Specific Design Patterns

Thomas Edward Bradley
Daniel Méndez Rodríguez

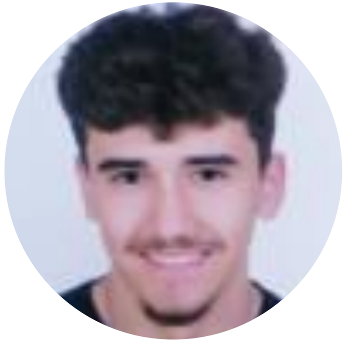
20.03.2023

CONTENTS

OUR TEAM MEMBERS

Thomas Edward Bradley

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

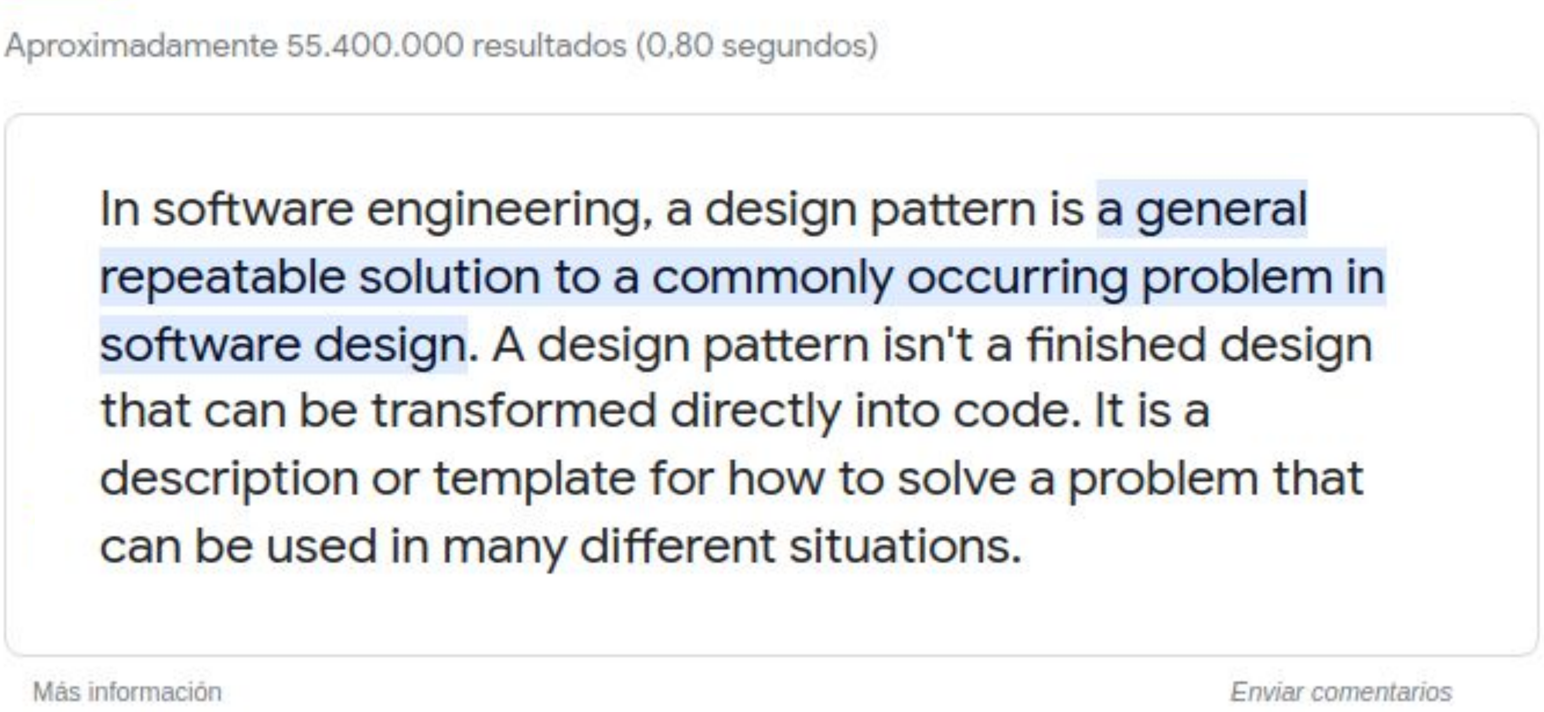


Daniel Méndez Rodríguez

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.

Pattern types

Creational

Object creation mechanisms.

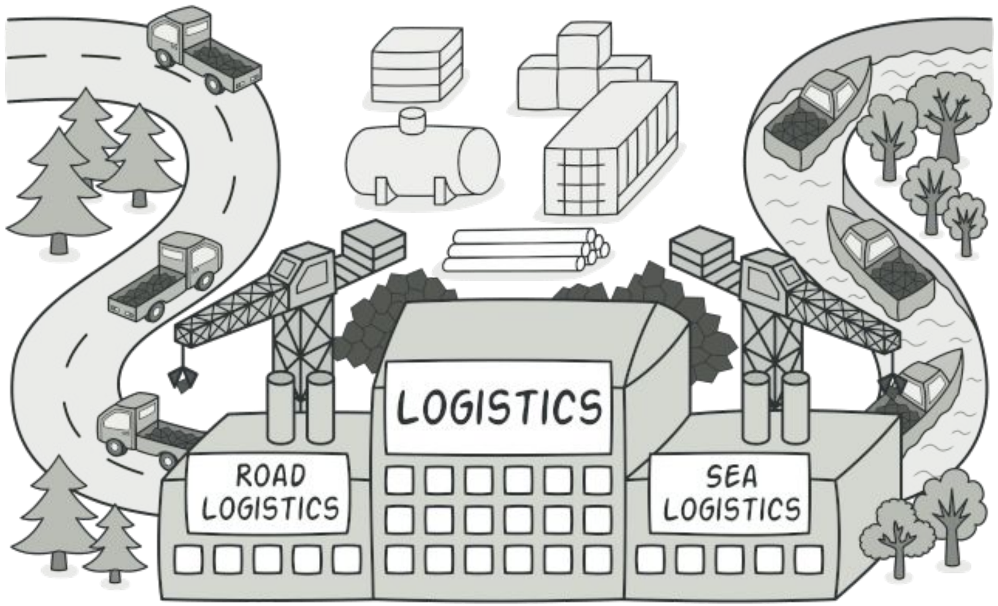
Structural

Assemble objects and classes into larger, flexible and efficient structures.

Behavioral

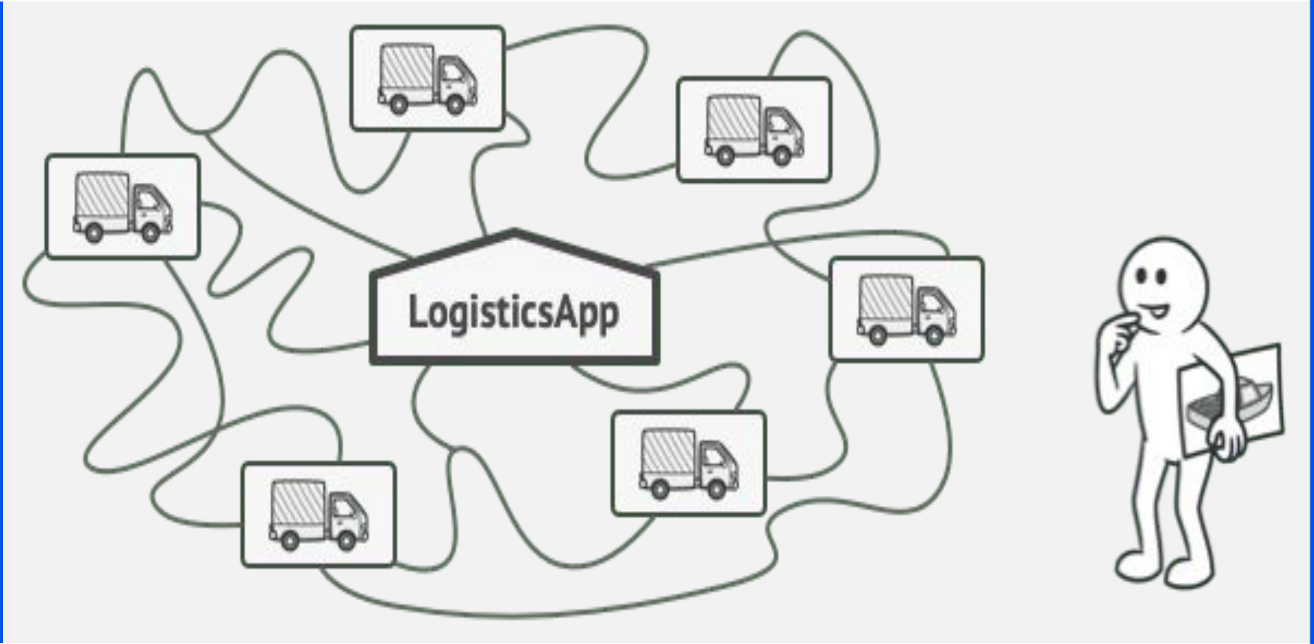
Algorithms and the assignment of responsibilities between objects.

