Specific Design Patterns

Thomas Edward Bradley
Daniel Méndez Rodríguez

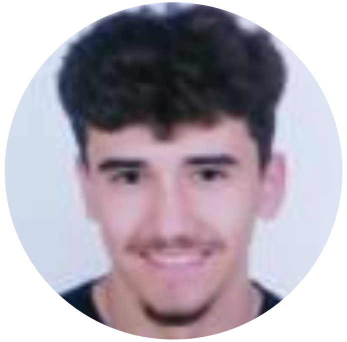
20.03.2023

CONTENTS

OUR TEAM MEMBERS

Thomas Edward Bradley

thomas.edward.bradley.20@ull.edu.es

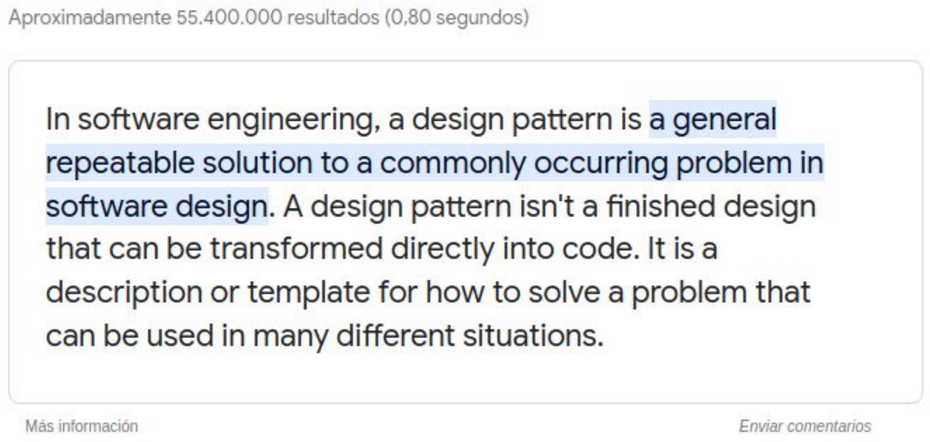


Daniel Méndez Rodríguez

daniel.mendez.33@ull.edu.es

Brief Refresher

What are design patterns?



Aproximadamente 55.400.000 resultados (0,80 segundos)

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Más información

Enviar comentarios

Pattern Types

Creational

Object creation mechanisms

Structural

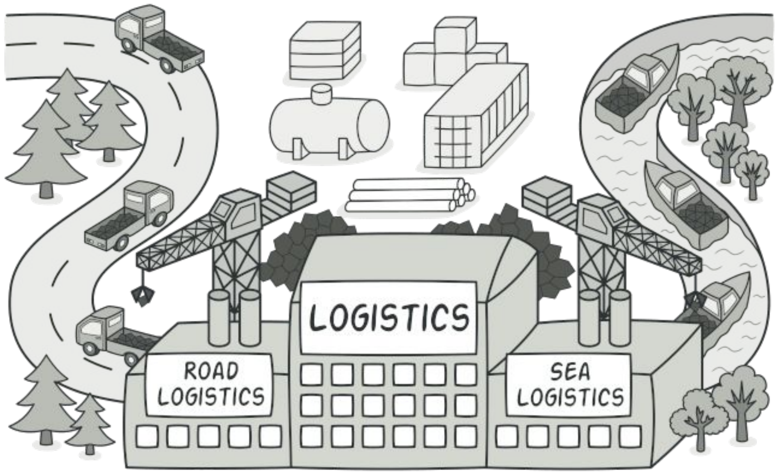
Assemble objects and classes into larger, flexible and efficient structures

Behavioral

Algorithms and the assignment of responsibilities between objects

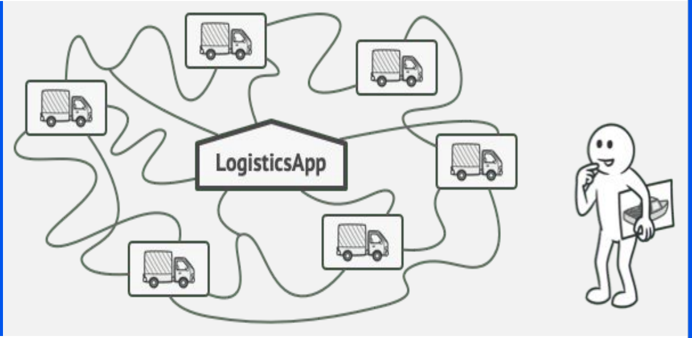
Factory Method

🏆 🏆 🏆



Example

Logistics Management Application

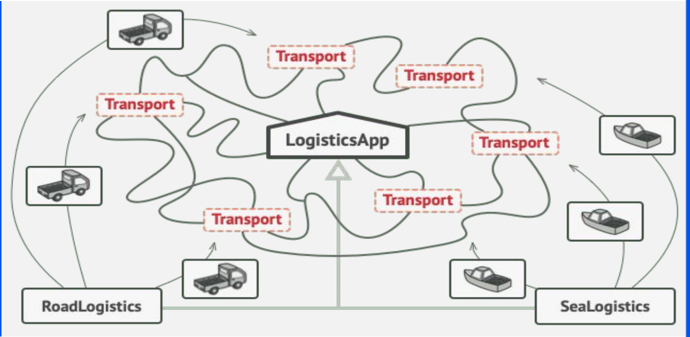


Problem

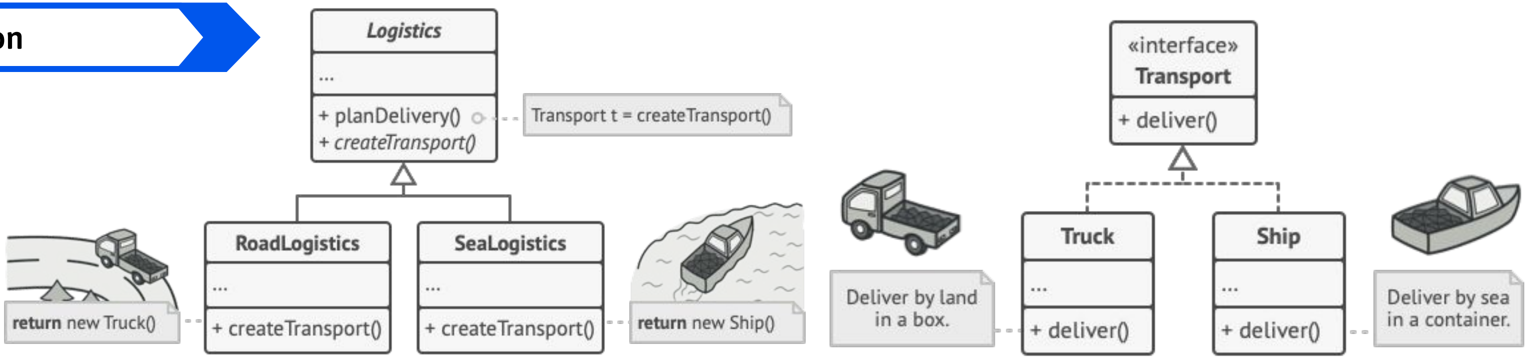
- You create a logistics management application for trucks (with most of the code being for the Truck class)
- You get asked to add ships (which would require changing the entire codebase)
- Adding any more vehicles would also require this change

