interpreter, *Template Method*, *Chain of Responsibility*, *Command*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy* and *Visitor*.

Design patterns are not always relevant. Some are lambasted by developers, probably rightly.

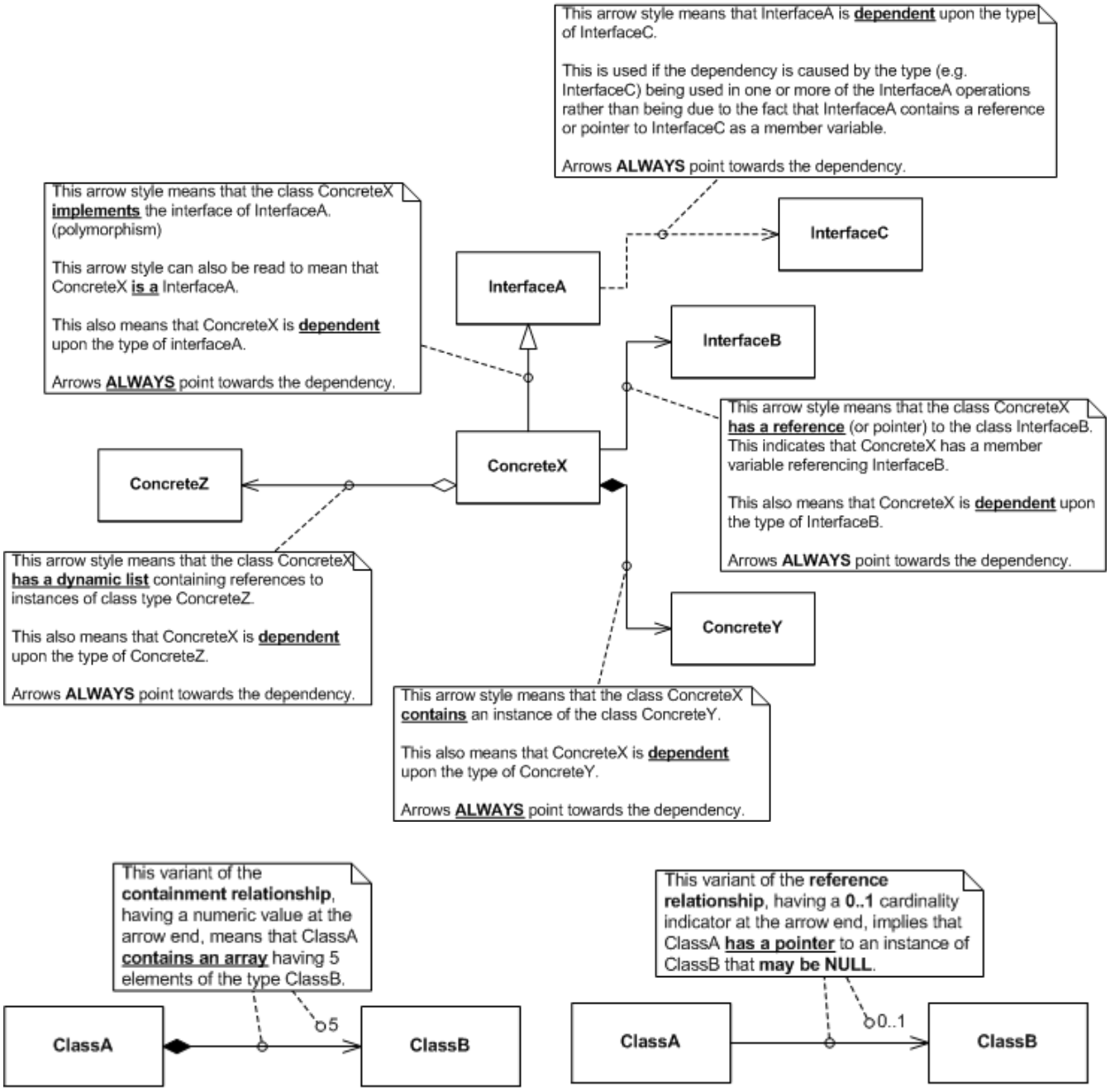
UML

UML means **Unified Modeling Language**.
It is a uniformed and practical way to represent classes and their interactions.

