

The background is a dark blue gradient. On the left, there is a large, semi-transparent circular image of a circuit board. Overlaid on this and the background are several geometric shapes: a blue parallelogram and a green parallelogram in the upper left, and a series of white, stepped, geometric lines in the upper right.

Design Patterns

Esther Jorge Paramio
alu0101102498@ull.edu.es

Manuel Andrés Carrera Galafate
alu0101132020@ull.edu.es

Index

1. Introduction
 - Design patterns definition.
 - History.
2. Why design patterns?
3. Classification of design patterns.
4. Observations
5. References.



Introduction

There are a lot of problems that we find over and over again.

Also, the problems that we find probably have been found before by someone else and also been solved already.




Definition of design pattern.

Reusable solution to a commonly occurring problem within a given context.

“Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.”

Patterns can be applied to many different areas of human endeavor.

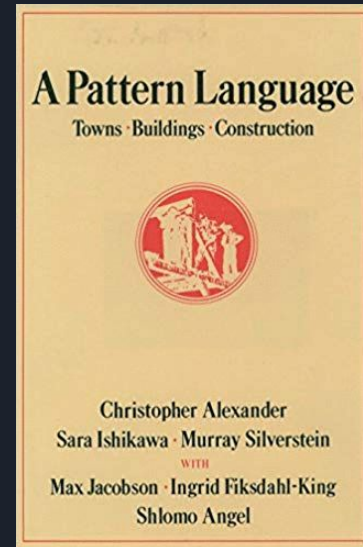




“Each pattern describes a problem which occurs over and over again in our environment, and then **describes** the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” - Christopher Alexander

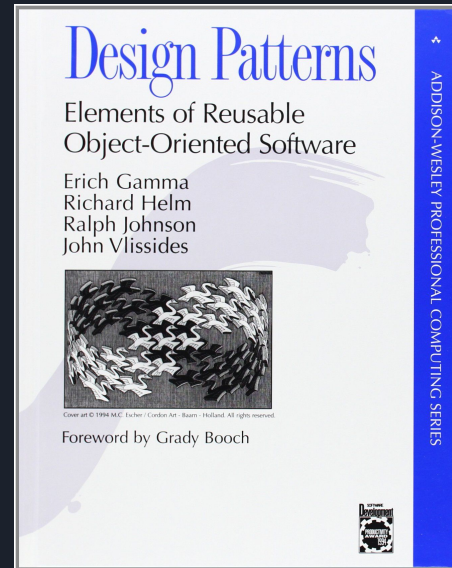
History of design patterns.

Patterns originated as an architectural concept by Christopher Alexander as early as 1977



History of design patterns.

From this initial idea Eric Gamma, Richard Helm, Ralph Johnson and Jhon Vlissides (also known as the Gang of Four) came up with the idea of applying this concept into Software Development





Why design patterns?



Why design patterns?

They provide you with a way to solve issues related to software development using a proven solution.

Makes the communication between designers more efficient.

Makes it easier to reuse successful designs and avoid alternatives that diminish reusability.

Productivity and endeavor efficiency



Classification of design patterns



Classification of design patterns

- Three groups:
 - **Creation patterns**, provide object creation mechanisms that increase flexibility and reuse of existing code.
 - **Structural patterns**, explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
 - **Behavioral patterns**, take care of effective communication and the assignment of responsibilities between objects.



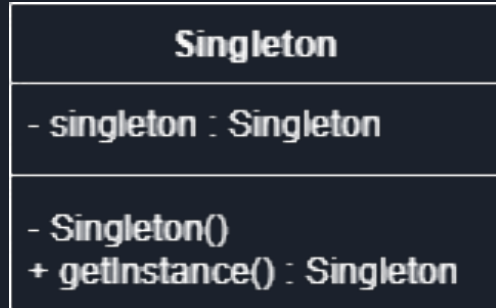
Singleton

- Definition
 - Ensure a class has only one instance and provide a global point of access to it.
- Advantages
 - It guarantees that a class has only one instance.
 - Instance control.
 - Flexibility
- Disadvantages
 - Cannot abuse.



