Design Patterns

Esther Jorge Paramio
alu0101102498@ull.edu.es

Manuel Andrés Carrera Galafate
alu0101132020@ull.edu.es

Index

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.

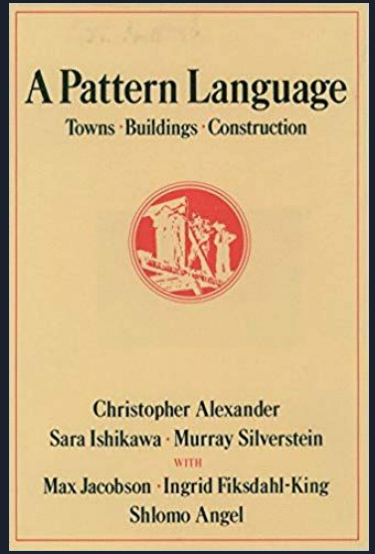
“Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.”

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

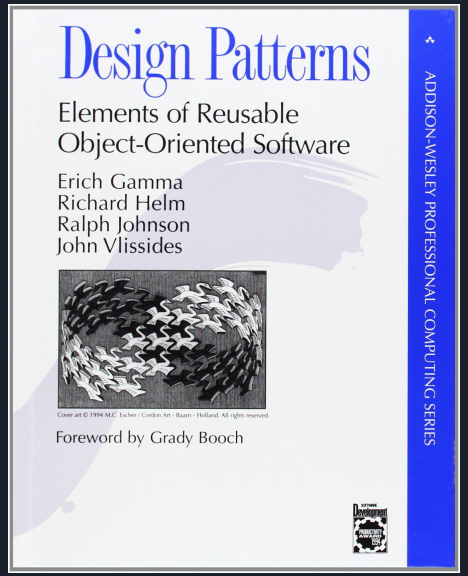
History of design patterns.

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 Jhonson 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.

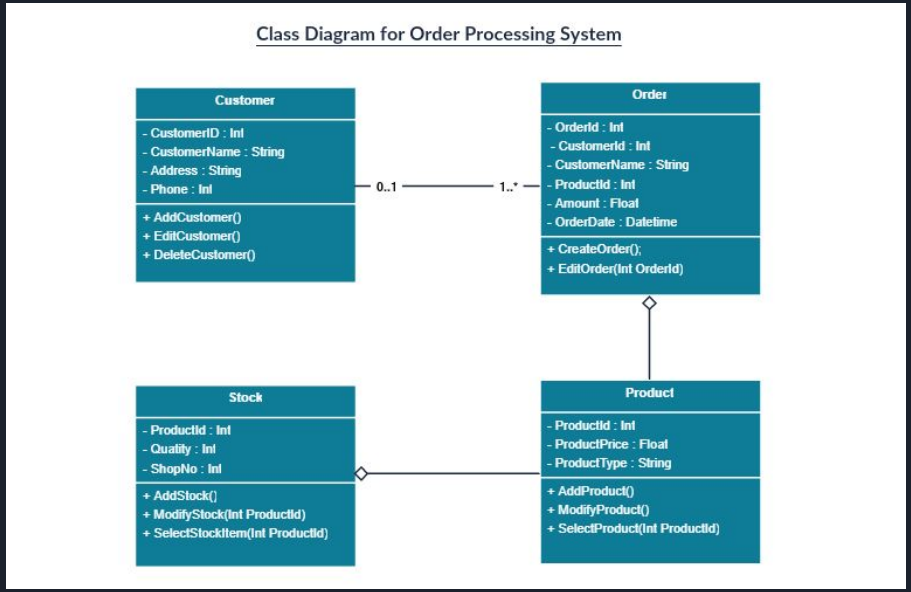
Makes it easier to reuse successful designs and avoid alternatives that diminish reusability.

UML diagrams

A UML diagram is a diagram based on the UML (Unified Modeling Language) with the purpose of visually representing a system.