A class is represented by a box like the one below:


Detail view of class.
This style is typically used when trying to specify the interface of a class.
Some diagrams omit the member variable (bottom) section.
These are two views of the same class named ClassNameX.
Abreviated view of class.
This style is generally used in diagrams that are trying to show the relationships between classes.
ClassNameX
+aPublicOperationA(arg1: TypeA) failed:bool
-aPrivateOperationB()
#aProtectedOperationC()
+_aPublicVariableX: TypeM
-_aPrivateVariableY: TypeN
#_aProtectedVariableZ: TypeO
ClassNameX

Here are the different relations among them:



Ouch… It looks a bit repellent at first glance, but it is not as tough as it seems.

EXERCISE 01

Files to hand in: ./Factory/Toy.java
./Factory/TeddyBear.java
./Factory/Gameboy.java
./Factory/Factory.java
./Factory/GiftPaper.java
./Factory/NoSuchToyException.java

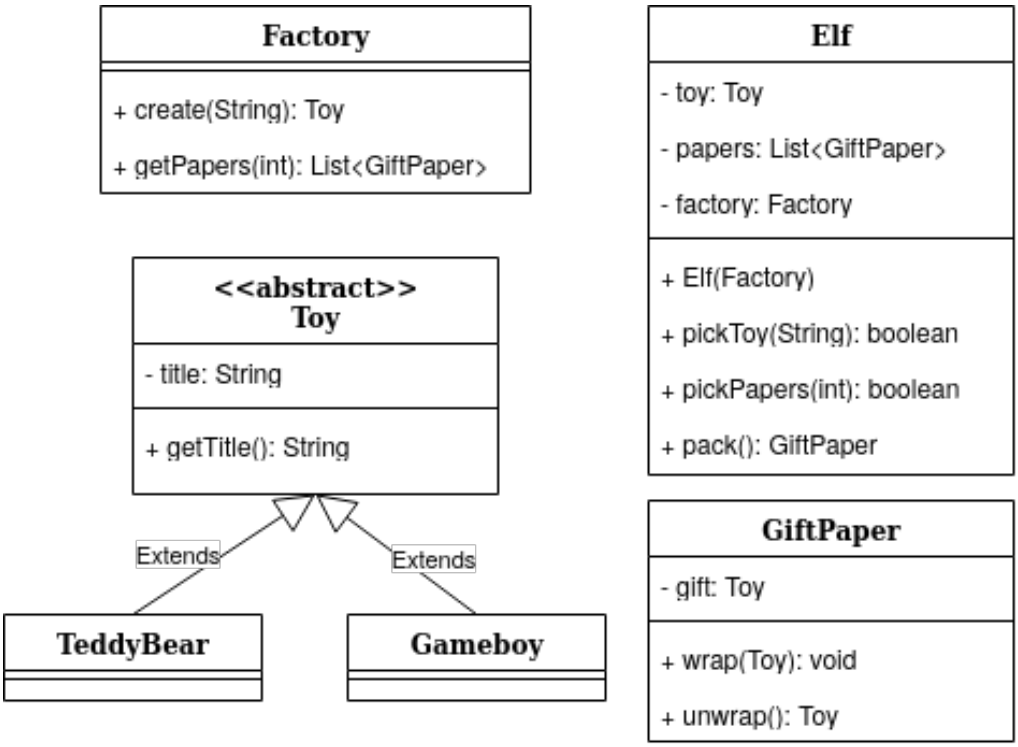
All the classes in this exercise must be in the `Factory` package.
They must have a public visibility.

The factory methods allow to encapsulate objects creations.
This is useful when the creation process is complex, when it depends on configuration files or user entries for example.

Today Santa is asking you to manage his toy factory.
Create the classes `Toy`, `TeddyBear`, `Gameboy`, `GiftPaper` and `Factory` with their attributes and methods as defined in the diagram below.
The `GiftPaper` has two methods: `wrap` that act as the attribute setter and `unwrap` that both returns the attribut and set it to null.
The `Factory` class contains a `create` method.

- If the parameter is “teddy”, it returns a `TeddyBear`;
- If it is “gameboy”, it returns a `Gameboy`;
- Else it must throw a `NoSuchToyException` with the following message: `No such toy: [toy name].`

Where *[toy name]* is the name that was passed in parameter.
Finally, the *getPapers* method of your `Factory` create a `List` containing *n* `GiftPaper`, *n* being the number in parameter.