Example

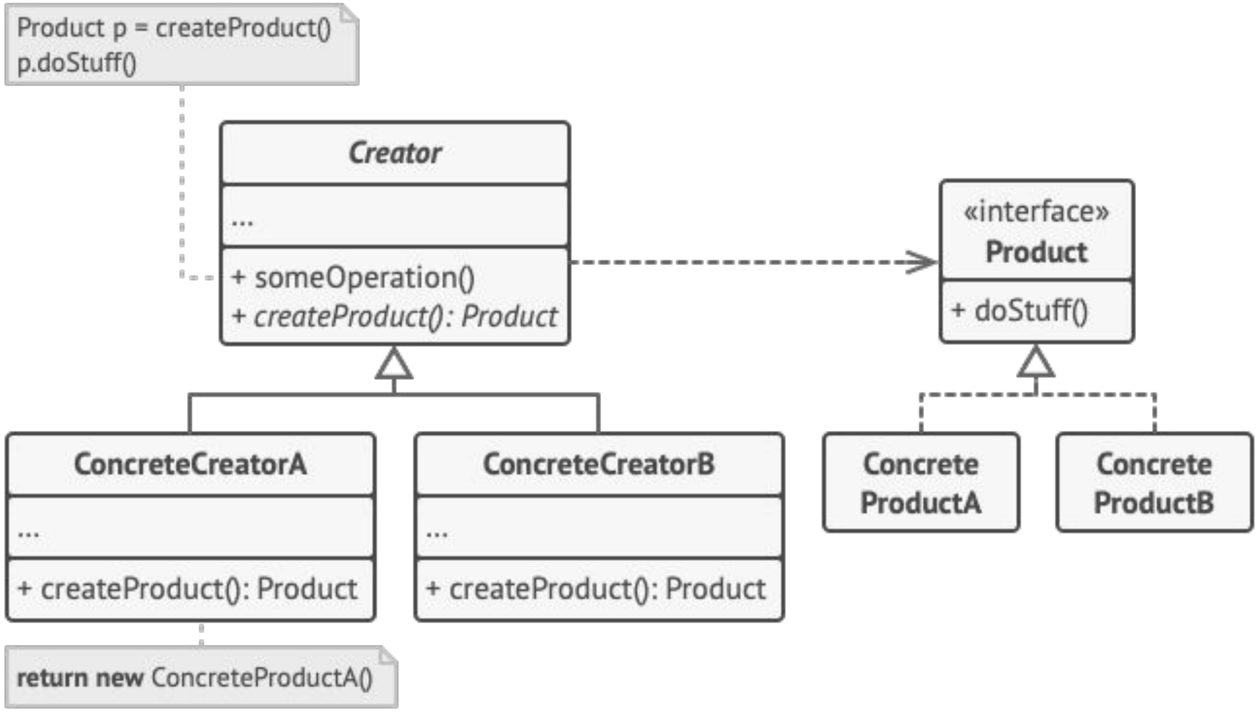
Logistics Management Application



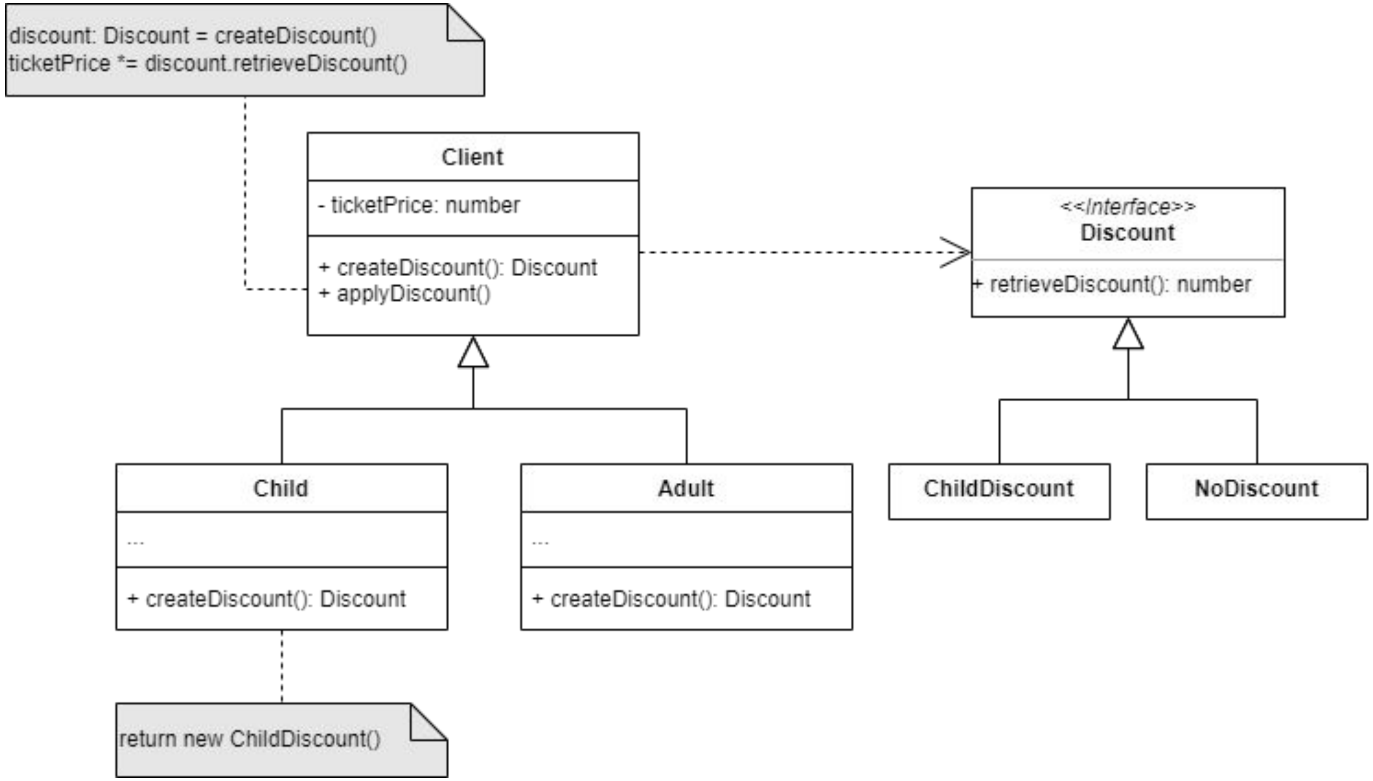
Solution



Theory


Product p = createProduct()
p.doStuff()
Creator
...
+ someOperation()
+ createProduct(): Product
«interface»
Product
+ doStuff()
ConcreteCreatorA
...
+ createProduct(): Product
ConcreteCreatorB
...
+ createProduct(): Product
Concrete ProductA
Concrete ProductB
return new ConcreteProductA()

Case Study


discount: Discount = createDiscount()
ticketPrice *= discount.retrieveDiscount()
Client
- ticketPrice: number
+ createDiscount(): Discount
+ applyDiscount()
<<Interface>>
Discount
+ retrieveDiscount(): number
Child
...
+ createDiscount(): Discount
Adult
...
+ createDiscount(): Discount
ChildDiscount
NoDiscount
return new ChildDiscount()

Applicability

You don't know the types and dependencies your code should work with

To provide users a way to extend your library or framework

To save system resources by reusing existing objects

Advantages & Disadvantages

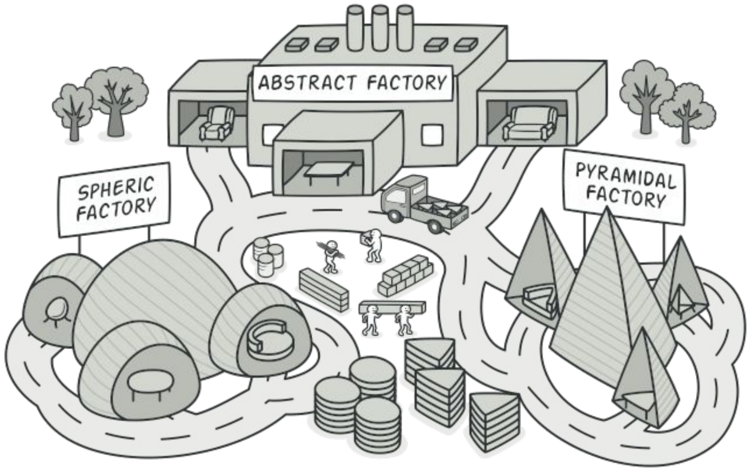
Advantages

- Avoid tight coupling between the creator and the concrete products.
- Single Responsibility Principle.
- Open/Closed Principle.

