# Patrons de Disseny

*En aquest document es descriuen idees generals sobre els* ***patrons de disseny****.*

*El material ha estat elaborat a partir de informació diversa trobada a la WEB, basada en el llibre:*

**Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (*Gang of Four: GoF*)**
***Patrones de diseño. Elementos de software orientado a objetos reutilizable***
Pearson Educación, 2003

**Toni Sellarès**

## What is Design Patterns?

Design patterns are recurring solutions to software design problems you find again and again in real-world application development.

Design patterns are about design and interaction of objects, as well as providing a communication platform concerning elegant, reusable solutions to commonly encountered programming challenges.

A pattern is a solution to a problem in a context, it documents in an abstract and compact form

- the problem
- the context in which it occurs
- a good solution

and it embodies wisdom about how the solution addresses the problem.

## Why Design Patterns ?

- **Experience**

    - good designs require experienced designers/architects.
    - reusing solutions that have worked in the past instead of creating everything new from basic principles.
    - learning from other people’s experience.

- **Vocabulary, names for design patterns**

    - designing and thinking yourself on a higher abstraction level communicating about designs and design patterns with other.

- **More than just OO**

    - finding good classes with right granularity, finding right interaction patterns for a specific task at hand.
    - systems for the future do not represent just current reality: how to find the future-oriented concepts?

## Describing Design Patterns

A commonly used format is the one used by the **Gang of Four**.

It contains the following sections:

- **Pattern Name and Classification:** Every pattern should have a descriptive and unique name that helps in identifying and referring to it. Additionally, the pattern should be classified according to a classification such as the one described earlier. This classification helps in identifying the use of the pattern.
- **Intent:** This section should describe the goal behind the pattern and the reason for using it. It resembles the problem part of the pattern.
- **Also Known As:** A pattern could have more than one name. These names should be documented in this section.
- **Motivation:** This section provides a scenario consisting of a problem and a context in which this pattern can be used. By relating the problem and the context, this section shows when this pattern is used.
- **Applicability:** This section includes situations in which this pattern is usable. It represents the context part of the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams can be used for this purpose.
- **Participants:** A listing of the classes and objects used in this pattern and their roles in the design.
- **Collaboration:** Describes how classes and objects used in the pattern interact with each other.
- **Consequences:** This section describes the results, side effects, and trade offs caused by using this pattern.
- **Implementation:** This section describes the implementation of the pattern, and represents the solution part of the pattern. It provides the techniques used in implementing this pattern, and suggests ways for this implementation.
- **Sample Code:** An illustration of how this pattern can be used in a programming language
- **Known Uses:** This section includes examples of real usages of this pattern.
- **Related Patterns:** This section includes other patterns that have some relation with this pattern, so that they can be used along with this pattern, or instead of this pattern. It also includes the differences this pattern has with similar patterns.

## Do patterns replace thinking ?

- Patterns represent knowledge, yet this knowledge still has to be acquired, even if the acquisition is made easy
- Patterns have to be tailored to a concrete problem, and implemented
- Choosing among patterns, and combining them, is no automatic process!

## Type of Patterns

They are categorized in three groups:

- **Creational Patterns.** Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object.

- **Structural Patterns**. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects.

- **Behavioural Patterns**. The Behavioural class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioural object patterns describe how a group of objects co-operate to perform a task that no single object can carry out alone.

## Catalog of Design Patterns

The classification is based on two criteria: type and scope.

The scope criterion groups the patterns in class and object patterns. Class patterns are based on relationships between classes, mainly inheritance structures. Object patterns dynamically let objects reference each other.

| **Scope \ Purpose** | **Creational** | **Structural** | **Behavioural** |
|---|---|---|---|
| **Class** | Factory Method | Adapter (class) | Interpreter<br>Template method |
| **Object** | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Façade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

**Classification in Gamma et al. 1995.**

## SUMMARY OF CREATIONAL PATTERNS

**Factory Method.** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Abstract Factory.** Provide an interface for creating families of related or dependent objects without specifying their concrete classes

**Builder.** Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**Prototype.** Specify the kinds of objects to create using a prototypical instance, and create new  objects by copying this prototype.

**Singleton.** Ensure a class has only on instance, and provide a global point of access to it.

## SUMMARY OF STRUCTURAL PATTERNS

**Adapter.** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

**Bridge.** Decouple an abstraction from its implementation so that the two can vary independently.

**Composite.** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.

**Decorador.** Attach additional responsibilities to an object dynamically. Decorator provides a flexibility to sub classing for extending functionality.

**Façade –** Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

**Flyweight.** Use sharing to support large numbers of fine-grained objects efficiently.

**Proxy.** Provide a surrogate or placeholder for another object to control access to it.

## SUMMARY OF BEHAVIOURAL PATTERNS

**Chain of Responsibility.** Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle request. Chain the receiving objects and pass the requests along the chain until an object handles it.

