Design Patterns

Definition

Design patterns are solutions to problems commonly encountered by object-oriented developers. These design patterns are born from the different tests and experience of many developers and therefore represent the best practices used.

These are actually 'patterns' / ‘templates’ that are used to answer a particular problem and are reusable in as many object-oriented programming languages as possible.

Families of Design Patterns

Design Patterns are generally grouped into 3 big family / category:

<u>Creational Patterns</u>:

These Design Patterns allow to create objects while masking the logic of creation. They will deal with the object creation mechanism. This gives the program more flexibility in deciding which objects should be created depending on the situation.

<u>Structural Patterns</u>:

These Design Patterns deal with the composition of classes and objects. The inheritance concept is used to compose interfaces and define methods for composing objects to obtain new features.

<u>Behavioral Patterns</u>:

These Design Patterns solve problems related to interaction and communication between classes. These design patterns will therefore increase the communication flexibility between these classes.

GoF - A bit of history

In the years 1994-1995, four (4) authors published a book entitled '*Design Patterns - Elements of Reusable Object-Oriented Software*'.
This book will become THE reference for Design Patterns.

These authors are known as the *Gang of Four (GoF)*.

UML and UML class diagrams

, for Unified Modeling language, is a graphical modeling language, composed of a set of diagrams, to visualize the design of a software: its operation, its actions etc ...

The class diagrams are used in object-oriented modeling. They describe the structure of a system by modeling its classes, attributes, and the relationships between objects.

A class is represented as a rectangle. In this rectangle several things: the name of the class, the attributes and the methods.

Some Design Patterns

We will see, after this course, some design patterns, the problems they answer and their implementation in PHP.

Factory

Problem

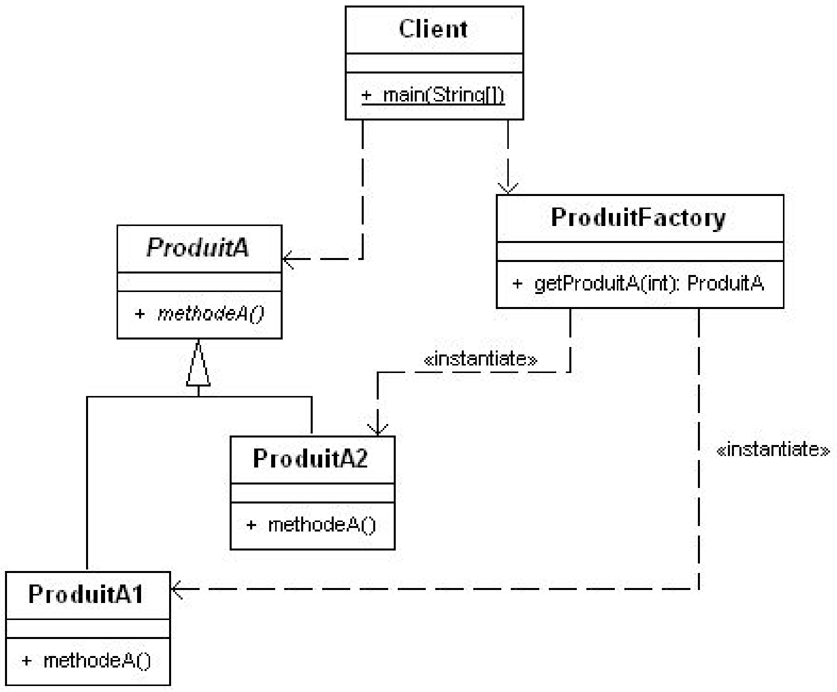
The *Factory pattern* or *Factory Method* allows object creation based on context. It is therefore used when the type of the object is not defined upstream and must be created dynamically.

For this, we will use a factory / factory that provides the objects by hiding the details of the implementation.

The objects are all from a common interface (abstract class or interface). The 'fabricated' object is therefore always of the type of the parent class and, thanks to polymorphism, the operations and methods executed are those of the created instance.

UML

Here is the class diagram of a possible implementation of the Factory design pattern:



Source: https://www.jmdoudoux.fr/java/dej/chap-design-patterns.htm

Theclass *ProductA* is the abstract class mother of all the types of products.

Theclasses *Product1* and *ProductA2* are concrete implementations of products (subclasses).

Theclass *ProductFactory* is our factory that will return the instance of the desired product class. The creation of this instance is based on the data sent to this factory.

Build a Factory in PHP

Here is an example of setting up Factory in PHP.
We use the UML diagram above.

Product classes:

```
abstract class ProductA {
        public abstract function methodA ();
}

class ProductA1 extends ProductA {
        public function methodA () {
                echo "Method A of product A1";
        }
}

class ProductA2 extends ProductA {
        public function methodA () {
                echo "Method A2 product A2";
        }
}
```

Our factory:

```
class ProductFactory {
        public static function createProductA (string $
productType): ProductA {
                $ productA = null;

                switch ($ productType) {
                        case "A1":
                                $ productA = new ProductA1 ();
                                break;
                        case "A2":
                                $ productA = new ProductA2 ();
                                break;
                        default:
                                throw new InvalidArgumentException
("Unknown Product Type");
                }

                return $ productA;
        }
}
```

Example of using our factory:

```
$ product = ProductFactory :: createProductA ('A1');
$ product-> Methoda ();
$ product = ProductFactory :: createProductA ('A2');
$ product-> Methoda ();
```

Singleton

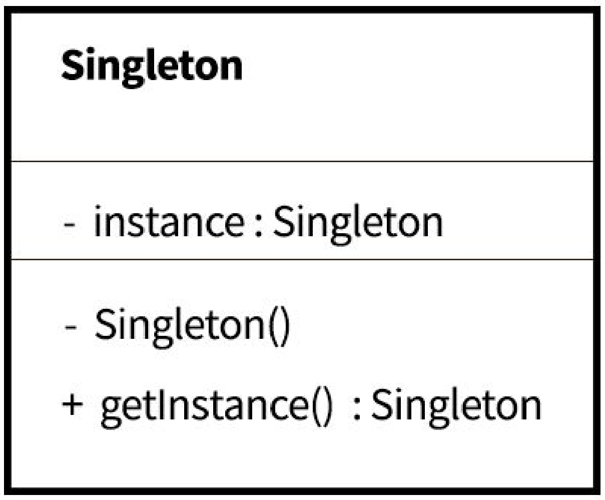
Problem

The Singleton design pattern is used when there is a need for a single instance of a class in your application. It must also provide a 'global' access point to this instance.

Singleton is used when you have a unique service within your application. For example: database connection, centralized logging system, configuration management etc ...

UML

Here is the class diagram of the Singleton design pattern:



Building a Singleton in PHP

To validate the Singleton, there are several things to respect:

1. It is necessary to guarantee the singleton uniqueness. For this, we must prohibit instantiation with the operator 'new'. To do this, simply declare a 'private' constructor.
   It will also be necessary to think of putting the methods __clone () and __wakeup () in 'private' also.
2. So, to recover an instance of this class, you will have to go through an intermediate method instead of using the constructor.
   This method will be static since there is still no reference to the instance of the class at the time of the call.
3. Because the method is static, the single instance of the class will be static as well.

Here is an implementation of Singleton:

```
class Singleton {

        private static $ instance;

        public static function getInstance (): Singleton {
                if (static :: $ instance === null) {
                        static :: $ instance = new static ();
                }

                return static :: $ instance;
        }

        private function __construct () {}
        private function __clone () {}
        private function __wakeup () {}
}
```