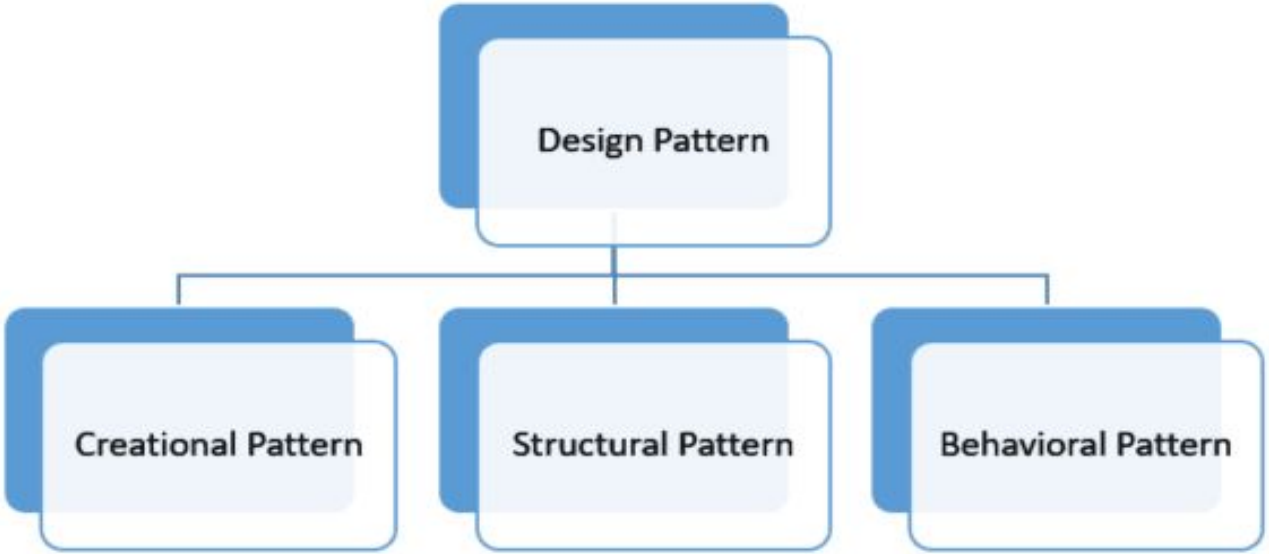
On design patterns UML class diagrams are the most relevant.

Example


Class Diagram for Order Processing System
Customer
- CustomerID : Int
- CustomerName : String
- Address : String
- Phone : Int
+ AddCustomer()
+ EditCustomer()
+ DeleteCustomer()
0..1
1..*
Order
- OrderId : Int
- CustomerId : Int
- CustomerName : String
- ProductId : Int
- Amount : Float
- OrderDate : Datetime
+ CreateOrder();
+ EditOrder(Int OrderId)
Stock
- ProductId : Int
- Quatity : Int
- ShopNo : Int
+ AddStock()
+ ModifyStock(Int ProductId)
+ SelectStockItem(Int ProductId)
Product
- ProductId : Int
- ProductPrice : Float
- ProductType : String
+ AddProduct()
+ ModifyProduct()
+ SelectProduct(Int ProductId)

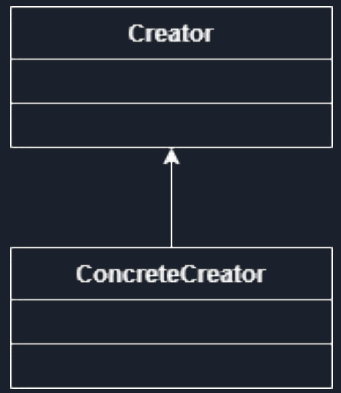
Classification of design patterns

Classification of design patterns



Classification of design patterns

- ***Creation patterns***, provide object creation mechanisms that increase flexibility and reuse of existing code. We can apply it when:
  - A system should be independent of how its objects and products are created.
  - Constructing different representation of independent complex objects.
  - The class instantiations are specified at run-time.



Classification of design patterns

- We have differents types of enemies:

```
class Boo implements Entity {

}
```

```
class Koopa implements Entity {

}
```

```
class Goomba implements Entity {

}
```

Classification of design patterns

- All of them inherit from the entity class:

```
interface Entity {
  updateLogic(): void;
}
```

```
class Boo implements Entity {

}
```

```
class Koopa implements Entity {

}
```

```
class Goomba implements Entity {

}
```

Classification of design patterns

- We will implement the logic necessary to make decision randomly, for example, with random numbers:

```
function gameLogic() {
  ...
  //More code above
  if (shouldSpawnEnemy()) {
    let randomNum = Math.random();
    let enemy;
    if (randomNum > 0.66) {
        enemy = new Koopa();
    } else if (randomNum > 0.33) {
        enemy = new Goomba();
    } else {
        enemy = new Boo();
    }
  }
  ...
  //More code below, use enemy
}
```