Singleton

- When to use it?
 - When there must be exactly one instance of a class
 - When the sole instance should be extensible through inheritance.
- UML



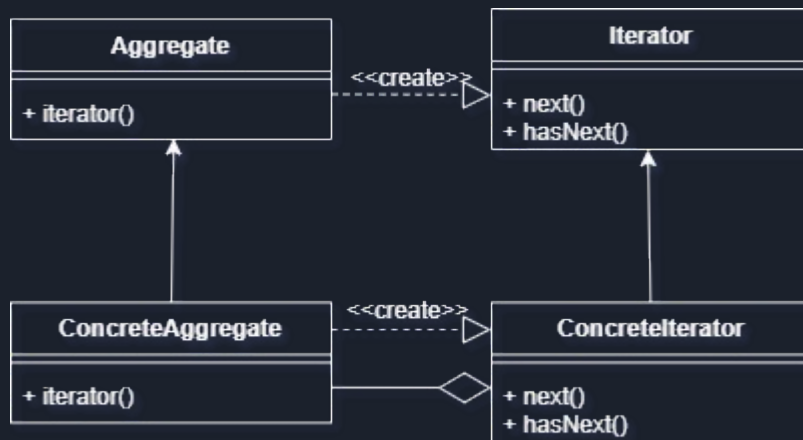


Iterator

- Definition
 - Accesses the elements of an object sequentially without exposing its underlying representation.
- Advantages
 - Access the elements of a collection object in sequential manner without any need to know it's underlying implementation.
 - The distribution of responsibilities is emphasized.
 - Parallelism and concurrency are facilitated.

Iterator

- When to use?
 - When you want to allow multiple routes in aggregate objects through a uniform interface, that is, polymorphic iteration.
- UML