Factory
+ create(String): Toy
+ getPapers(int): List<GiftPaper>
<<abstract>>
Toy
- title: String
+ getTitle(): String
Extends
Extends
TeddyBear
Gameboy
Elf
- toy: Toy
- papers: List<GiftPaper>
- factory: Factory
+ Elf(Factory)
+ pickToy(String): boolean
+ pickPapers(int): boolean
+ pack(): GiftPaper
GiftPaper
- gift: Toy
+ wrap(Toy): void
+ unwrap(): Toy

EXERCISE 02

Files to hand in: ./Factory/Toy.java
./Factory/TeddyBear.java
./Factory/Gameboy.java
./Factory/Factory.java
./Factory/GiftPaper.java
./Factory/NoSuchToyException.java
./Factory/Elf.java

All the classes in this exercise must be in the `Factory` package.
They must have a public visibility.

Add to your classes the `Elf` class defined in the previous diagram.

- **pickToy**
  The **Elf** will try to pick the corresponding Toy from the Factory.
  If this kind of toy doesn't exist he will say:

```
I didn't find any [toyName].
```

The **Elf** cannot get something if he already has something in his hands.
If the **Elf** can get a toy, the method must display:

```
What a nice one! I would have liked to keep it...
```

If he cannot display it:

```
Minute please?! I'm not that fast.
```

This method retuns `true` if, and only if, a new toy has been picked-up.

- **pickPapers**
  Get *nb* pieces of GiftPaper from the Factory.
  It always returns `true`.

- **pack**
  makes the **Elf** pack the Toy in his hand in a GiftPaper, return it and say:

```
And another kid will be happy!
```

If the Elf has no Toy in his hands instead it will say:

```
I don't have any toy, but hey at least it's paper!
```

If the **Elf** doesn't have **GiftPaper** anymore, it must display:

```
Wait... I can't pack it with my shirt.
```

If there's no GiftPaper, the method will return `null`.

The Elf cannot directly create objects, it **must** use the factory to do so.

EXERCISE 03

Files to hand in: ./Composite/Sentence.java
./Composite/Word.java
./Composite/SentenceComposite.java

All the classes in this exercise must be in the `Composite` package.
They must have a public visibility.

Create a `Sentence` Interface that only contains a **print** method. This method takes no parameter and returns nothing.
Create a `Word` class that implements `Sentence` and override the **print** function to display a String that was passed to its constructor.

You now have to create a composite class named `SentenceComposite` that also implements `Sentence`.
It must contain a `List<Sentence>` named *childSentence* as attribute.
Override its *print* function to make it iterate on its children to call their own *print* functions.

Finally, add two functions: **add** and **remove** which both take an `Sentence` as parameter to add or remove a child.

Here is an example:

```
public static void main(String[] args) {
    Word w1 = new Word("word1");
    Word w2 = new Word("word2");
    Word w3 = new Word("word3");
    Word w4 = new Word("word4");

    SentenceComposite sc1 = new SentenceComposite();
    SentenceComposite sc2 = new SentenceComposite();
    SentenceComposite sc3 = new SentenceComposite();

    sc1.add(w1);
    sc1.add(w2);
    sc1.add(w3);

    sc2.add(w4);

    sc3.add(sc2);
    sc3.add(sc1);
    sc3.print();
}
```

Terminal

```
~/T-JAV-500> java Example
word4
word1
word2
word3
```

EXERCISE 04

Files to hand in: ./Observer/Observable.java
./Observer/Order.java
./Observer/Observer.java
./Observer/Customer.java

All the classes in this exercise must be in the `Observer` package.
They must have a public visibility.

Let's start an application for customers to see in real time the state of the orders they placed.
We'll use the observer design pattern.

This design pattern can be used when one object has several observers, when one observer has several objects to observe or when several objects are followed by several observers.

Create an `Observable` interface with two methods:

- *addObserver* : takes an `Observer` as parameter
- *notifyObservers* : returns a `boolean`, false if the observer is null

Create a `Order` class which inherits from the `Observable` interface.
This class has four attributes: *position* (`String`), *destination* (`String`), *timeBeforeArrival* (`int`) and *observers* (`List<Observer>`, that you will create later).
Add a getter for the first three ones.

It also has a **setData** method which takes two strings and one int as parameters to set respectively the position, the destination and the time.