Classification of design patterns

- We can create two abstract classes, one that randomly selects what type of enemy is going to appear and another that randomly generates the level of difficulty that this enemy is going to have:

```
interface EnemyFactory {
  createEnemy(): Entity
}
```

```
class RandomEnemyFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns enemies randomly
  }
}
```

```
class RandomDifficultEnemyFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns only difficult enemies
  }
}
```

Classification of design patterns

- Even if we want create another class, we simply inherit from the parent class again:

```
interface EnemyFactory {
  createEnemy(): Entity
}
```

```
class RandomEnemyFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns enemies randomly
  }
}
```

```
class RandomDifficultEnemyFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns only difficult enemies
  }
}
```

```
class GoombaFactory implements EnemyFactory {
  createEnemy(): Entity {
    //returns only goombas
  }
}
```

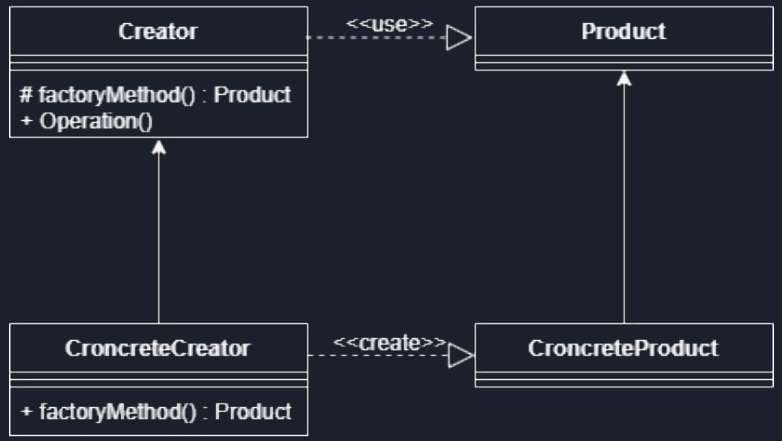
Classification of design patterns

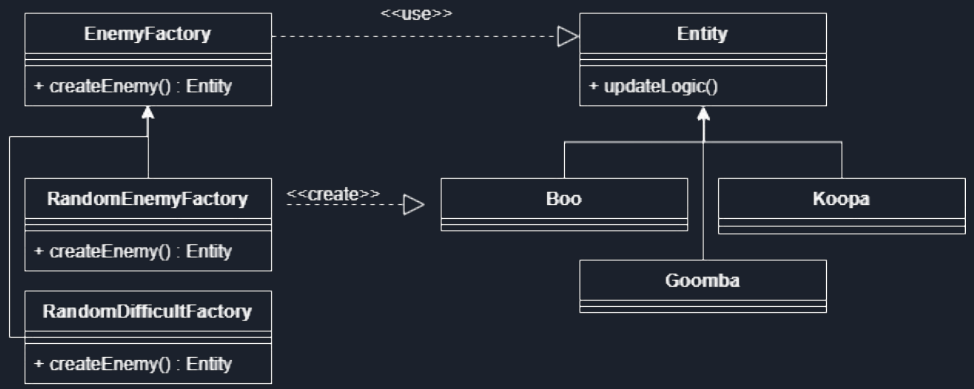
- We have managed to generalize the interface that will allow us to make many changes in our game without modifying the rest of the code:

```
class Game {
  private enemyFactory: EnemyFactory;

  constructor(enemyFactory: EnemyFactory) { //Dat DI
   this.enemyFactory = enemyFactory;
  }

  function gameLogic() {
    ...
    //More code above
    if (shouldSpawnEnemy()) {
      // this.enemyFactory is of type EnemyFactory
      let enemy = this.enemyFactory.createEnemy();
    }
    ...
    //More code below
  }
}
```