Disadvantages

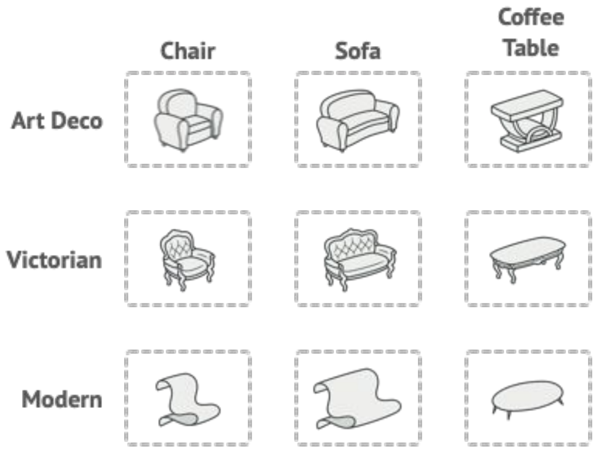
- The code may become more complicated since you need to introduce a lot of new subclasses

Abstract Factory



Example

Furniture Shop Simulator



Problem

Currently you are running a furniture shop:

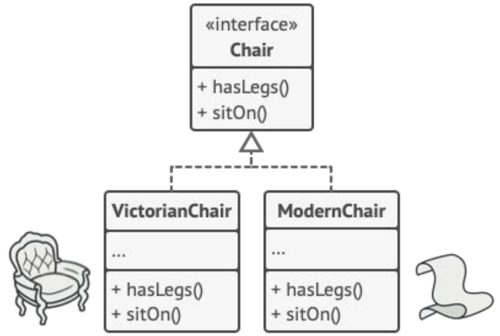
- You have a family of related furniture and several styles of this family.
- Customers get angry when they receive non-matching furniture
- As your furniture catalog updates quite frequently, you don’t want to have to update the code every time

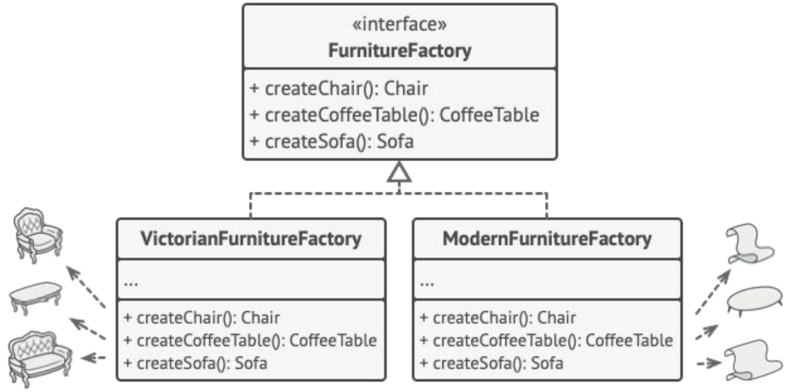
Example

Furniture Shop Simulator

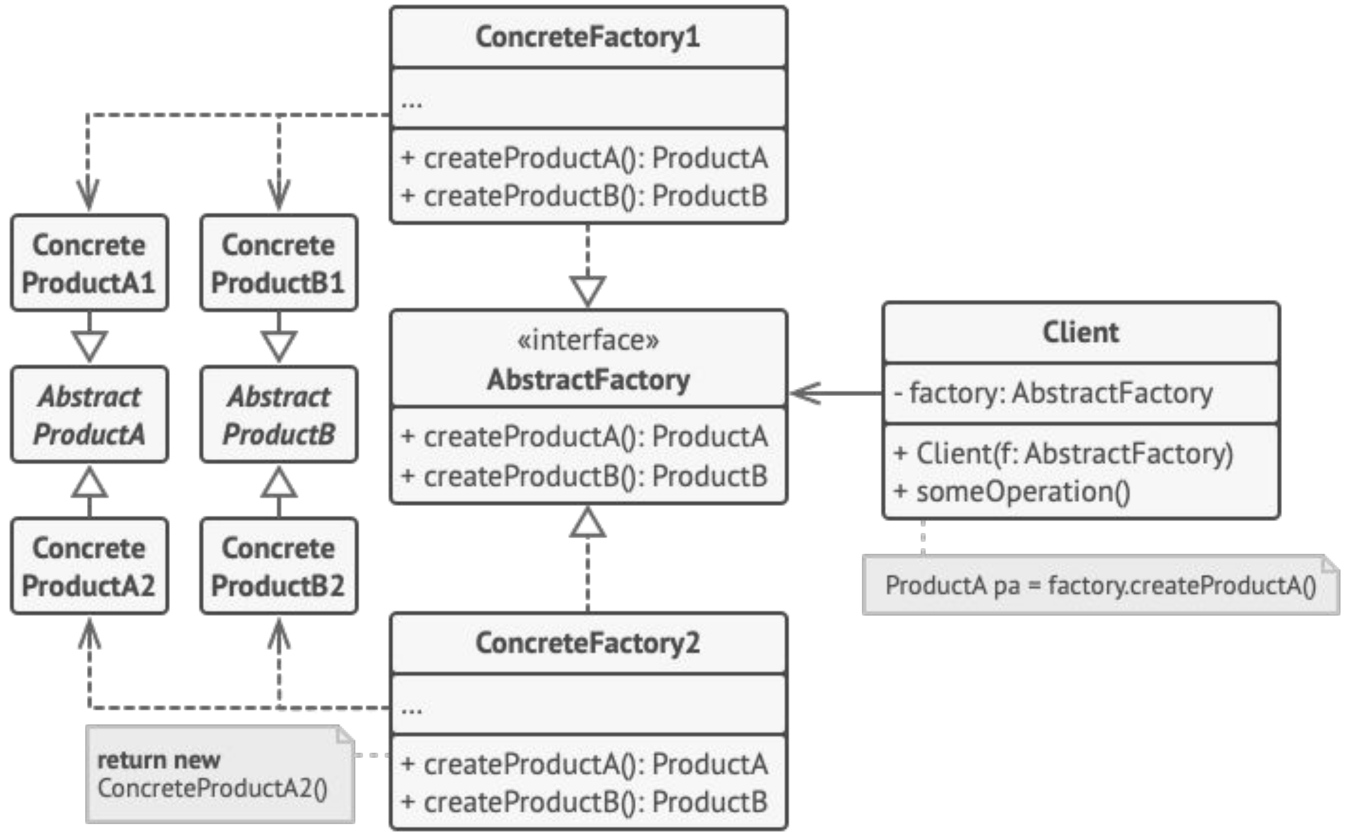
Solution

- Have an interface for each furniture piece of the family
- Have an Abstract factory that produces each abstract piece of furniture
- The client will have the furniture requested regardless of the concrete factory called, and the rest of the furniture family will combine with the requested piece style