Factory Method



Example

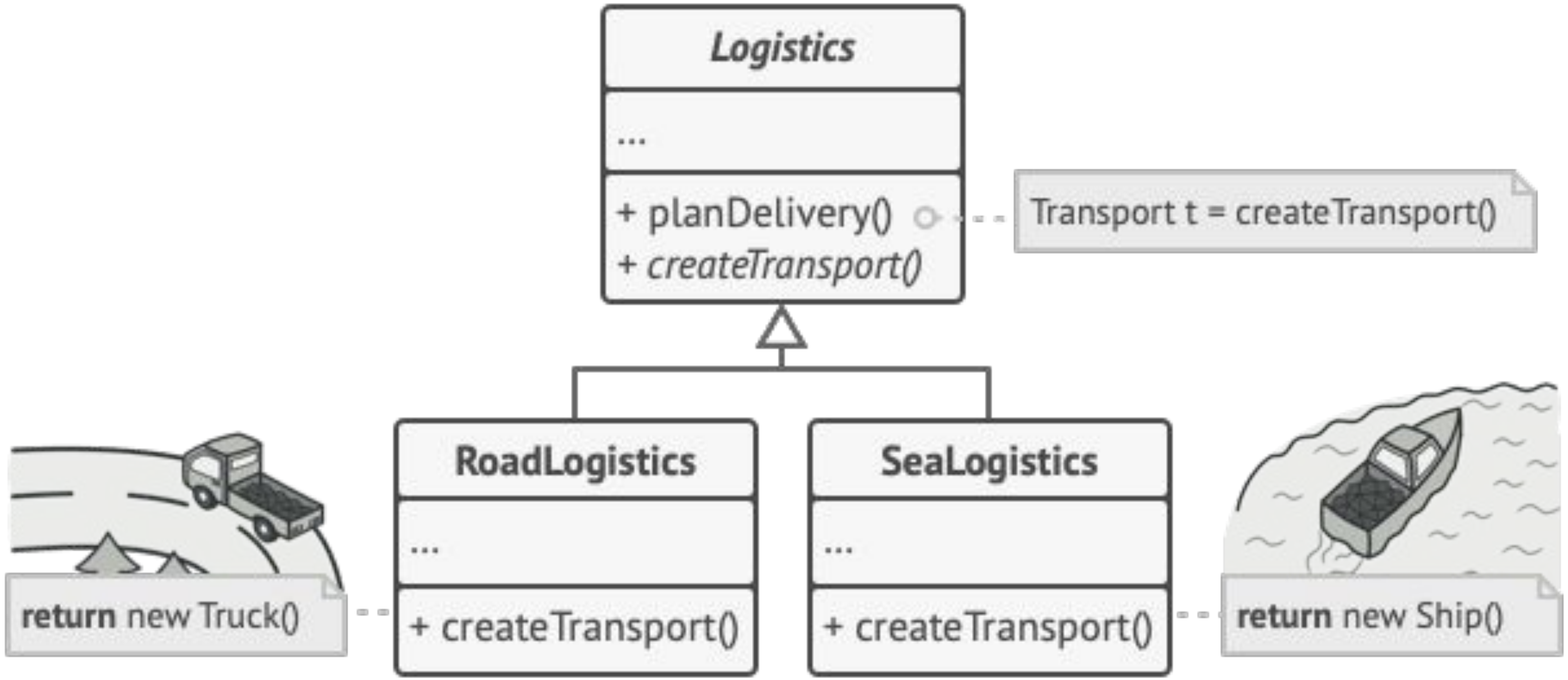
Logistics Management Application



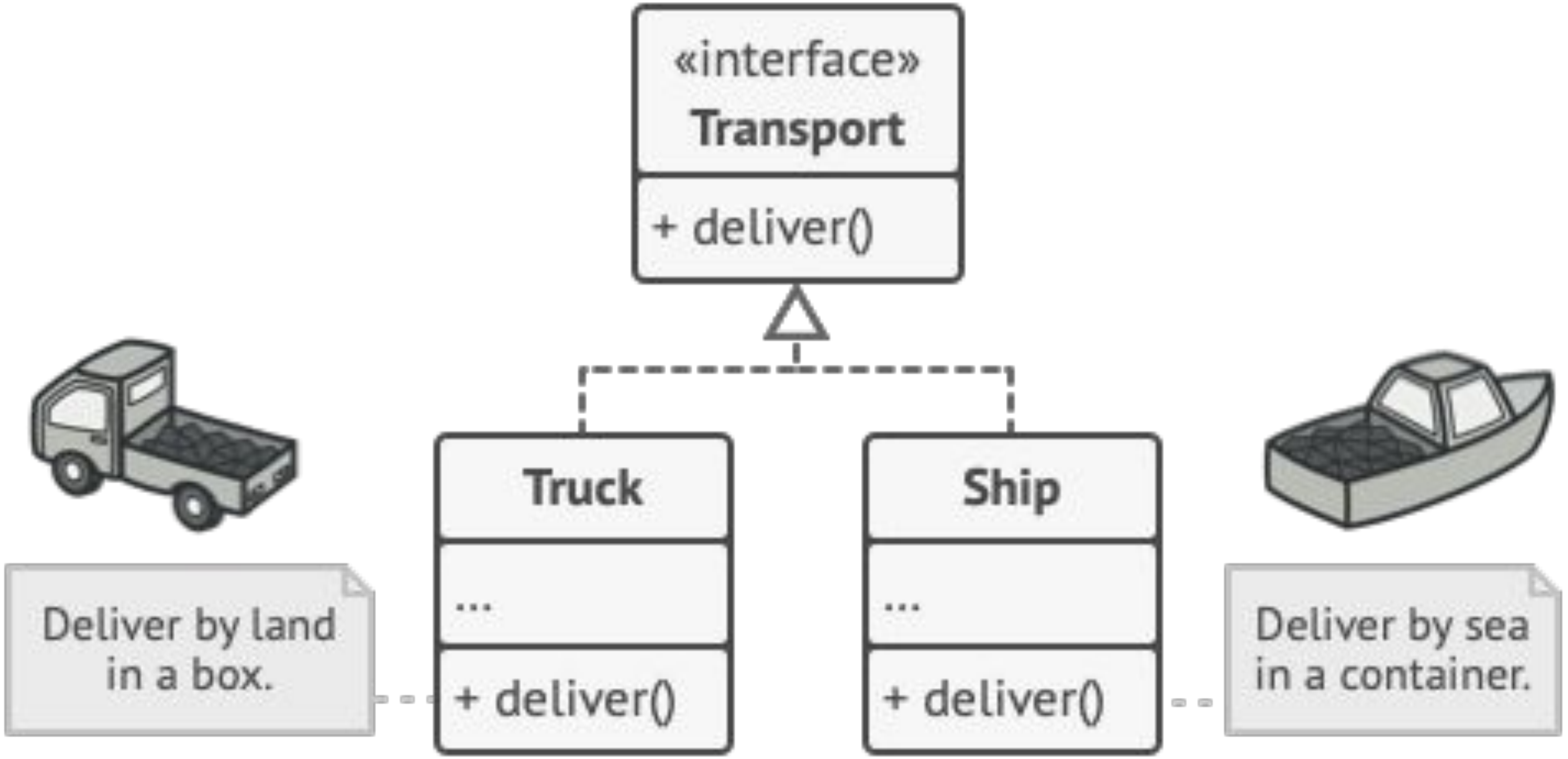
Problem

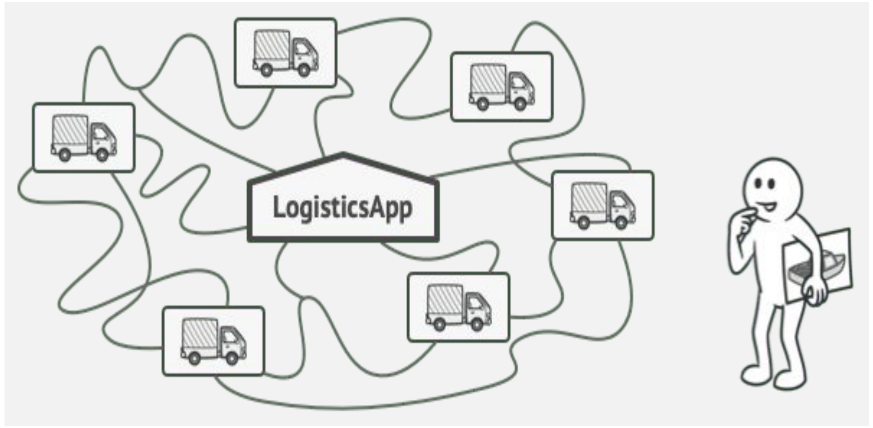
- You create a logistics management application for trucks (with most of the code being for the Truck class).
- You get asked to add ships (which would require changing the entire codebase).
- Adding any more vehicles would also require this change.