Classification of design patterns

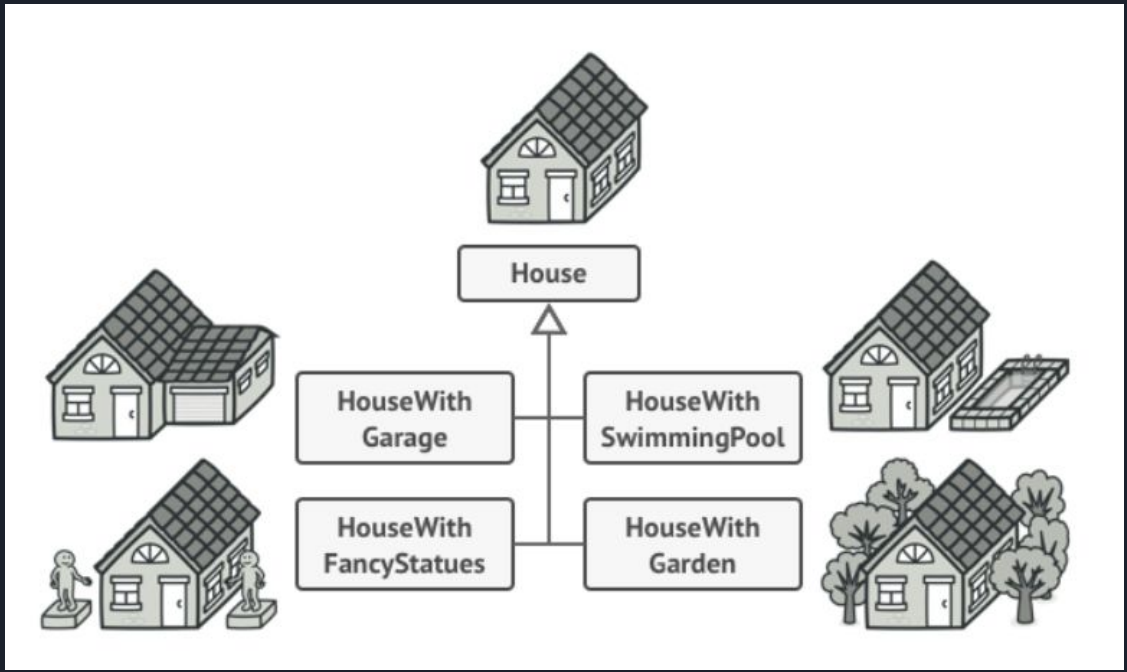
- Factory: define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses.





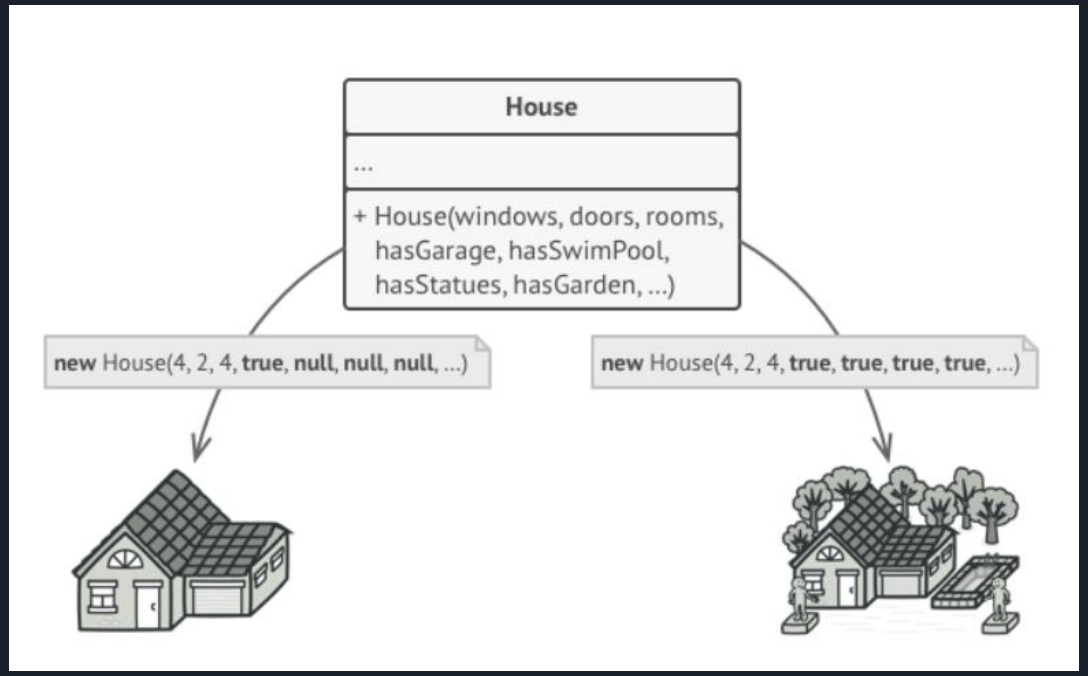
Classification of design patterns

- We want to build a house:



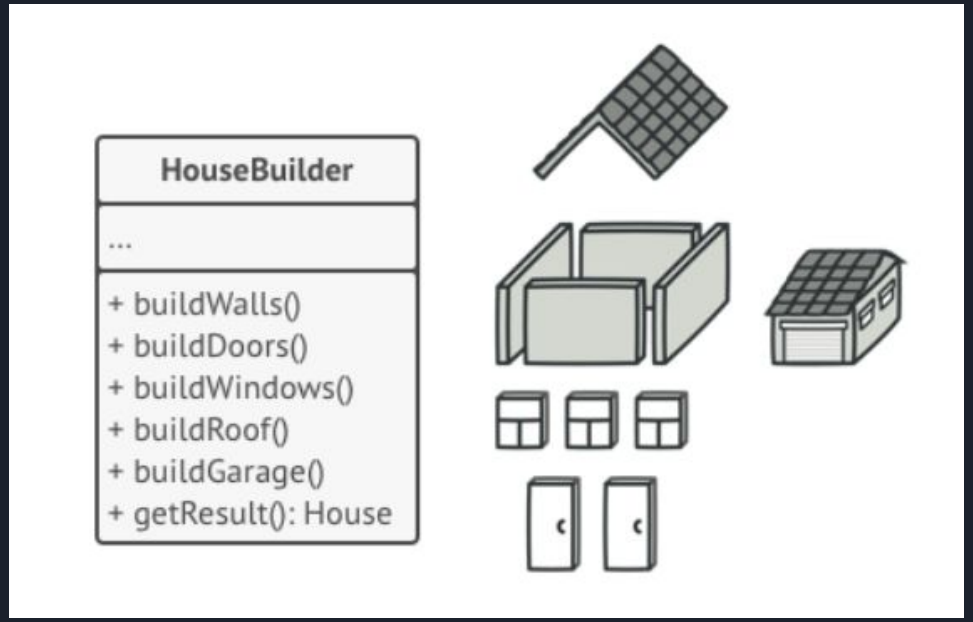
Classification of design patterns

- An option would be create a giant constructor directly in the basic House class with all possible parameters that control the house object:



Classification of design patterns

- The builder pattern extracts the object's construction code from its own class and moves it to separate objects called the constructors

