# Lecture 13: Design Patterns

## What is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such away that you can use this solution a million of times over, without ever doing it the same way twice". Even though Alexander was talking about patterns in buildings and towns, *what he says is true about object-oriented design patterns*.

### *Design Pattern Descriptions*

- A name
- A problem to guide when to apply the pattern
- A solution describes the elements that make up the pattern
  - Relationships
  - Collaborations
  - Responsibilities
- Consequences of applying the pattern and trade-offs
- A pattern should also include
  - An example
  - At least three known uses

According to the GoF Book we have three kinds of patterns:

- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Proxy
- Behavioural Patterns
  - Chain of Responsibility
  - Command
  - Interpretor Mediator
  - Observer
  - State
  - Strategy
  - Template Method

*Beware, Many other patterns exist!*

# 1. Creational Patterns

## *1.1. Factory Method Pattern*

The essence of the Factory Method Pattern is to define an interface for creating an object, but let the classes which implement the interface decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses.

In object-oriented programming, a factory is an object for creating other objects. It is an abstraction of a constructor, and can be used to implement various allocation schemes.

A factory object typically has a method for every kind of object it is capable of creating. These methods optionally accept parameters defining how the object is created and return the object created.

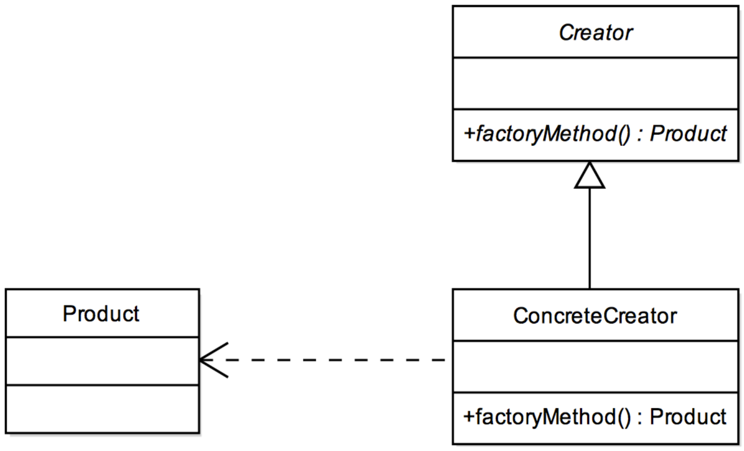
### Common Usage

Factory methods are common in toolkits and frameworks where library code needs to create objects of types which may be subclassed by applications using the framework.

### Applicability

The factory pattern can be used when:

- The creation of an object precludes its reuse without significant duplication of code.
- The creation of an object requires access to information or resources that should not be contained within the composing class.
- The lifetime management of the generated objects must be centralised to ensure a consistent behaviour within the application.



## Example:

Consider as an example a program to read image files and make thumbnails from them. The program supports different image formats, represented by a *read* class for each format:

```
public interface ImageReader {
   public DecodedImage getDecodedImage();
}
public class GifReader implements ImageReader {
   public DecodedImage getDecodedImage() {
      // ...
      return decodedImage;
   }
}
public class JpegReader implements ImageReader {
   public DecodedImage getDecodedImage() {
      // ...
      return decodedImage;
   }
}
```

Each time the program reads an image it needs to create a reader of the appropriate type based on some information in the file. This logic can be encapsulated in a factory method:

```
public class ImageReaderFactory {
   public static ImageReader getImageReader(InputStream is) {
      int imageType = determineImageType(is);

      switch(imageType) {
         case ImageReaderFactory.GIF:
            return new GifReader(is);
         case ImageReaderFactory.JPEG:
            return new JpegReader(is);
         // etc.
      }
   }
}
```

## 2. Structural Patterns

### *2.1. Adapter Pattern*

In computer programming, the adapter pattern (often referred to as the wrapper pattern or simply a wrapper) is a design pattern that translates one interface for a class into a compatible interface.

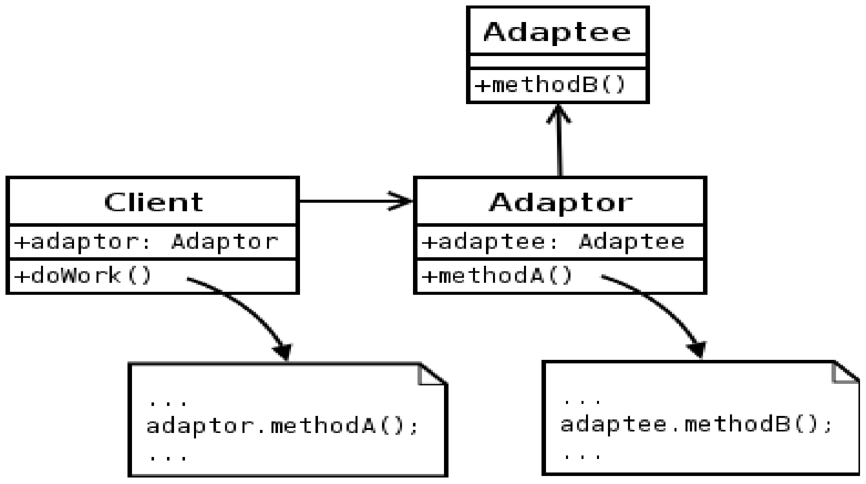
An adapter allows classes to work together that normally could not because of incompatible interfaces, by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface, and the amount of code necessary to do this is typically small. The adapter is also responsible for transforming data into appropriate forms. For instance, converting the format of dates (e.g. YYYYMMDD to MM/DD/YYYY or DD/MM/YYYY).