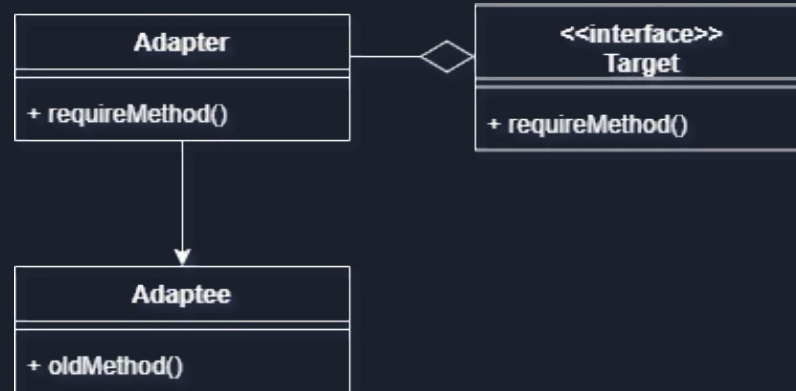


Adapter

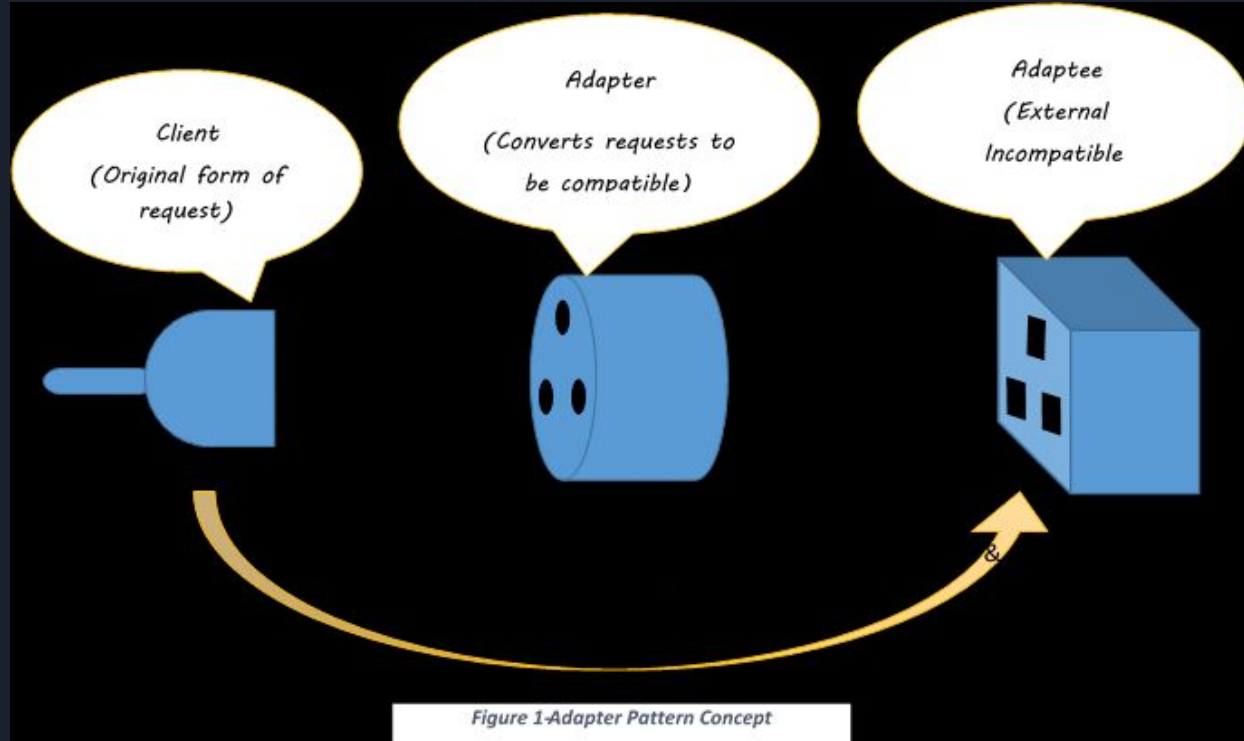
- Definition
 - Allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- Advantages
 - Allows the interface of an existing class to be used from another interface.
 - Simplicity.
- Disadvantages
 - Inflexibility.

Adapter

- When to use?
 - When you want to make existing incompatible classes work with others without modifying their source code.
- UML



Adapter





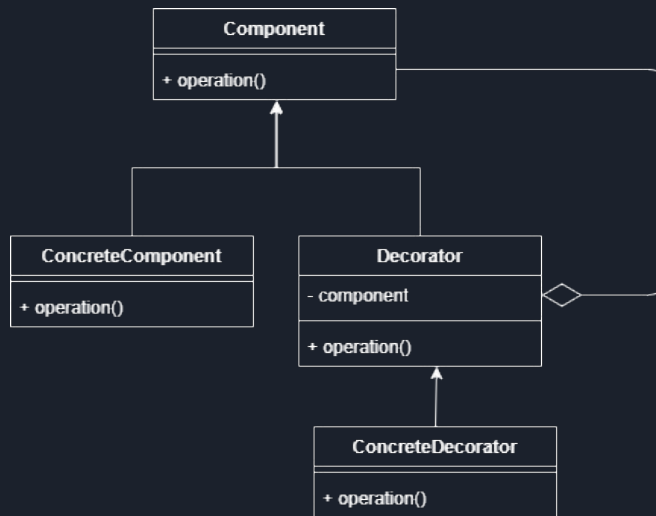
Decorator

- Definition
 - Dynamically adds/overrides behaviour in an existing method of an object.
- Advantages
 - Provide a surrogate or placeholder for another object to control access to it.
 - Add a wrapper and delegation to protect the real component from undue complexity.
 - More flexible than inheritance.
 - Non-complex classes.
- Disadvantages
 - A decorator and its components are not identical.
 - Many small objects.