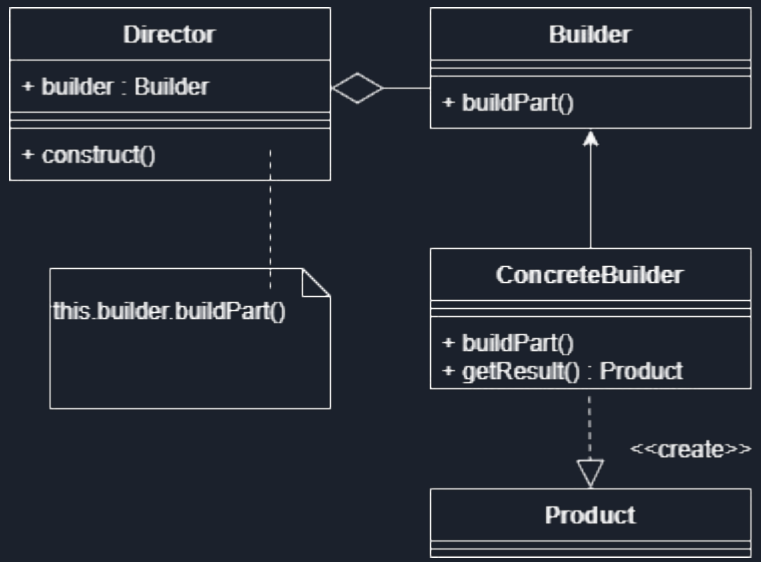


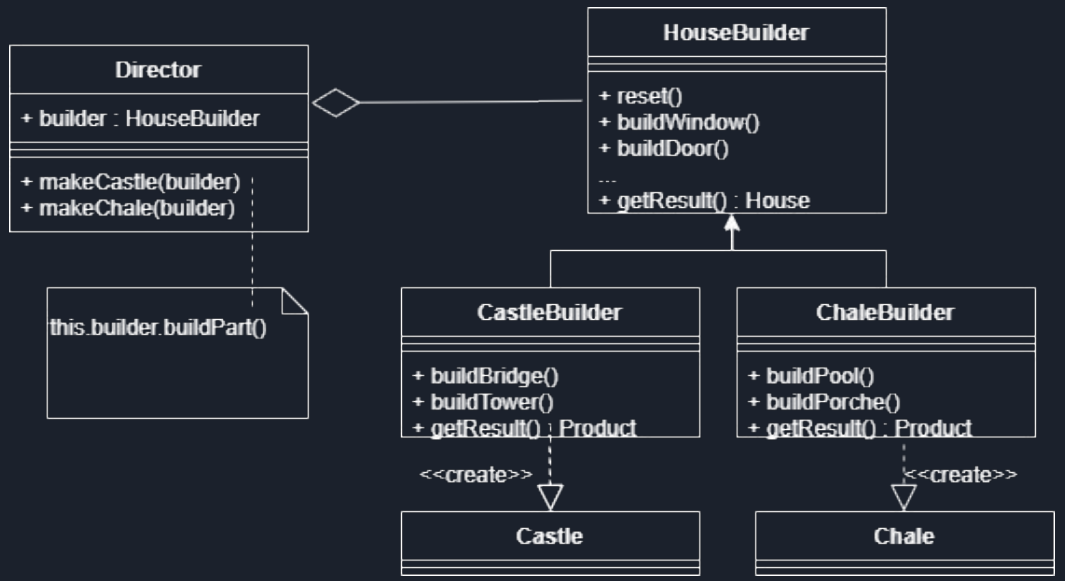
Classification of design patterns

- You can create several different builder classes that implement the same set of building steps, but in a different manner:

Classification of design patterns

- Builder: lets you construct complex objects step by step.





Classification of design patterns

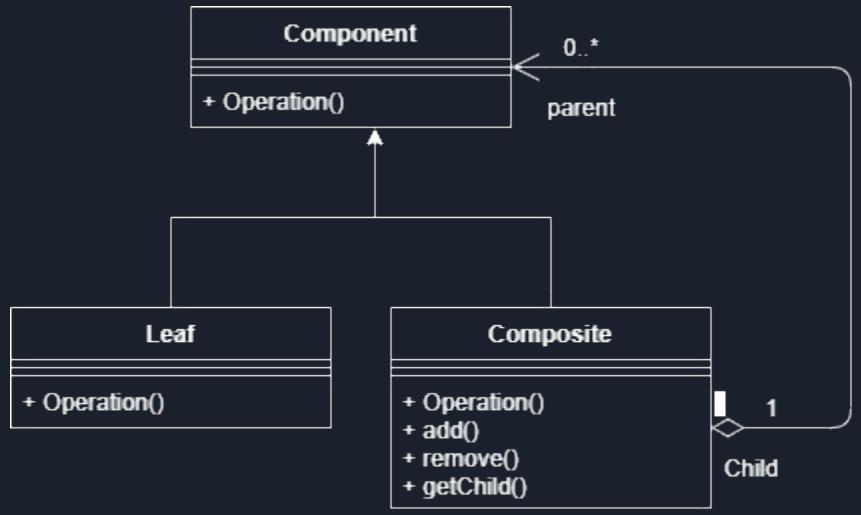
- ***Structural patterns***, explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.

Classification of design patterns

- We want that when the enemy dies, the rest of enemies die too:

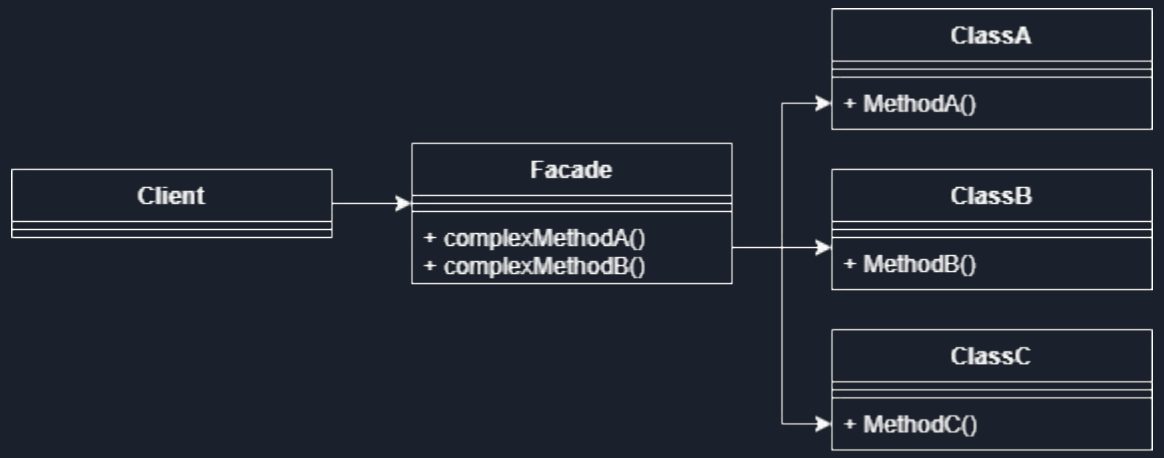
Classification of design patterns

- Composite: lets you compose objects into tree structures and then work with these structures as if they were individual objects.



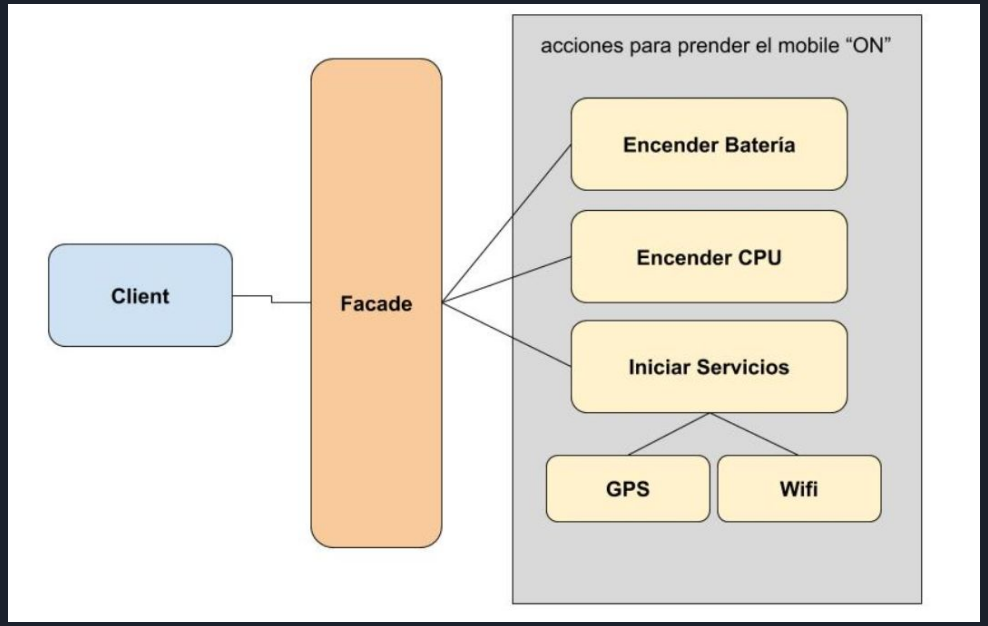
Classification of design patterns

- Facade: provides a simplified interface to a library, a framework, or any other complex set of classes.