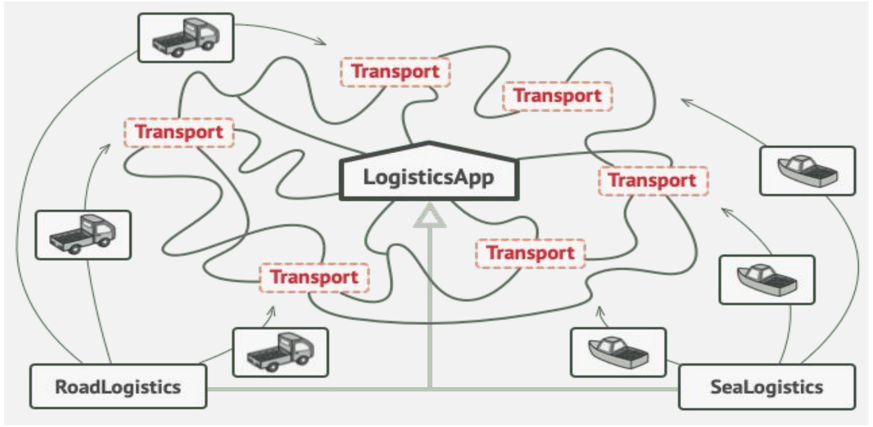
Solution



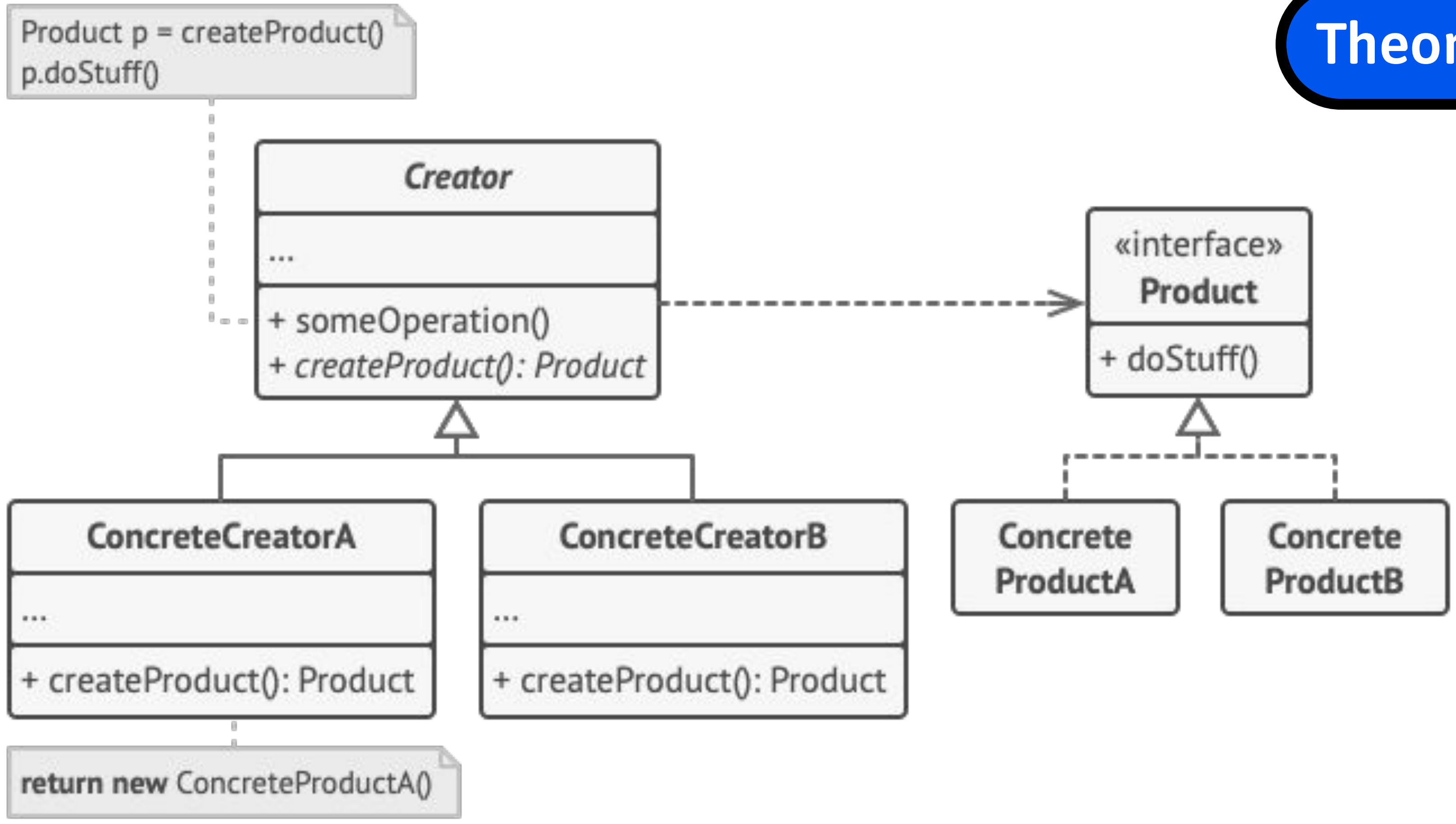
Logistics
...
+ planDelivery()
+ createTransport()
Transport t = createTransport()
RoadLogistics
...
+ createTransport()
SeaLogistics
...
+ createTransport()
return new Truck()
return new Ship()


«interface»
Transport
+ deliver()
Truck
...
+ deliver()
Ship
...
+ deliver()
Deliver by land in a box.
Deliver by sea in a container.