Decorator

- When to use?
 - When you need to support resource-hungry objects, and you don't want to instantiate such objects unless and until they are requested by the client.
 - When it cannot be inherit or it is not practical.
 - When you want to add responsibilities to other objects dynamically.

- UML



Decorator



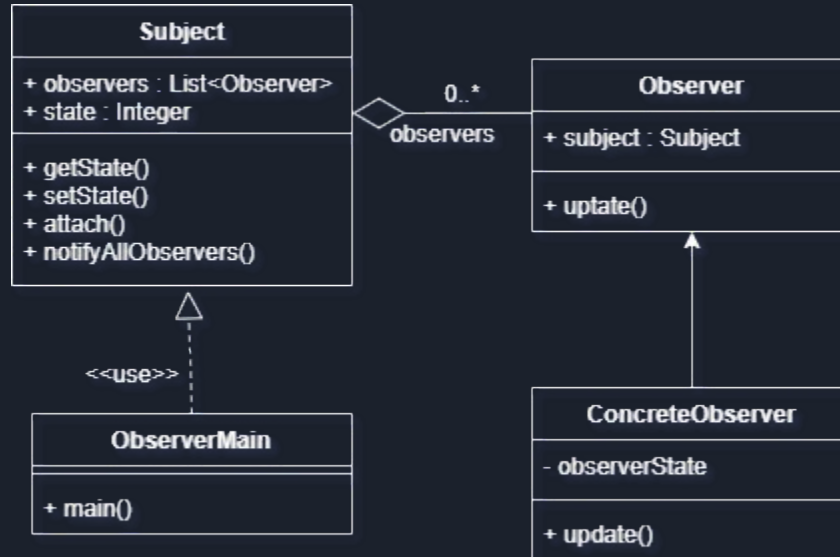


Observer

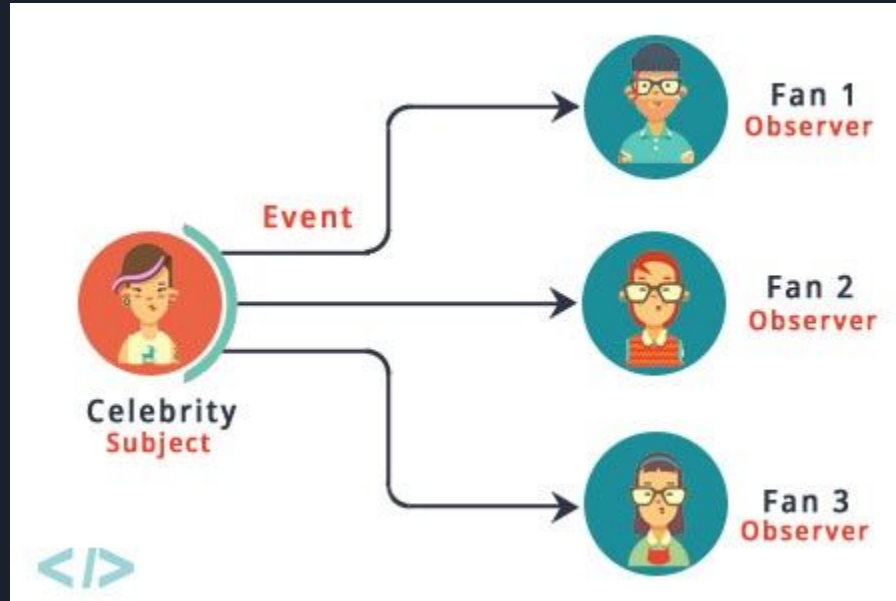
- Definition
 - Is a publish/subscribe pattern which allows a number of observer objects to see an event.
- Advantages
 - Automatic update of dependant objects.
 - Defines a one-to-many dependency between objects.
- Disadvantages
 - Very inefficient cascading unexpected updates could occur.