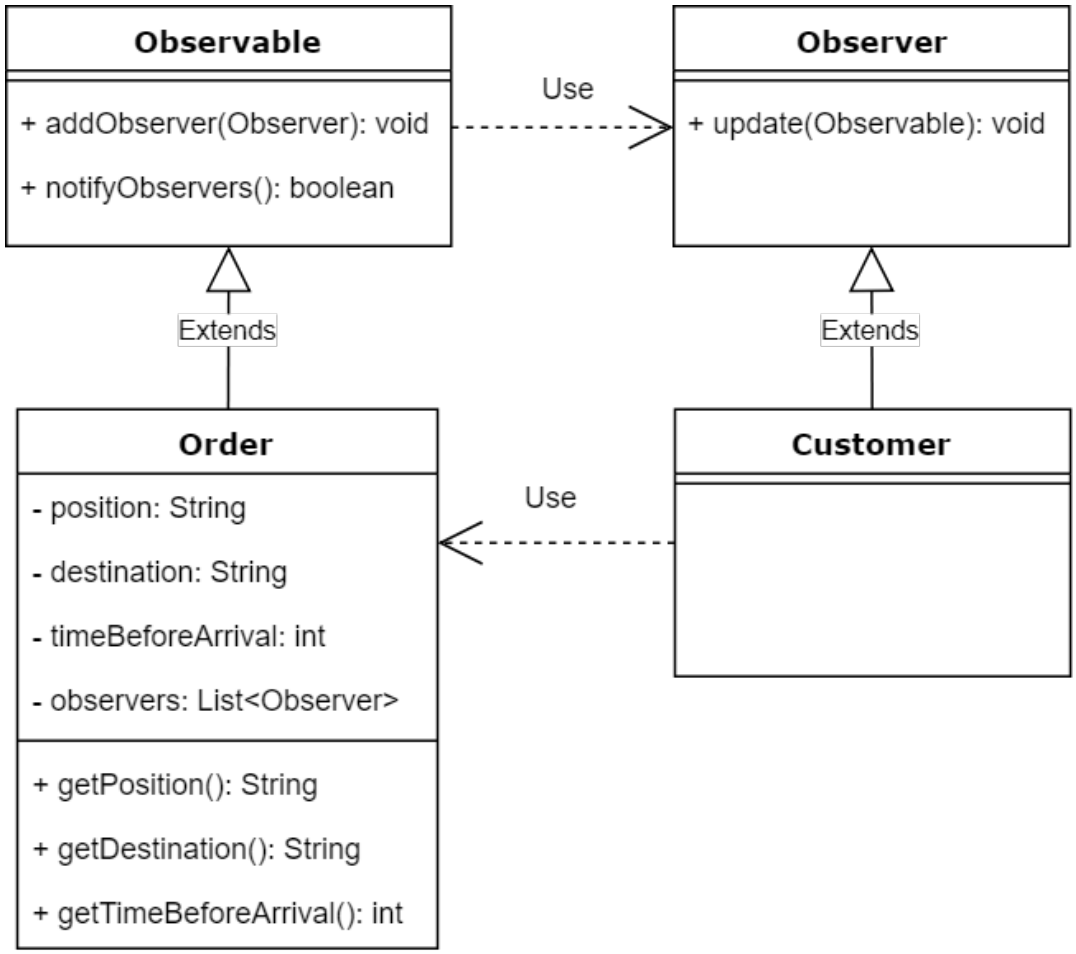
Then, create an `Observer` interface which only has one **update** method that takes an `Observable` as parameter.

Create now a `Customer` class which implements the `Observer` interface.
Its **update** method must display the delivery information the following way:

```
Position ([position]), [time] minutes before arrival at [destination].
```

Use the *instanceof* keyword.

The **notifyObservers** method of your `Order` class should calls the **update** method of its `observers`. This method must be called automatically after updating the delivery's data



Let's see an example of how it should work:

```
public static void main(String[] args) {
    Order order = new Order();
    Customer customer = new Customer();

    order.addObserver(customer);
    order.setData("123.5326, 237.9277", "6W 40th Street, New York", 10);
    order.notifyObservers();
}
```

Terminal

```
~/T-JAV-500> java Example
Position (123.5326, 237.9277), 10 minutes before arrival at 6W 40th Street, New
York.
Position (123.5326, 237.9277), 10 minutes before arrival at 6W 40th Street, New
York.
```

EXERCISE 05

Files to hand in: ./Decorator/Warrior.java
./Decorator/BasicWarrior.java
./Decorator/KingWarrior.java
./Decorator/StuffDecorator.java
./Decorator/Shield.java
./Decorator/FireSword.java

All the classes in this exercise must be in the `Decorator` package.
They must have a public visibility.

The Decorator design pattern is useful to enrich dynamically a basic class.
Let's see how to use it.

Create a **Warrior** abstract class with two attributes:

- an int, *hp*, for the health points,
- an int, *dmg*, for the damage points it causes.

Add a getter only for each attribute: *getHp*, *getDmg*.