LogisticsApp

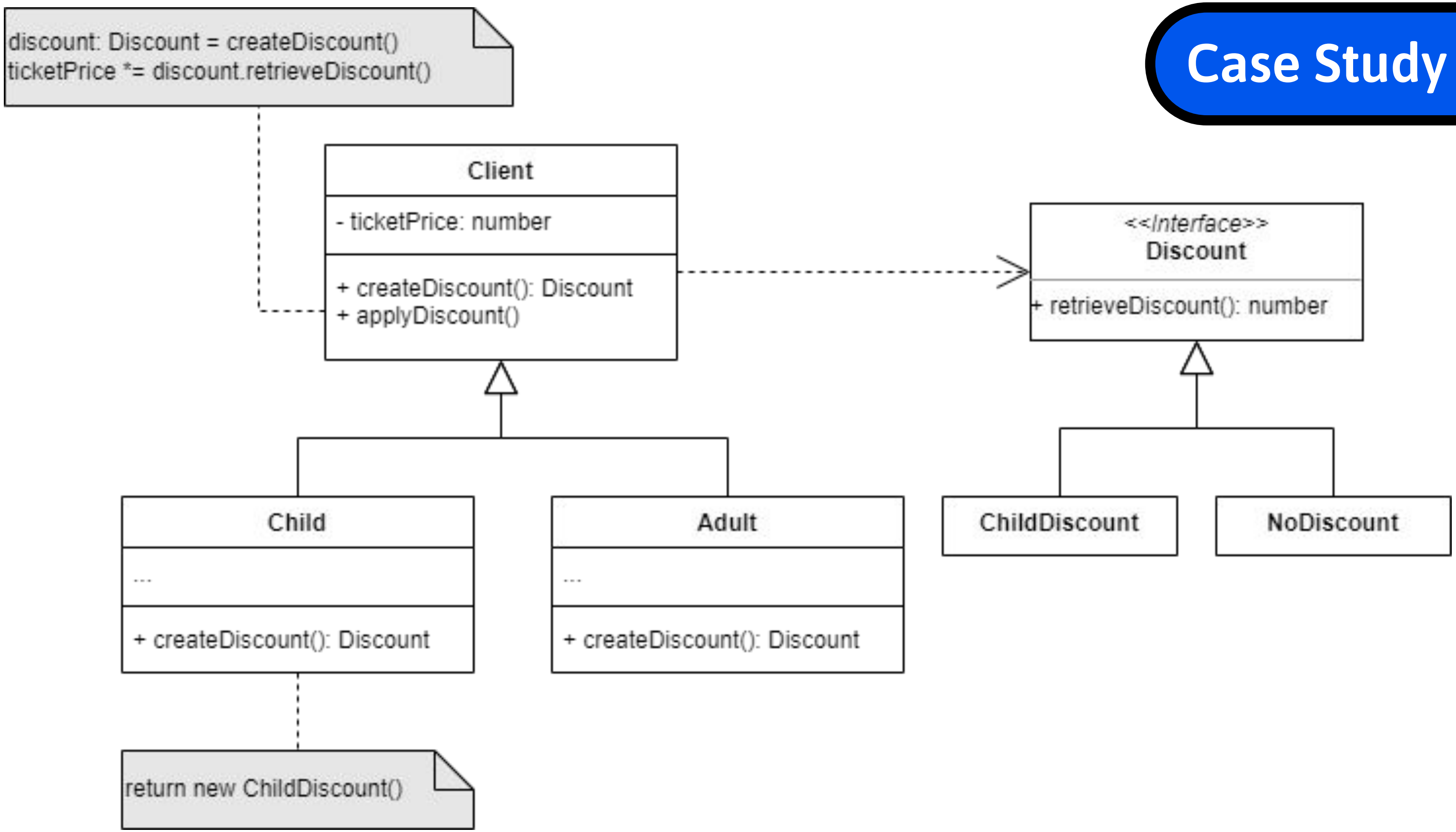

Transport
Transport
Transport
LogisticsApp
Transport
Transport
Transport
RoadLogistics
SeaLogistics

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

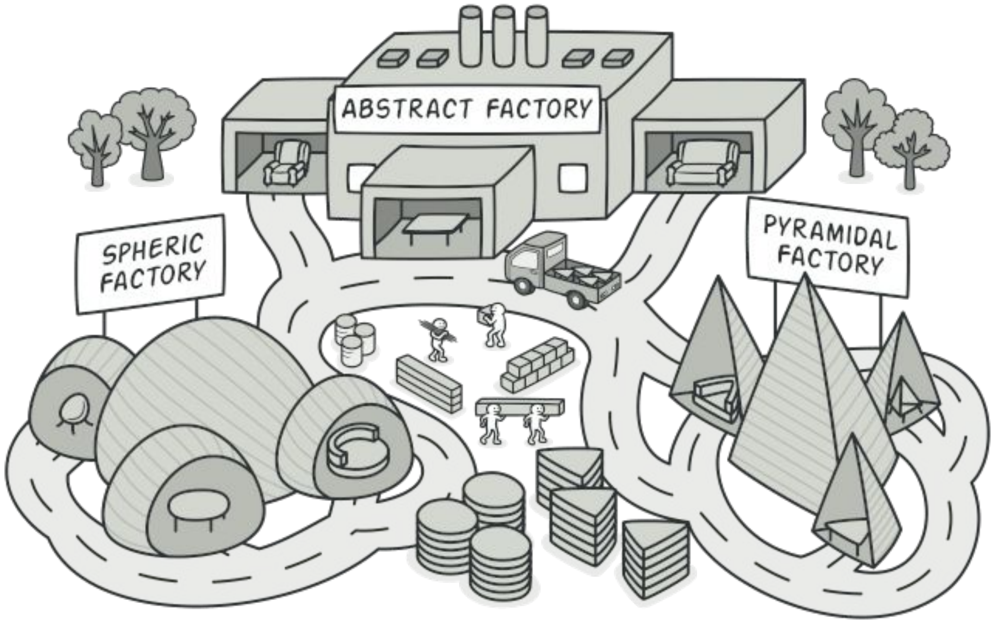
- You don’t know the types and dependencies your code should work with.
- To provide users a way to extend your library or framework.
- To save system resources by reusing existing objects.

Advantages and Disadvantages

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

- More complicated code (due to a lot of new subclasses).

Abstract Factory



Example

Furniture Shop

	Chair	Sofa	Coffee Table
Art Deco			
Victorian			
Modern			

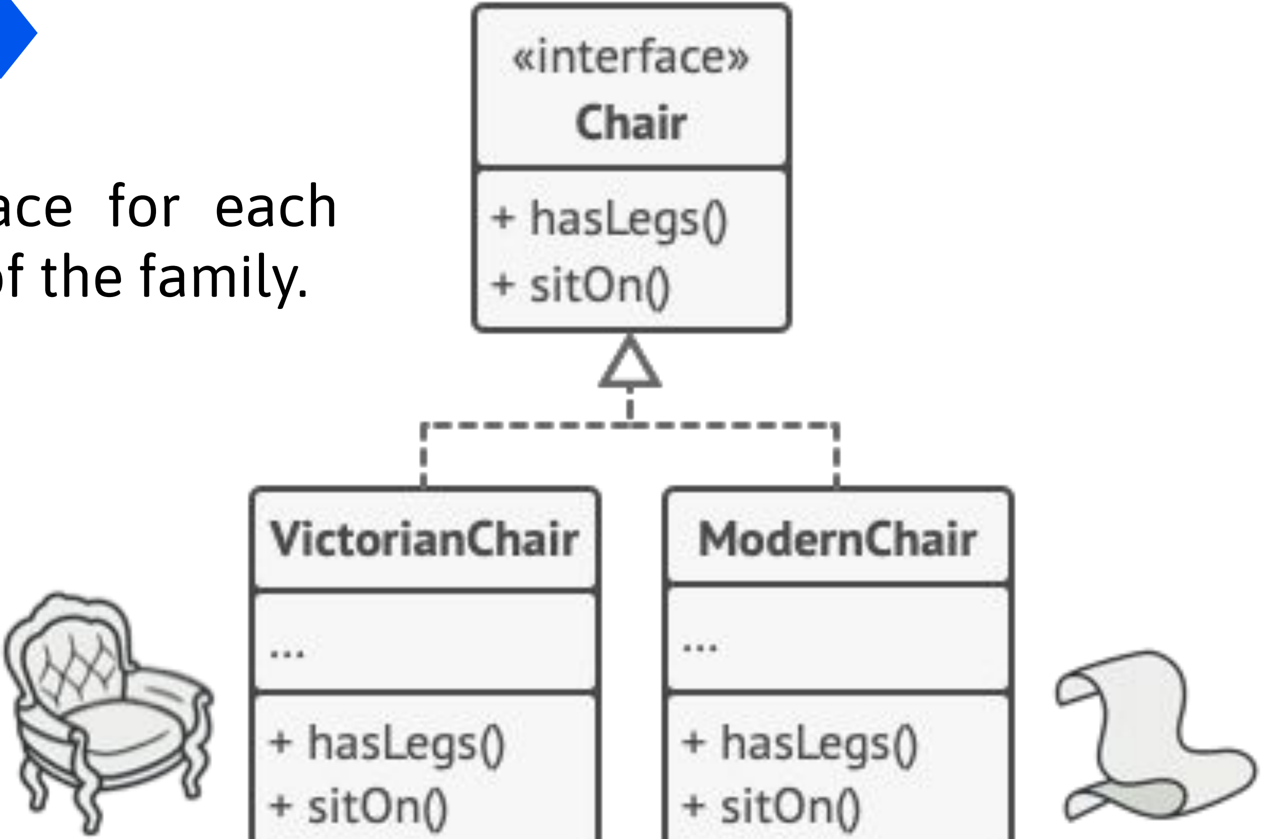
Problem

Currently you are running a furniture shop:

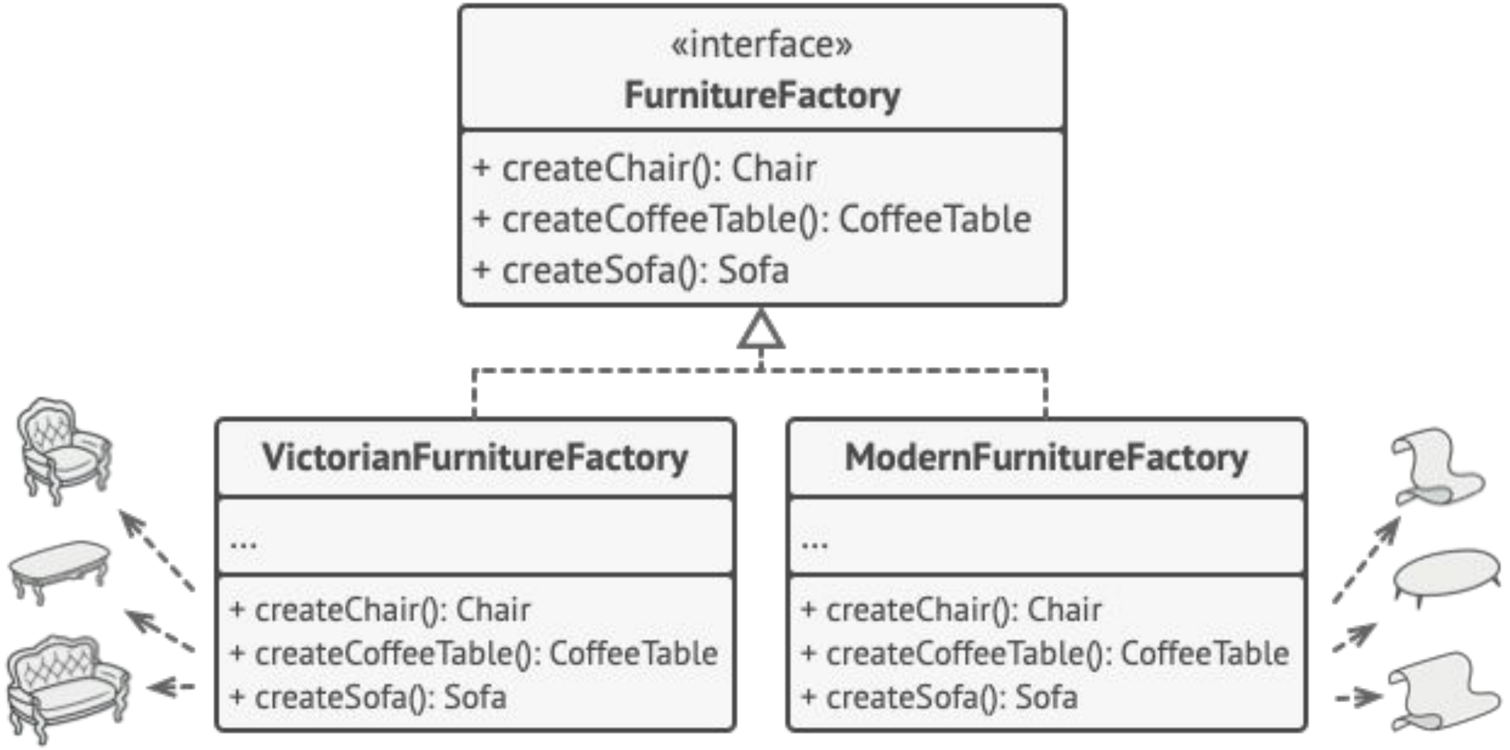
- You have a family of related furniture and several styles of this family.
- Customers get angry when they receive non-matching furniture.
- As your furniture catalog updates quite frequently, you don’t want to have to update the code every time.

Solution

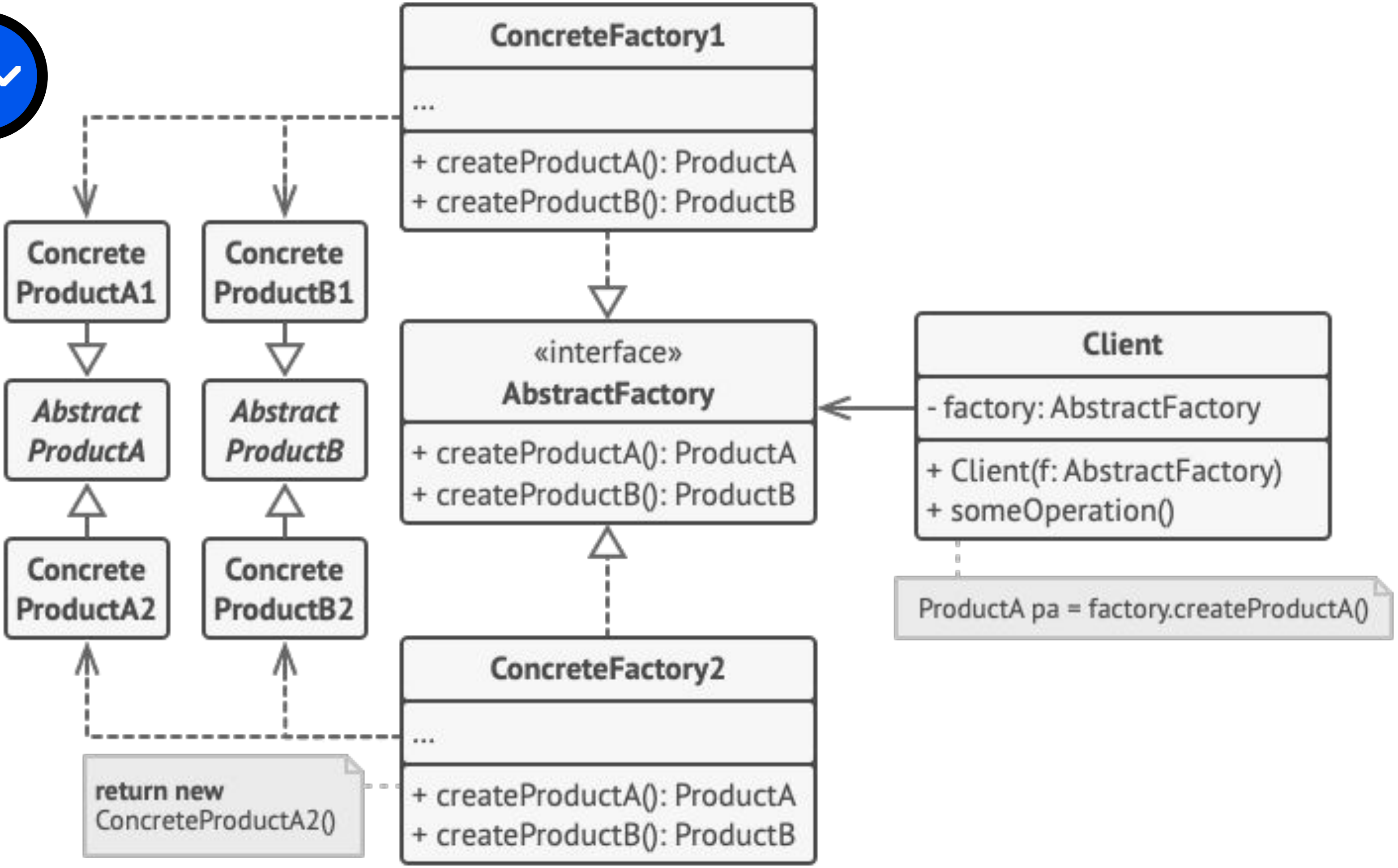
- Have an interface for each furniture piece of the family.