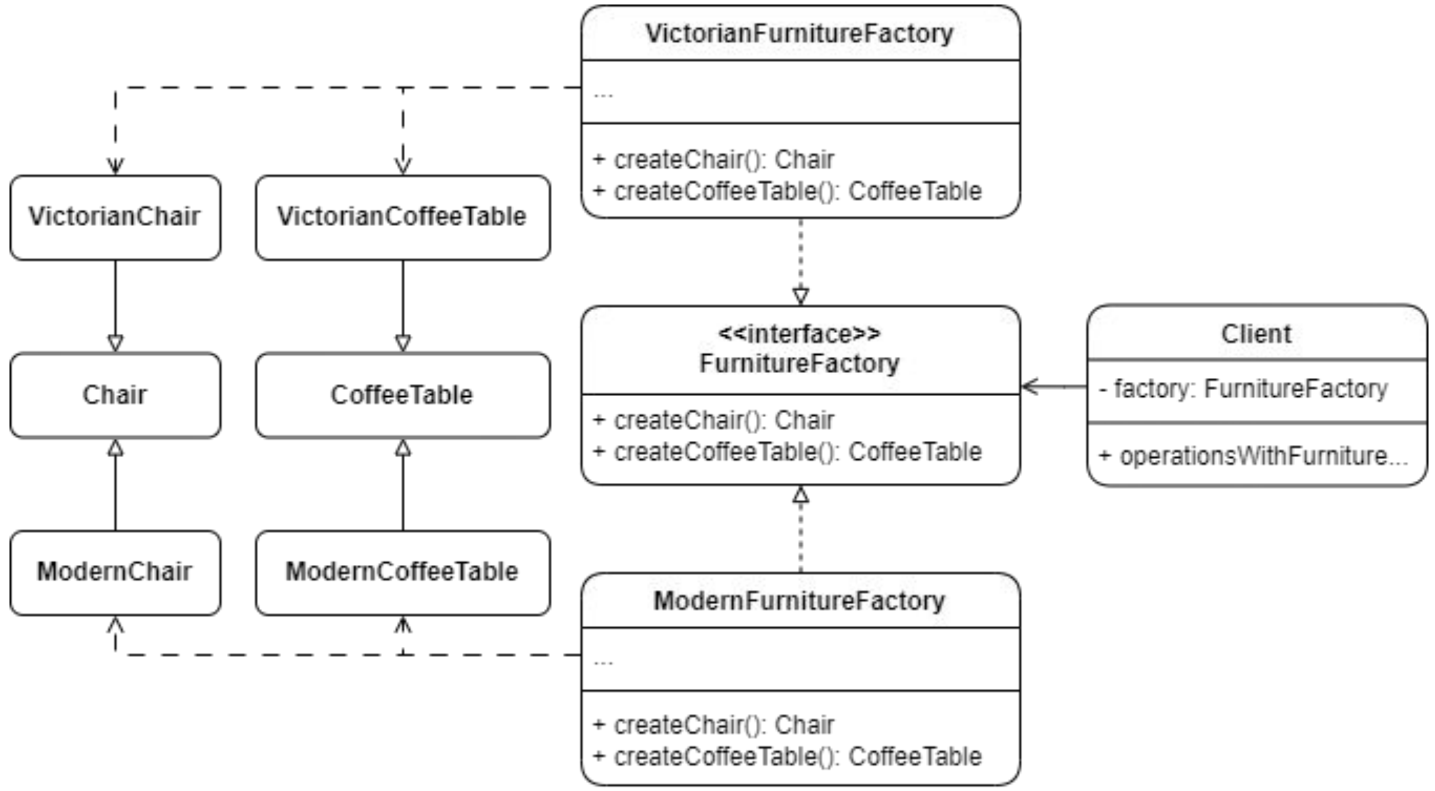




Theory


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
Abstract ProductA
Abstract ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

Case Study



VictorianFurnitureFactory
...
+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
VictorianChair
VictorianCoffeeTable
Chair
CoffeeTable
<<interface>>
FurnitureFactory
+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable
Client
- factory: FurnitureFactory
+ operationsWithFurniture...
ModernChair
ModernCoffeeTable
ModernFurnitureFactory
...
+ createChair(): Chair
+ createCoffeeTable(): CoffeeTable

Applicability

When your code needs to work with various families of related products, but you don’t want it to depend on the concrete classes of those products.

When you have a class with a set of Factory Methods that blur its primary responsibility.

Advantages & Disadvantages

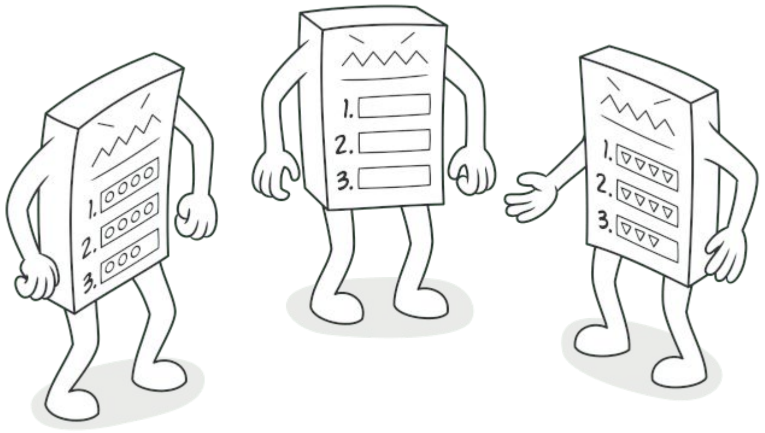
Advantages

- You make sure that the products from a factory are compatible with each other.
- The client does not need to know the exact implementation of the concrete products/factories by implementing an abstract interface.
- Single Responsibility Principle.
- Open/Closed Principle.

Disadvantages

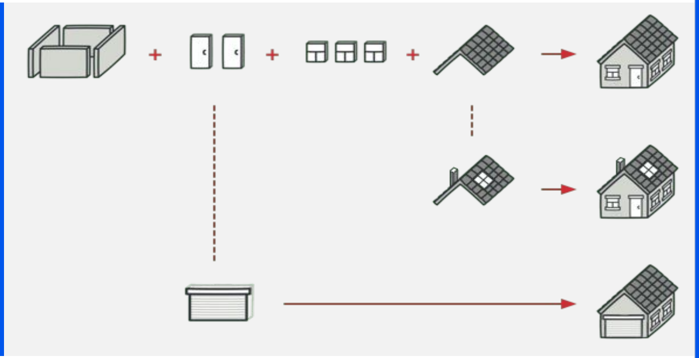
- The code may become more complicated than it should be.