Observer

- When to use?
 - When a change to an object requires changing others.
 - When an object should be able to notify others.
- UML



Observer



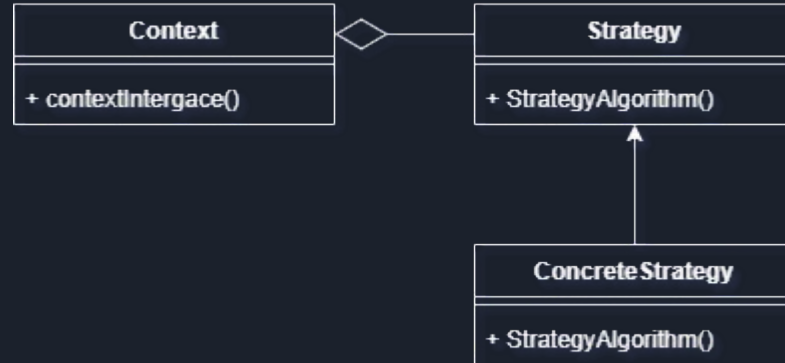


Strategy

- Definition
 - Allows one of a family of algorithms to be selected on-the-fly at runtime.
- Advantages
 - Reusable.
 - Costly multi-conditional behavior definitions are eliminated.
 - It is possible to offer different implementations of the same behavior.
- Disadvantages
 - Clients must have some knowledge of each strategy.
 - Useless information.

Strategy

- When to use?
 - When many related classes differ only in their behavior.
 - When different variants of the same algorithm are required.
 - When an algorithm uses data that customers don't have to know.
- UML





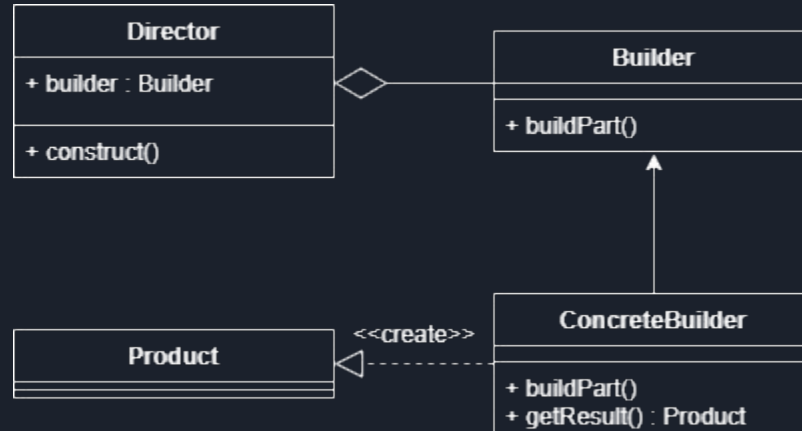
Builder

- Definition
 - Separate the construction of a complex object from its representation, allowing the same construction process to create various representations
- Advantages
 - Reduces coupling.
 - It allows to vary the internal representation of the object, respecting the builder class. In other words, we managed to make the construction of representation independent.
- Disadvantages
 - Increase the complexity of the code.