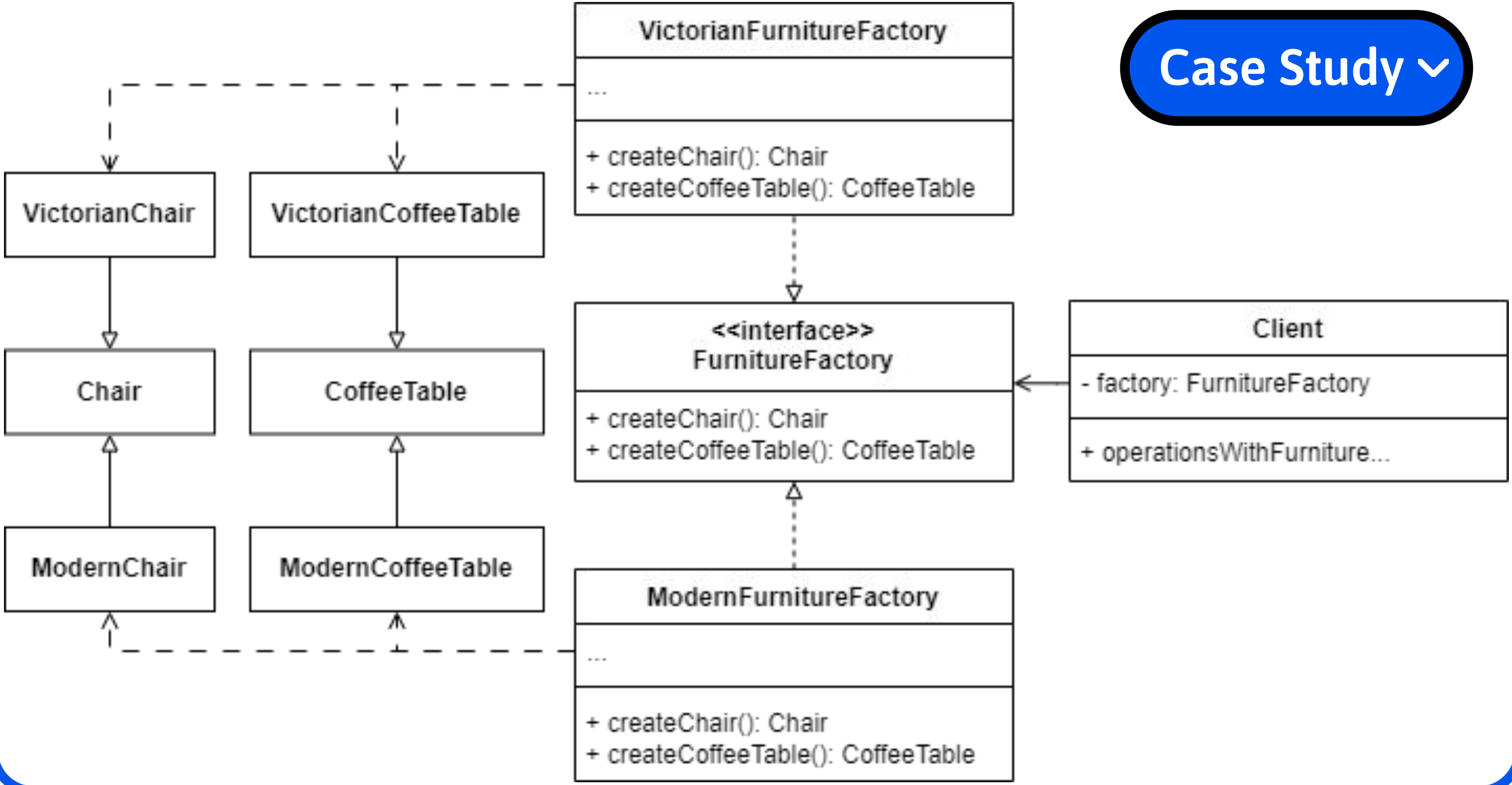
- Have an Abstract factory that produces each abstract piece of furniture.

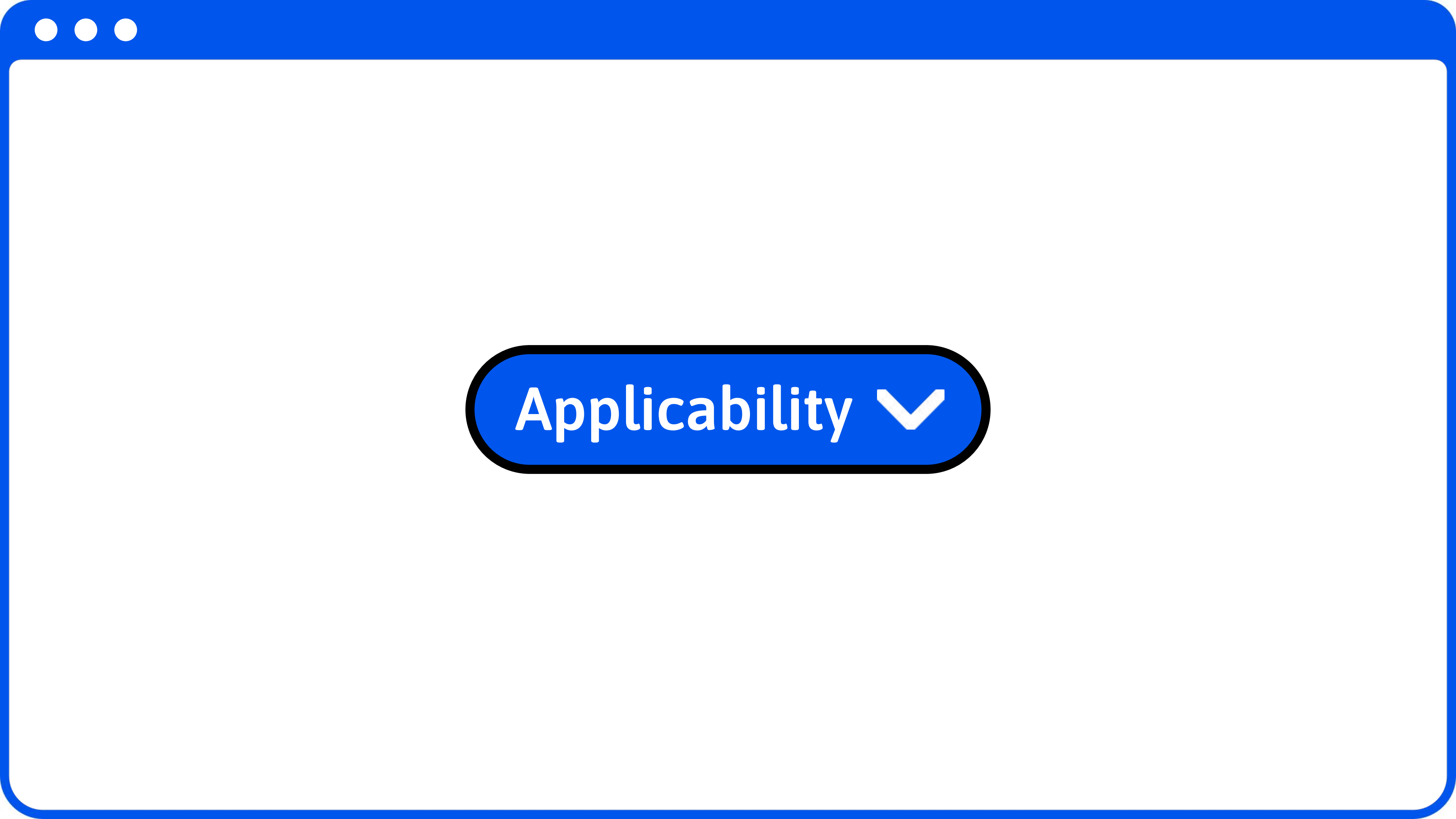


Theory


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


Applicability

- 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.

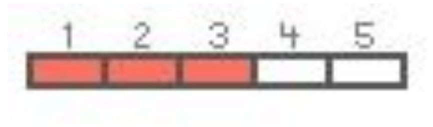
- When you have a class with a set of Factory Methods that blur its primary responsibility.

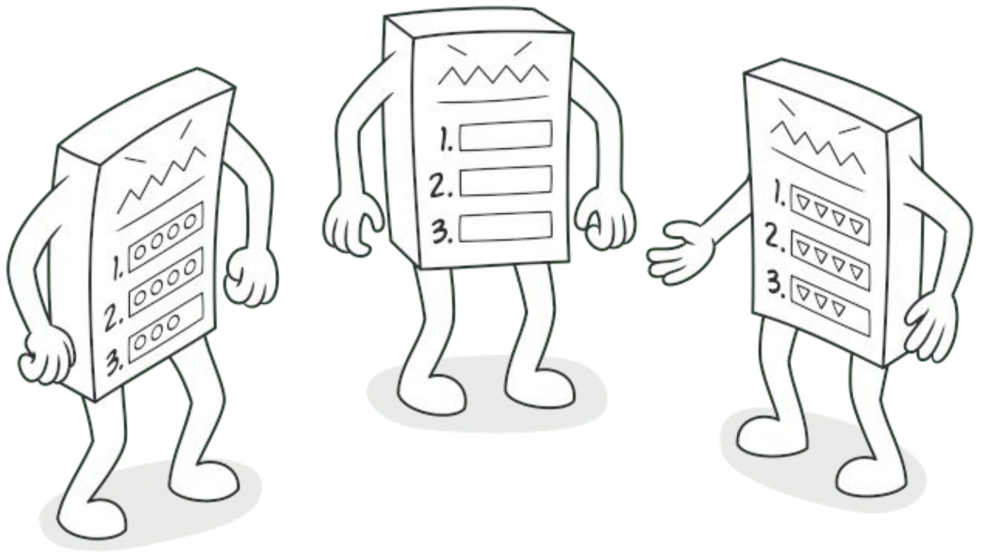
Advantages and Disadvantages

- You make sure that the products from a factory are compatible with each other.
- Client does not need to know the implementation of concrete products/factories.
- Single Responsibility Principle.
- Open/Closed Principle.