Template Method



Example

Mass Housing Construction



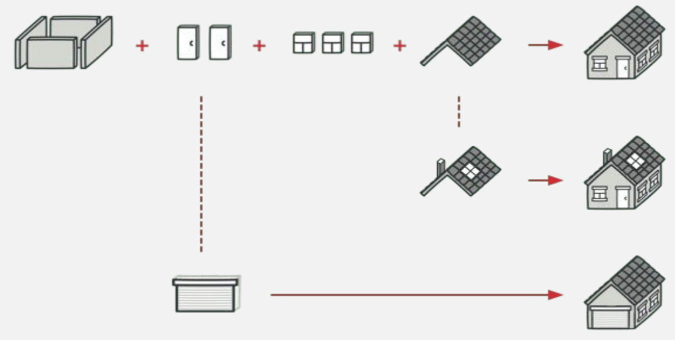
Problem

You’ve just started a company that builds houses

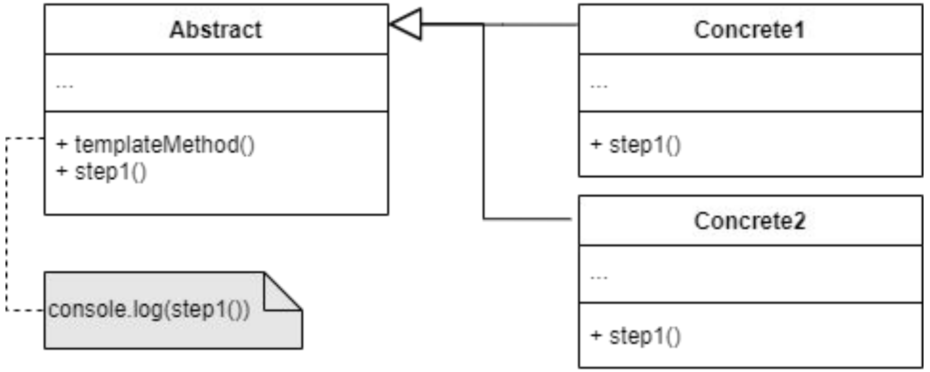
- You need to be able to build a house that uses any different combinations of parts
- Some of these parts can be optional and have no need to be implemented

Example

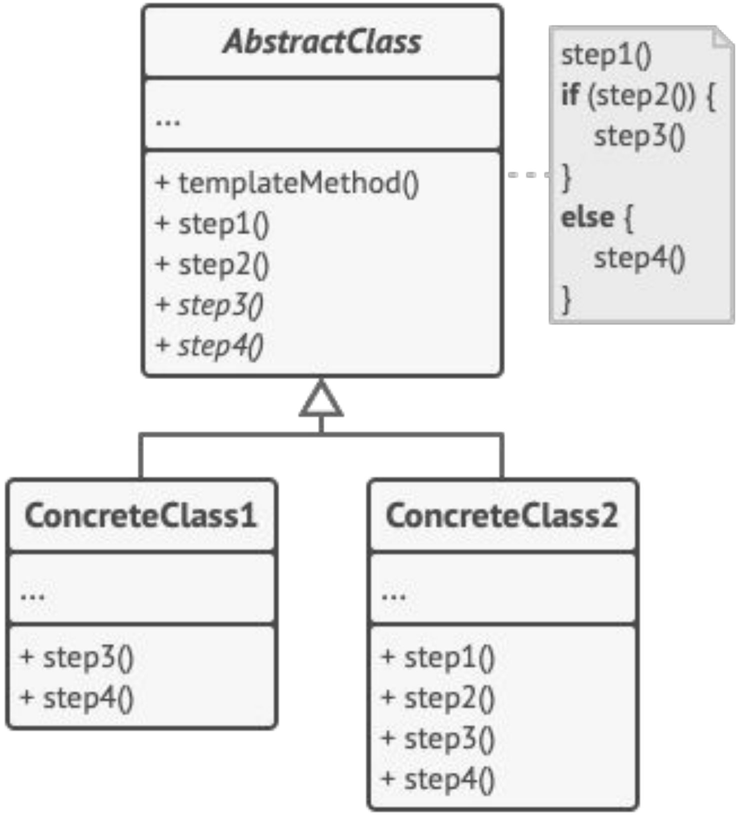
Mass Housing Construction



Solution

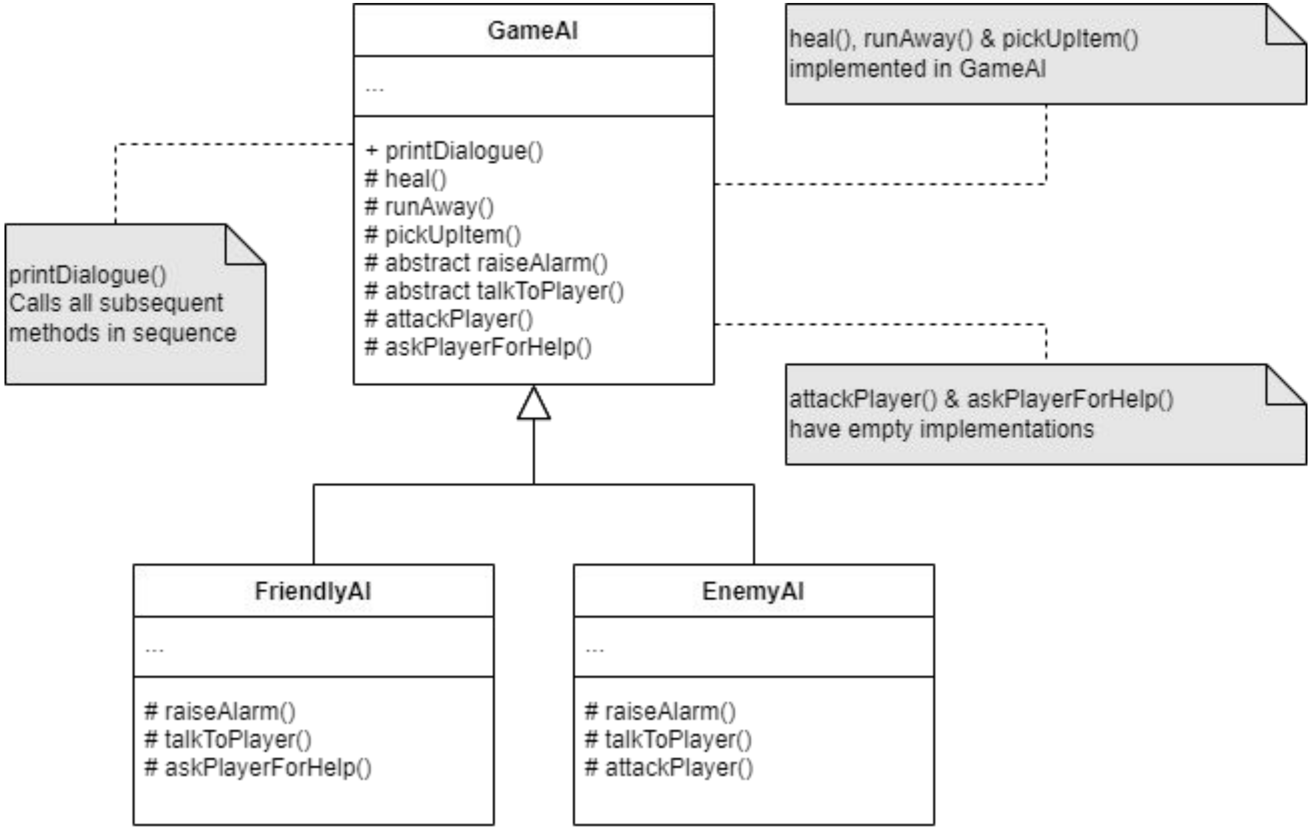


Theory



```
step1()
if (step2()) {
    step3()
}
else {
    step4()
}
```

Case Study


GameAI
...
+ printDialogue()
# heal()
# runAway()
# pickUpItem()
# abstract raiseAlarm()
# abstract talkToPlayer()
# attackPlayer()
# askPlayerForHelp()
heal(), runAway() & pickUpItem() implemented in GameAI
printDialogue() Calls all subsequent methods in sequence
attackPlayer() & askPlayerForHelp() have empty implementations
FriendlyAI
...
# raiseAlarm()
# talkToPlayer()
# askPlayerForHelp()
EnemyAI
...
# raiseAlarm()
# talkToPlayer()
# attackPlayer()

Applicability

To let clients extend only particular steps of an algorithm, but not the whole.