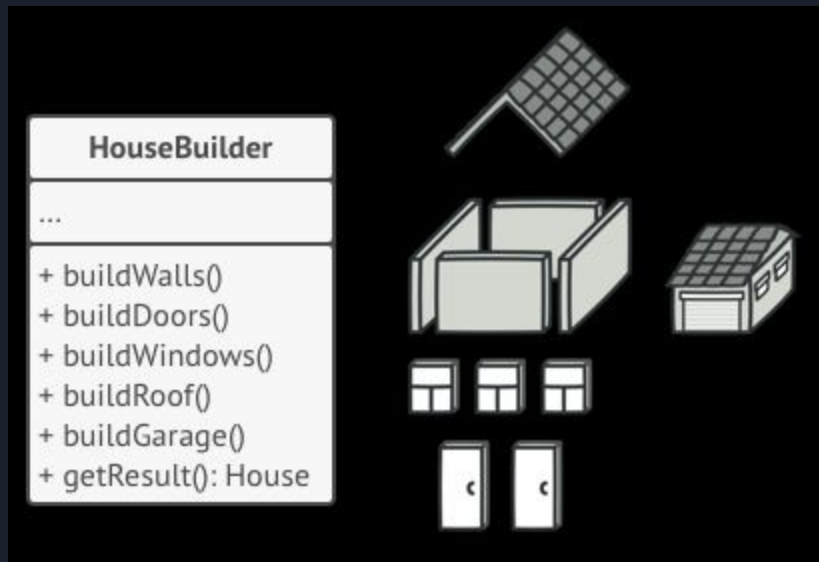
Builder

- When to use?
 - When our system deals with complex objects (made up of many attributes) but the number of configurations is limited.
 - When the algorithm for creating the complex object can be independent of its component parts and their assembly.

- UML



Builder



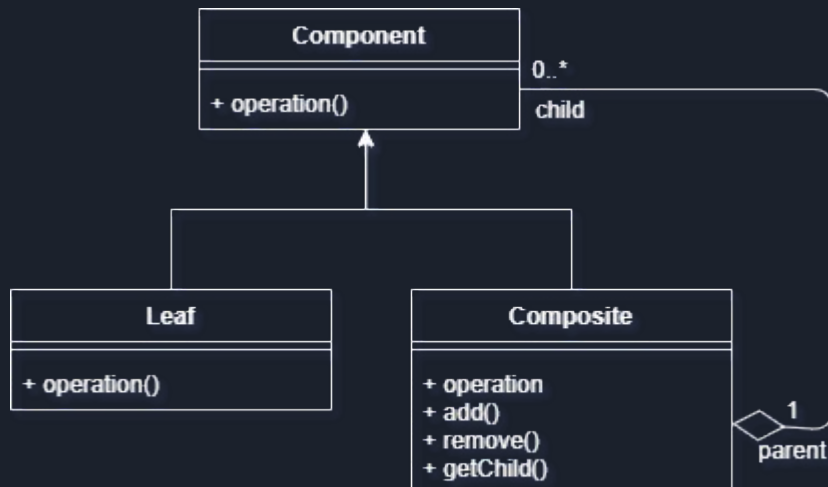


Composite

- Definition
 - Composes zero-or-more similar objects so that they can be manipulated as one object.
- Advantages
 - Allows treating complex and primitive objects uniformly.
 - Clients can treat both individual objects as well as object compositions uniformly.
 - New components can be easily added.
- Disadvantages
 - Too general design.

Composite

- When to use?
 - When you want to represent part-whole hierarchies of objects.
 - When you want clients to be able to ignore the difference between compositions of objects and individual objects.
- UML



Composite





Observations

It's important that you know that this exist, but...

Design patterns have to be properly been used.

Antipatterns.

Our goal.



Any Questions?



References:

Wikipedia: https://en.wikipedia.org/wiki/Software_design_pattern

Introduction to design patterns:

<http://community.wvu.edu/~hhammar/rts/adv%20rts/design%20patterns%20tutorials/IntroToDP-2pp.pdf>

Explanation of estrategy design pattern (spanish): <https://www.youtube.com/watch?v=VQ8V0ym2JSo>

Explanation of abstract factory design pattern (spanish): <https://www.youtube.com/watch?v=CVlpjFJN17U>

Little summary on design patterns history: https://youtu.be/uCI5qUTj_2Q

Creation design pattern: https://en.wikipedia.org/wiki/Creational_pattern#Definition

Builder example: <https://medium.com/better-programming/the-builder-pattern-in-javascript-6f3d85c3ae4a>

Composition example: <https://jsmanifest.com/the-composite-pattern-in-javascript/>

Singleton example: <https://www.sitepoint.com/javascript-design-patterns-singleton/>