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

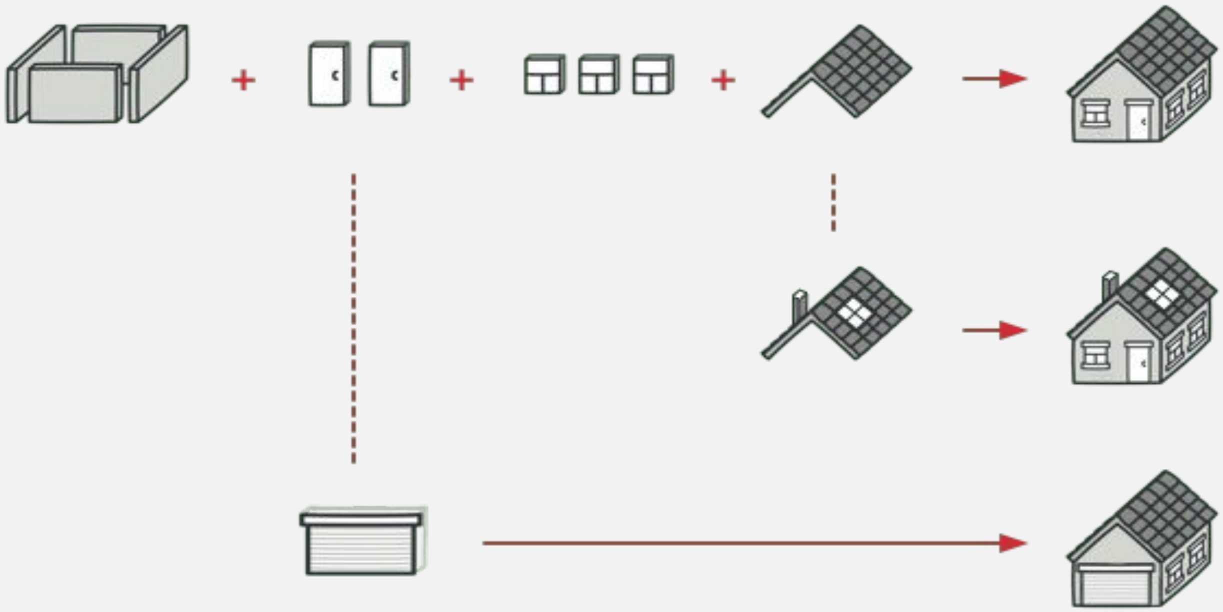
Template Method





Example

Mass Housing Construction

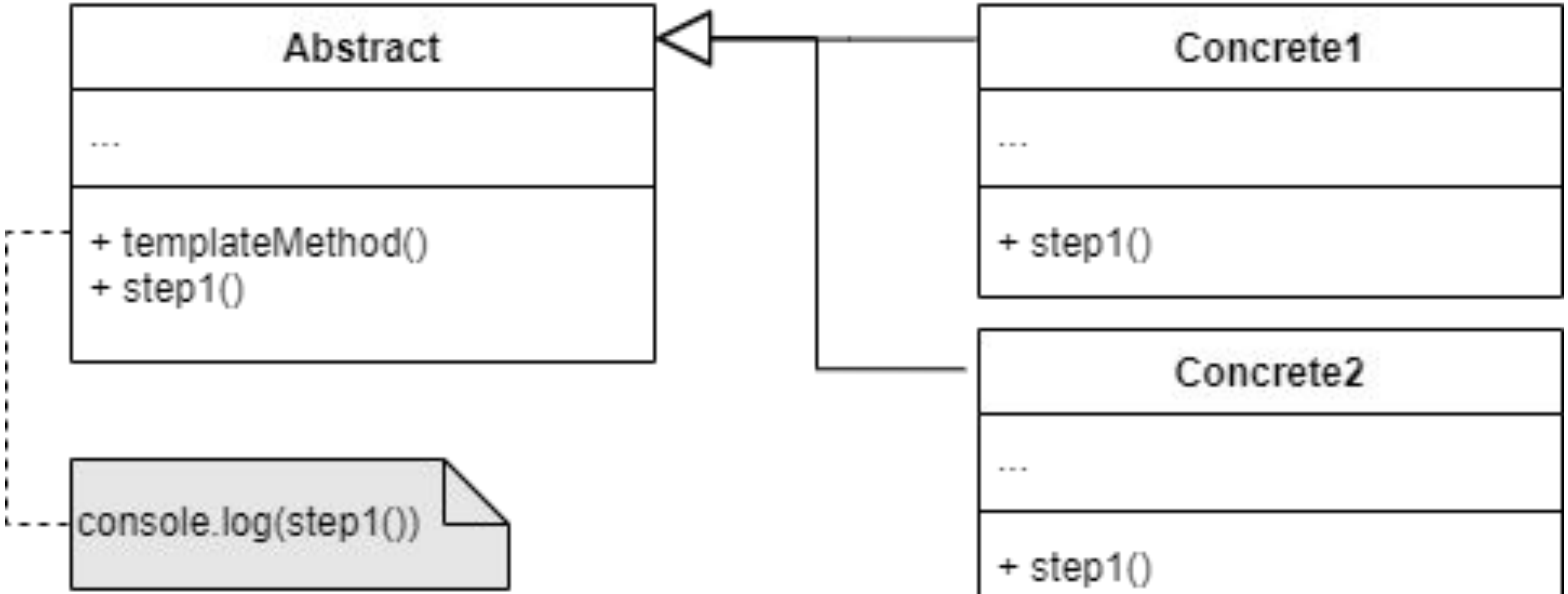


Problem

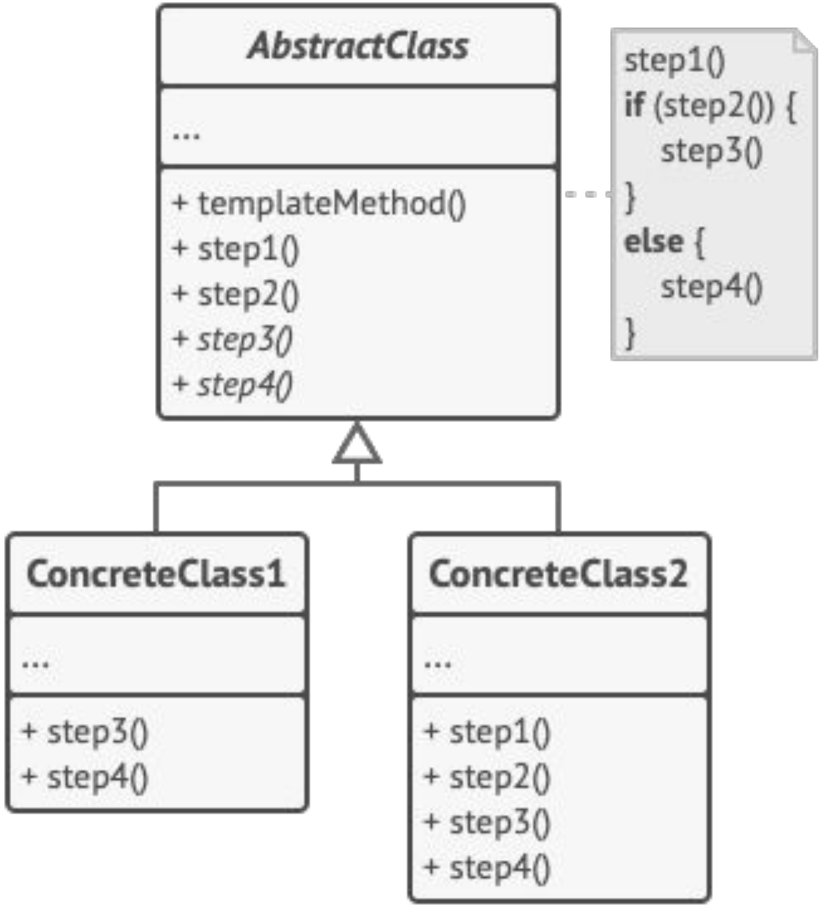
You’ve just started a company that builds houses:

- 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.