Several classes contain almost identical algorithms (forcing you to modify all of these when the algorithm changes)

Advantages & Disadvantages

Advantages

- Clients can only override certain parts of an algorithm, making them less affected by changes that happen to other parts of the algorithm.
- You can move the duplicate code into a superclass.

Disadvantages

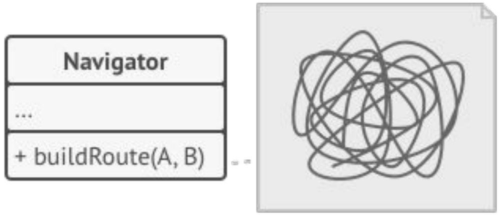
- Clients may be limited by the template provided.
- You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass.
- Template methods tend to be harder to maintain the more steps they have.

Strategy

🏆 🏆 🏆

Example

Furniture Shop Simulator



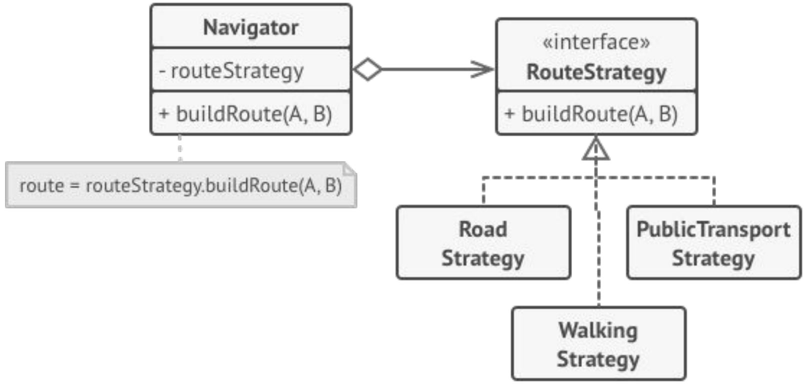
Problem

You are the owner of a navigation app for tourists:

- At the beginning it was very simple and only supported a map for orientation.
- As users had requested, you implemented a feature for automatic route planning, which only supported one type of navigation in the first version.
- Your app became popular and started implementing more and more types of navigation to the route planner, making the navigator very hard to maintain.
- Team work started to become inefficient as most of the time went to fixing integration conflicts.

Example

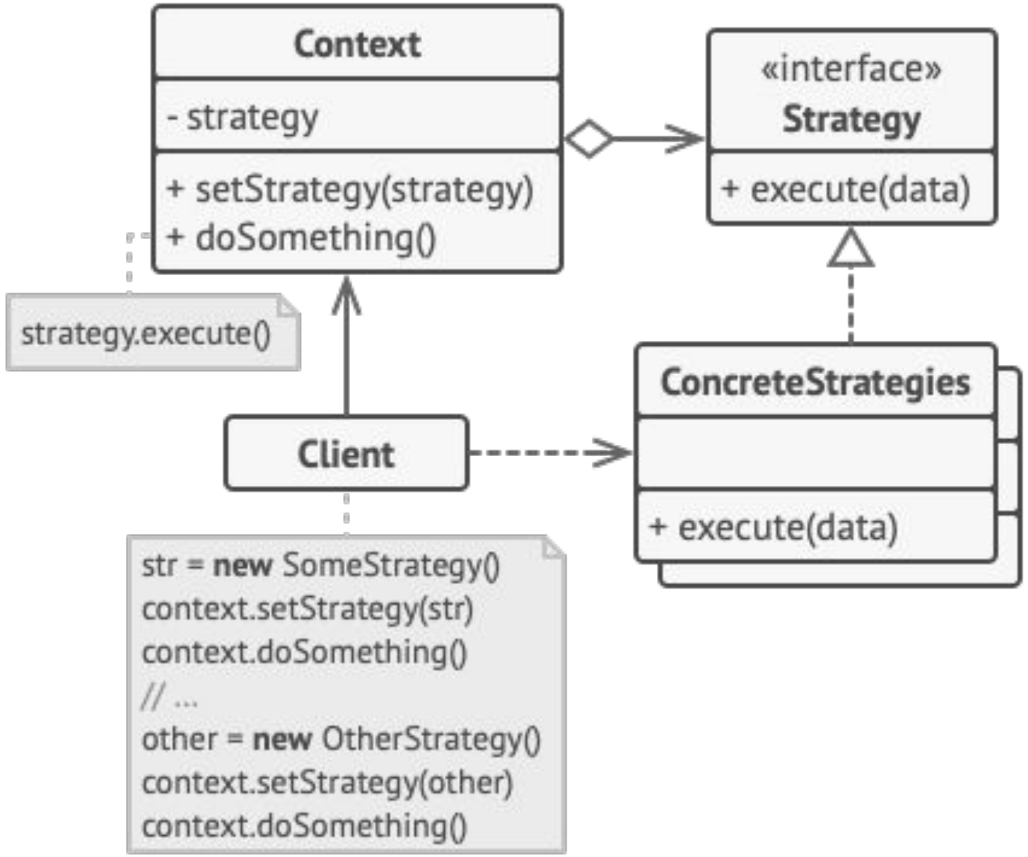
Furniture Shop Simulator



Solution

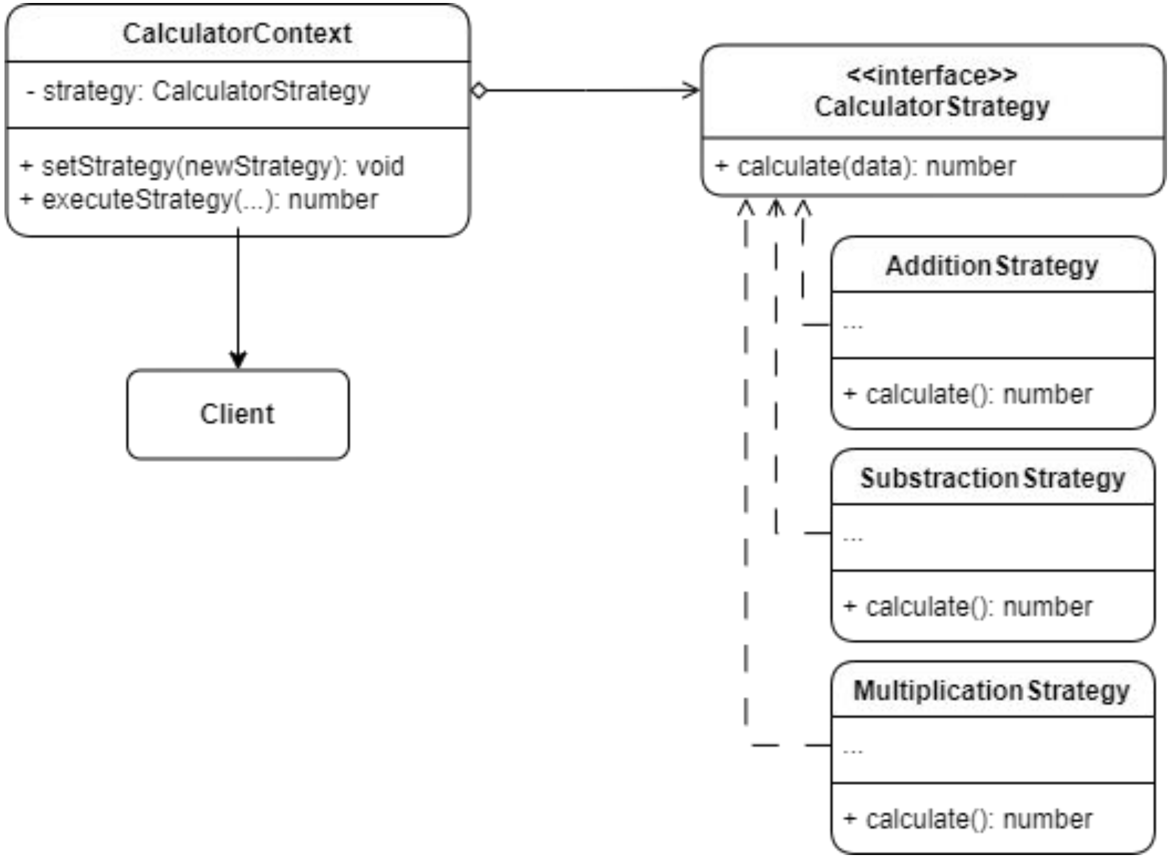
- Divide the several ways the original class generated the route into *strategies*.
- The original **Navigator** class is now the context that triggers the algorithm of a strategy selected by the user, but it doesn’t know its implementation.
- The users still can get routes generated by different strategies in the same execution as the context class allows to change the strategy selected on runtime.
- The code is much more clean, scalable and the difficulty of maintaining it decreased in a significant way.