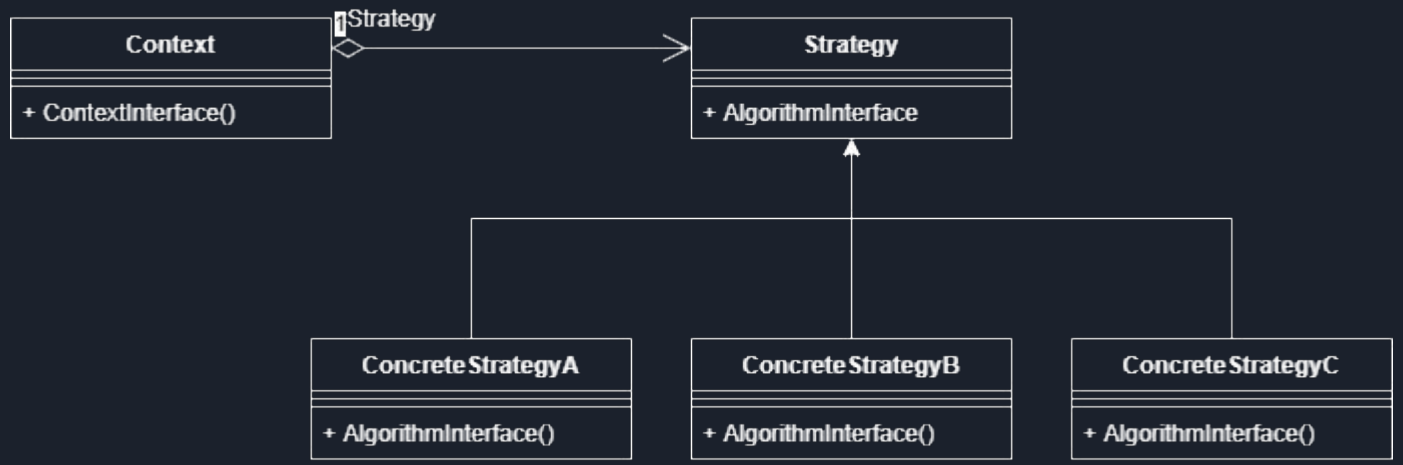
Classification of design patterns

- On a mobile phone that we must turn on and off. To do that it is necessary to carry out several actions:



Classification of design patterns

- ***Behavioral patterns***, take care of effective communication and the assignment of responsibilities between objects.



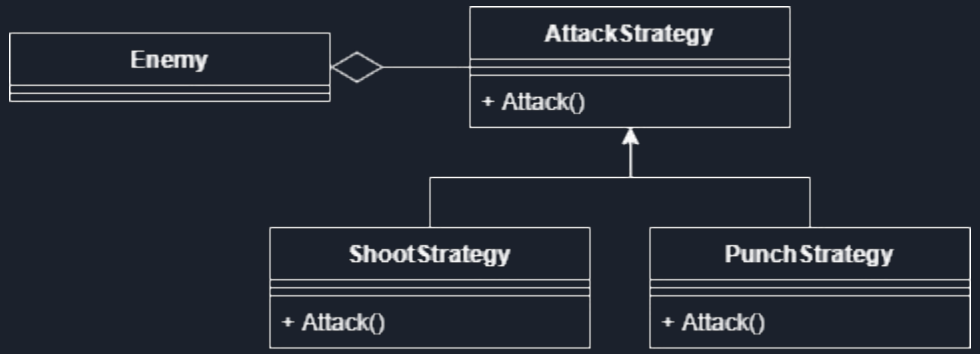
Classification of design patterns

- The attack depends on the type of enemy:

```
function computePokemonAttack(battle: BattleState, selectedAttack: Attack): BattleState {
  //Damage functions
  if (attack.type == 'PHYSICAL') {
   battle.foePokemon.makePhysicalDamage();
  }
  else if (attack.type == 'SPECIAL') {
   battle.foePokemon.makeSpecialDamage();
  }
  //Status change functions
  if (attack.diminishesPhysicalAttackStat) {
   battle.foePokemon.diminishPhysicalAttack();
  }
  if (attack.diminishesSpecialAttackStat) {
   battle.foePokemon.diminishSpecialAttack();
  }
  if (attack.diminishesSpeedStat) {
   battle.foePokemon.diminishSpeed();
  }
  if (attack.diminishesPhysicalDefenseStat) {
   battle.foePokemon.diminishPhysicalDefense();
  }
  if (attack.diminishesSpecialDefenseStat) {
   battle.foePokemon.diminishSpecialDefense();
  }

  //Battle settings functions
  if (attack.canMakeRain) {
    if (Math.random() > 0.5) {
      battle.startRain();
    }
  }
  if (attack.canMakeSandStorm) {
    if (Math.random() > 0.5) {
      battle.startSandStorm();
    }
  }
```

Classification of design patterns

- We want to create a new enemy with the same attack pattern than another enemy:



Making a design pattern

Making a design pattern

- Context
- Problem
- Solution

Documentation of the design pattern

- **Pattern Name and Classification**
- **Intent**
- **Also Known As**

Documentation of the design pattern

- **Motivation (Forces)**
- **Applicability**
- **Structure**

Documentation of the design pattern

- **Participants**
- **Collaboration**
- **Consequences**

Documentation of the design pattern

- **Implementation**
- **Sample Code**
- **Known Uses**

Documentation of the design pattern

- **Related Patterns**

Observations

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

Design patterns have to be properly been used.

Antipatterns.

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=CVIpjFJN17U

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/