### *Structure*

There are two ways of adapter patterns:

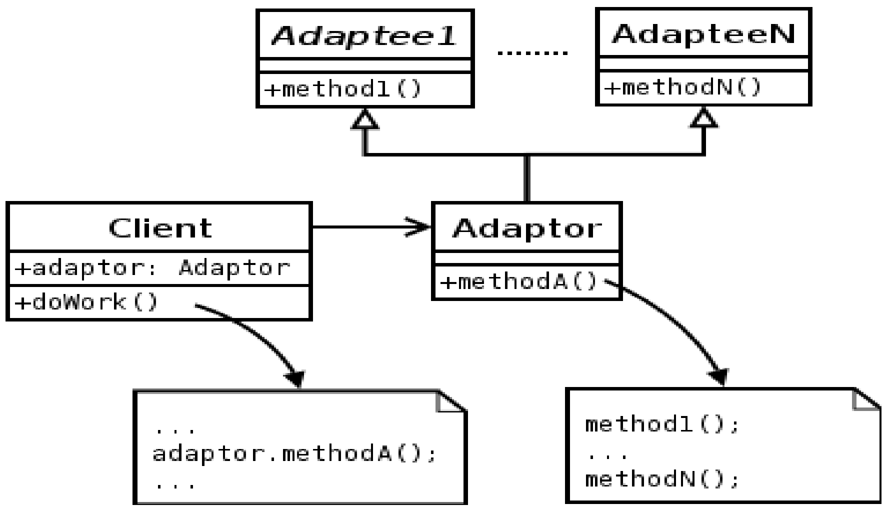
#### Object Adapter Pattern

In this type of adapter, the adapter contains an instance of the class it wraps. In this situation, the adapter makes calls to the instance of the wrapped object.



#### Class Adapter Pattern

This type of adapter uses multiple polymorphic interfaces to achieve its goal. The adapter is created by implementing or inheriting both the interface that is expected and the interface that is pre-existing. It is typical for the expected interface to be created as a pure interface class, especially in languages such as Java that do not support multiple inheritance.



The adapter pattern is useful in situations where an already existing class provides some or all of the services you need but does not use the interface you need. A good real life example is an adapter that converts the interface of a Document Object Model (DOM) of an XML document into a tree structure that can be displayed.

## Examples

The famous adapter classes in Java API are WindowAdapter, ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter and MouseMotionAdapter.

As you know, WindowListner interface has seven methods. Whenever your class implements such interface, you have to implement all of the seven methods. WindowAdapter class implements WindowListener interface and make seven empty implementation. When your class extends WindowAdapter class, you may choose the method you want without restrictions.

```
public interface Windowlistener {
   public void windowClosed(WindowEvent e);
   public void windowOpened(WindowEvent e);
   public void windowIconified(WindowEvent e);
   public void windowDeiconified(WindowEvent e);
   public void windowActivated(WindowEvent e);
   public void windowDeactivated(WindowEvent e);
   public void windowClosing(WindowEvent e);
}
```

```
public class WindowAdapter implements WindowListner{
    public void windowClosed(WindowEvent e){}
    public void windowOpened(WindowEvent e){}
    public void windowIconified(WindowEvent e){}
    public void windowDeiconified(WindowEvent e){}
    public void windowActivated(WindowEvent e){}
    public void windowDeactivated(WindowEvent e){}
    public void windowClosing(WindowEvent e){}
}
```

By composition, we can achieve adapter pattern. It is also called wrapper. For example, a Data class has already been designed and well tested. You want to adapt such class to your system. You may declare it as a variable and wrap or embed it into your class.

```
//well-tested class
class Data {
   public void add(Info){}
   public void delete(Info) {}
   public void modify(Info){}
   //...
}
//Use Data class in your own class
class AdaptData {
   Data data;
   public void add(Info i) {
       data.add(i);
       //more job
   }
   public void delete(Info i) {
      data.delete(i);
      //more job
   }
   public void modify(Info i) {
      data.modify(i);
      //more job
   }
   //more stuff here
   //...
}
```