Theory



```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

Case Study


CalculatorContext
- strategy: CalculatorStrategy
+ setStrategy(newStrategy): void
+ executeStrategy(...): number
Client
<<interface>>
CalculatorStrategy
+ calculate(data): number
Addition Strategy
...
+ calculate(): number
Substraction Strategy
...
+ calculate(): number
Multiplication Strategy
...
+ calculate(): number

Applicability

When you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.

When you have a lot of similar classes that only differ in the way they execute some behavior.

To isolate algorithms from the rest of the code.
The clients will interact with them through an interface.

When your class has a massive conditional statement that switches between different variants of the same algorithm.

Advantages & Disadvantages

Advantages

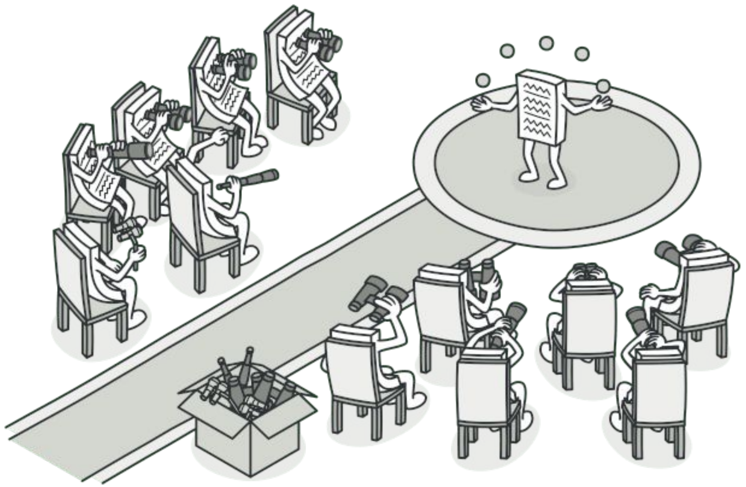
- You can swap algorithms used inside an object at runtime.
- You can isolate the implementation details of an algorithm from the code that uses it.
- You can replace inheritance with composition.
- Open/Closed Principle.

Disadvantages

- Overcomplicates the program if you only have a couple of rarely changing algorithms.
- Clients must be aware of the differences between strategies to be able to select a proper one.
- Can be replaced by implementing different versions of an algorithm inside a set of functions, which doesn’t bloat the code.

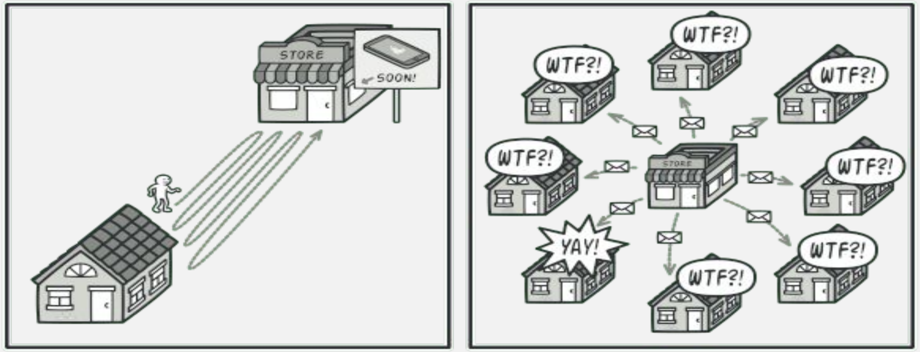
Observer

🏆 🏆 🏆



Example

Store & Spam



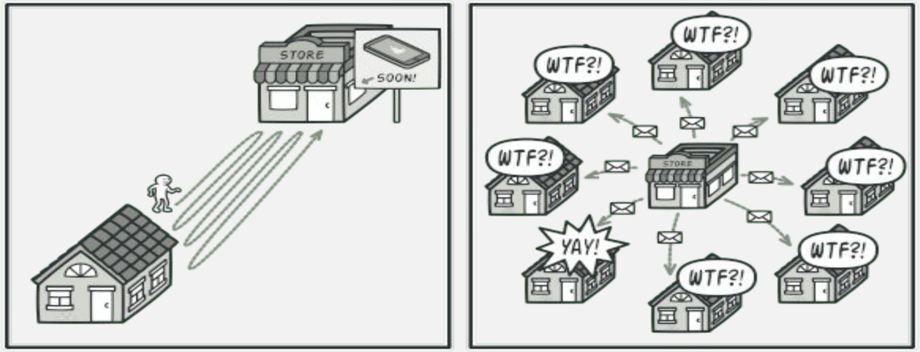
Problem

A customer is interested in buying a product from the store, which leaves us with two possible options:

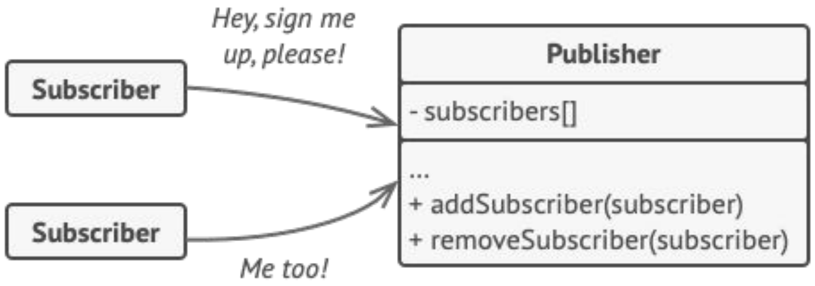
- They can either go there and check every day (waste a lot of time)
- The store can send an update to all it’s customers every day (unnecessary spam)

Example