Now create two classes **BasicWarrior** and **KingWarrior** which inherit from **Warrior**, and set the following attributes in their constructors:

```
**BasicWarrior**

hp:     40
dmg:    7
```

```
**KingWarrior**

hp:     60
dmg:    10
```

You now have to implement the decorator classes, whose goal is to add skills to our warriors.
Create the class **StuffDecorator** that inherits from **Warrior** and has a **Warrior** named *holder* as protected attribute.

Make sure your **StuffDecorator** class overrides the two getters from Warrior to return the value from `holder`.

Create a **Shield** class and a **FireSword** class which both inherit from the **StuffDecorator** class. Their constructor takes a **Warrior** as parameter to initialize their attribute *holder*.

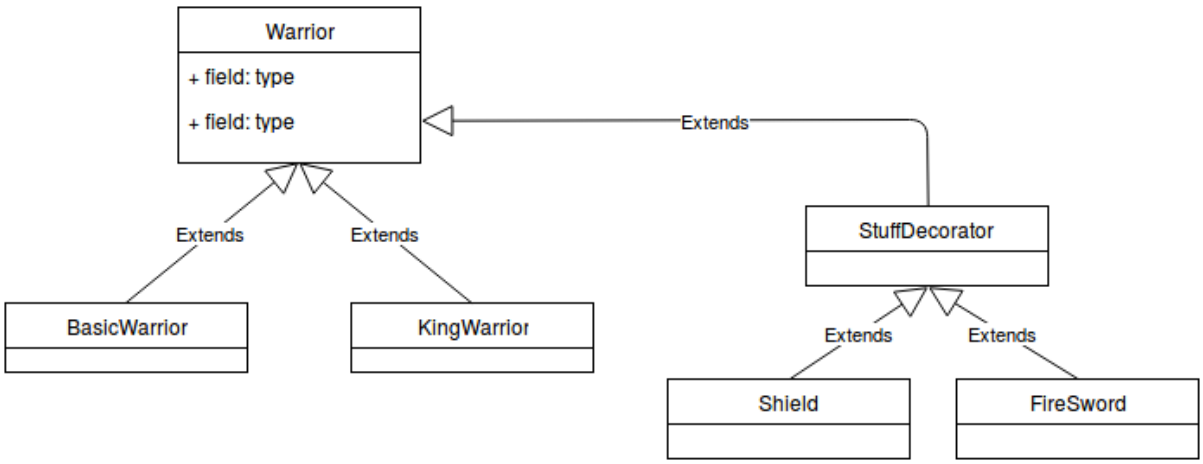
The **Shield** constructor must display:

```
May this shield protect me against every enemy.
```

And the **FireSword** displays:

```
I can slice and burn like the wind and the flames.
```

Now override the getters to make the **Shield** add 10 health points and the **FireSword** 3 damage points.



Here is an example:

```
public static void main(String[] args) {
    Warrior albert = new BasicWarrior();
    System.out.println("Albert has " + albert.getHp() + " health points.");
    albert = new Shield(albert);
    System.out.println("Albert has " + albert.getHp() + " health points.");

    Warrior georges = new KingWarrior();
    System.out.println("Georges has " + georges.getHp() + " health points and can
        hit " + georges.getDmg() + " damages.");
    georges = new FireSword(georges);
    georges = new Shield(georges);
    System.out.println("Georges has " + georges.getHp() + " health points.");
    System.out.println("Georges can hit " + georges.getDmg() + " damages.");
}
```

Terminal

```
~/T-JAV-500> java Example
Albert has 40 health points.
May this shield protect me against every enemy.
Albert has 50 health points.
Georges has 60 health points and can hit 10 damages.
I can slice and burn like the wind and the flames.
May this shield protect me against every enemy.
Georges has 70 health points.
Georges can hit 13 damages.
```

More informations

If design patterns are often handy, they are not always appropriate and are not the Alpha and Omega of OO conception.
Rather than trying to apply (even cleverly) these patterns, understanding the object model in depth is way more relevant.

To do so, here is a list of concepts and Wikipedia references you should read:

- SOLID
  Single responsibility
  a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

  Open/closed
  software entities… should be open for extension, but closed for modification

  Liskov substitution
  objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program

  Interface segregation
  many client-specific interfaces are better than one general-purpose interface

  Dependency inversion
  one should depend upon abstractions, not concretions.

Read some more about SOLID, this is the keystone of conception!

- Design by contract
- Composition over inheritance

These are advanced principles, but at some point in your Java understanding, you need to be familiar with them…