**Command.** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. Command de-couples the object that invokes the operation from the one that knows how to perform it.

**Interpreter.** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. Useful in designing language related applications and compilers.

**Iterator.** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Mediator.** Define an object that encapsulates how a set of objects interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

**Memento.** Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later. A memento is an object that stores a snapshot of the internal state of another object.

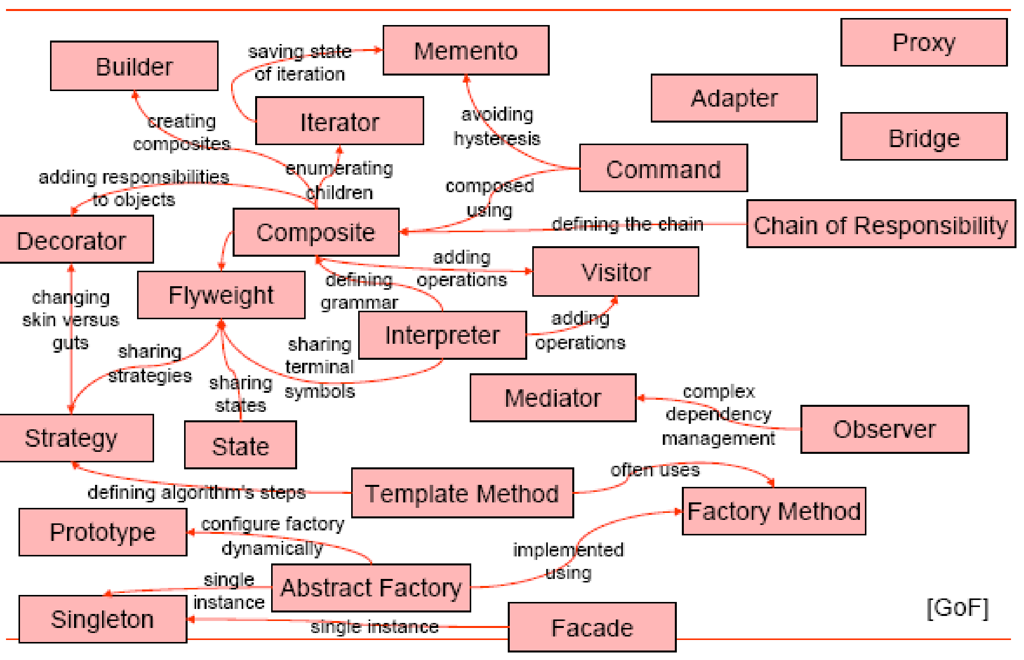
**Observer.** Define a one-to-many dependency between objects so that when one-object changes state, all its dependents are notified and updated automatically.

**State.** Allow an object to alter its behavior when its internal state changes. The object will appear to change its classes.

**Strategy.** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets algorithm vary independently from clients that use it.

**Template Method.** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm. This pattern is so fundamental that it is found in almost every abstract class.

**Visitor.** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



**Design Patterns Relationships**

## How to use Design Patterns

1) Understand problem and context

- What is the problem? What is the context?
- What is the abstraction of the problem?
- Deals it with the creation, structure or behavior of objects?

2) Find a pattern

- If no pattern: find a non-pattern solution

3) Map structure

4) Evaluate fit

- benefits outweigh consequences
- check principles
- If bad fit: go back and try alternative pattern

5) Implement and evaluate result

- If result does not satisfy: go back and try alternative pattern, or implement non-pattern solution

## Enable design for change

As an application grows it often becomes too complex, necessitating a redesign.

| **Causes for redesign** | **Solution *(Patterns)*** |
|---|---|
| Objects are created by specifying their class name, inhibiting flexibility of which subclass is created | Create object indirectly<br>*(Abstract factory, Method factory, Prototype)* |
| Dependence on specific operations | Limit specific operations dependence<br>*(Chain of Responsability, Command)* |
| Dependence of specific platforms | Limit platform dependencies<br>*(Abstract factory, Bridge)* |
| Dependence on object representation. | Encapsulation of implementation details fro the clients<br>*(Abstract factory, Bridge, Memento, Proxy)* |
| Algorithmic dependencies when algorithms are likely to be changed, extended, replaced during development and reuse | Isolate algorithm<br>*(Builder, Iterator, Strategy, Template Method, Visitor)* |
| Tight coupling | Make the classes loosely coupled<br>*(Abstract factory, Bridge, Chain of Responsability, Command, Facade, Mediator, Observer)* |
| Extending functionalities with subclasses | Avoid too many levels of subclassing<br>*(Bridge, Chain of Responsability, Composite, Decorator, Observer, Strategy)* |
| Adapting legacy code and interfaces when legacy source is either not available or too many classes need to be changed with unknown side-effects | Use adapter classes<br>*(Adapter, Decorator, Visitor)* |

**WEBS**

http://en.wikipedia.org/wiki/Design_pattern_(computer_science)

http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html