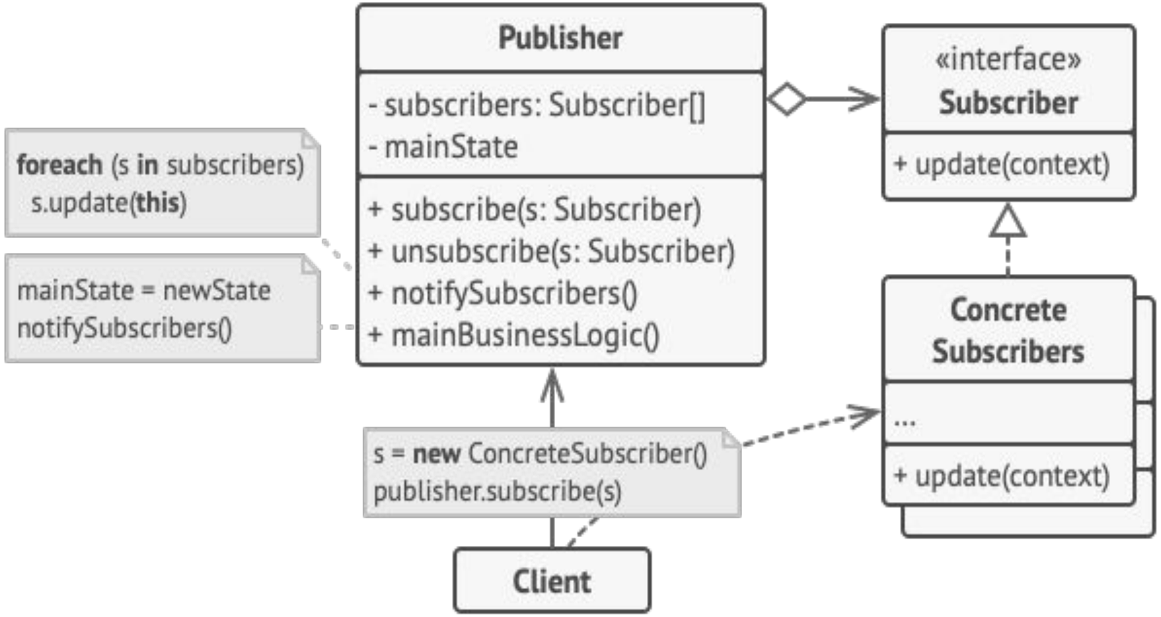
Store & Spam



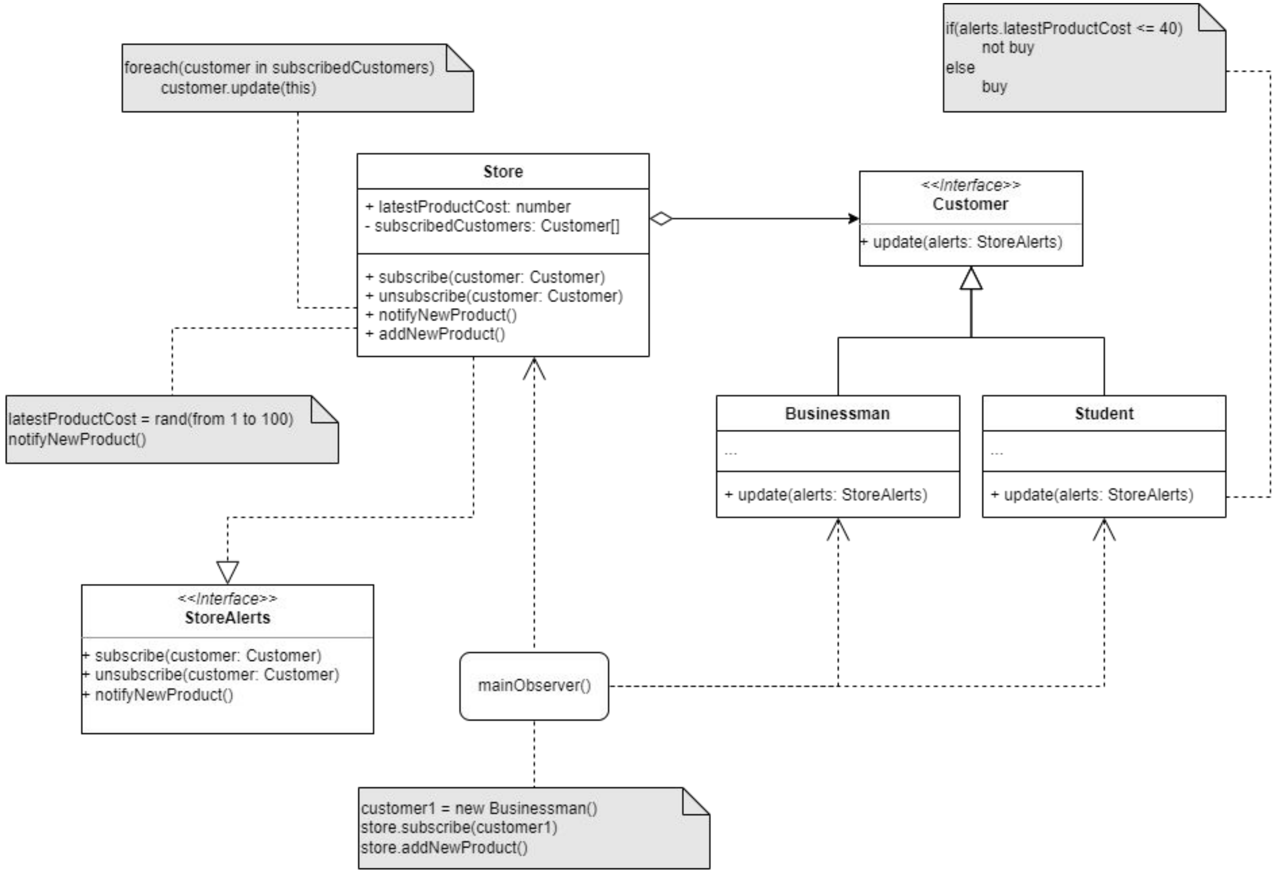
Solution



Theory


Publisher
- subscribers: Subscriber[]
- mainState
+ subscribe(s: Subscriber)
+ unsubscribe(s: Subscriber)
+ notifySubscribers()
+ mainBusinessLogic()
«interface»
Subscriber
+ update(context)
Concrete Subscribers
...
+ update(context)
foreach (s in subscribers)
s.update(this)
mainState = newState
notifySubscribers()
s = new ConcreteSubscriber()
publisher.subscribe(s)
Client

Case Study



foreach(customer in subscribedCustomers)
customer.update(this)
if(alerts.latestProductCost <= 40)
not buy
else
buy
Store
+ latestProductCost: number
- subscribedCustomers: Customer[]
+ subscribe(customer: Customer)
+ unsubscribe(customer: Customer)
+ notifyNewProduct()
+ addNewProduct()
<<Interface>>
Customer
+ update(alerts: StoreAlerts)
Businessman
...
+ update(alerts: StoreAlerts)
Student
...
+ update(alerts: StoreAlerts)
latestProductCost = rand(from 1 to 100)
notifyNewProduct()
<<Interface>>
StoreAlerts
+ subscribe(customer: Customer)
+ unsubscribe(customer: Customer)
+ notifyNewProduct()
mainObserver()
customer1 = new Businessman()
store.subscribe(customer1)
store.addNewProduct()

Applicability

Changes to the state of one object may require changing another, and the set of objects is unknown beforehand or changes dynamically.

Some objects in your app must observe others, but only for a limited time or in specific cases.

Advantages & Disadvantages

Advantages

- Open/Closed Principle.
- You can establish relations between objects at runtime.

Disadvantages

- Subscribers are notified in random order.

Bibliography

- Refactoring Guru: Design Patterns in Typescript
- Design Patterns: Elements of Reusable Object-Oriented Software, GoF
- Design Patterns Wikipedia page
- Example of use of Abstract Factory pattern
- PAI Design Patterns 22-23

Questions