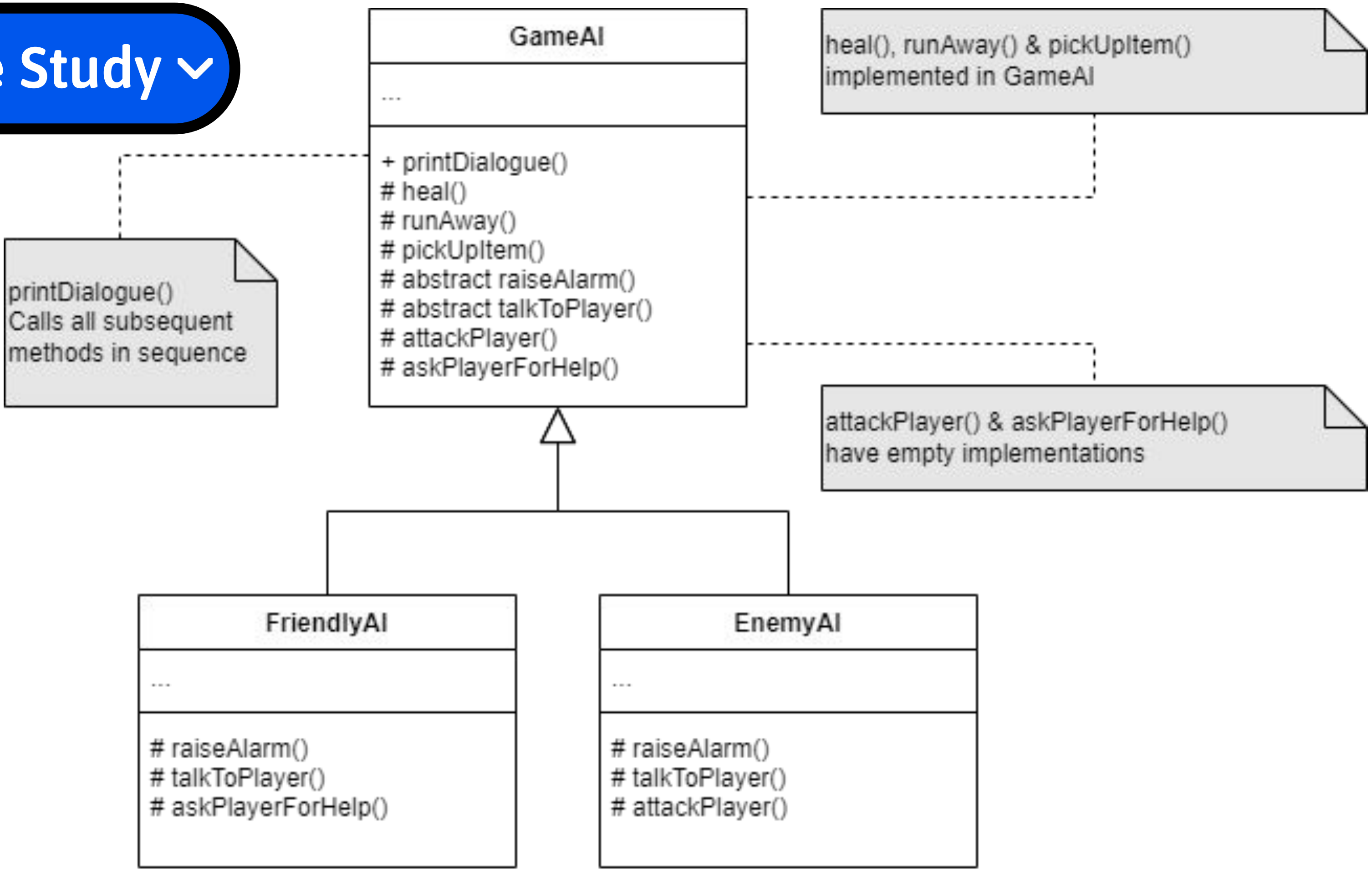
Solution

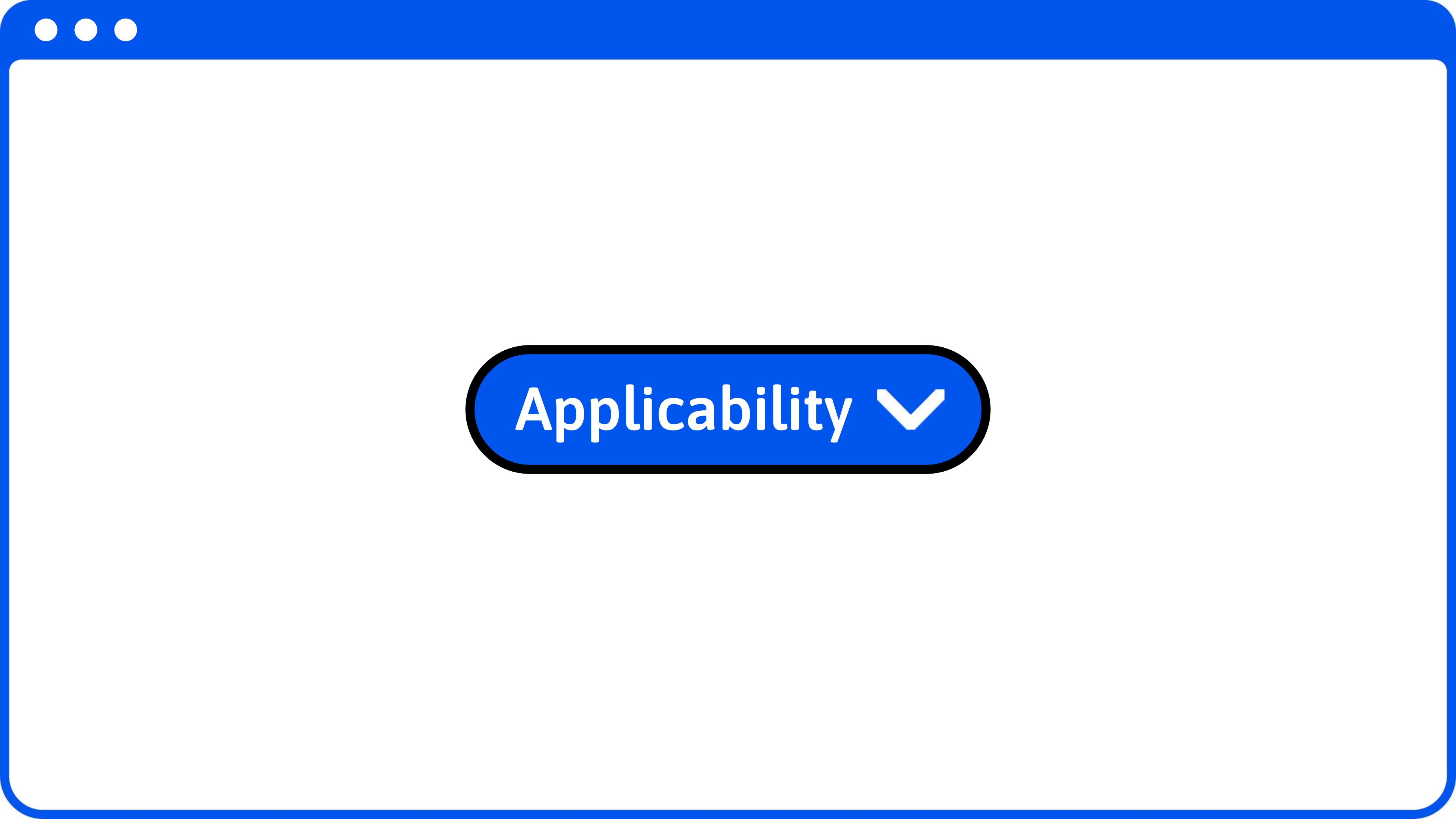

Abstract
...
+ templateMethod()
+ step1()
console.log(step1())
Concrete1
...
+ step1()
Concrete2
...
+ step1()

Theory


AbstractClass
...
+ templateMethod()
+ step1()
+ step2()
+ step3()
+ step4()
step1()
if (step2()) {
    step3()
}
else {
    step4()
}
ConcreteClass1
...
+ step3()
+ step4()
ConcreteClass2
...
+ step1()
+ step2()
+ step3()
+ 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 and Disadvantages

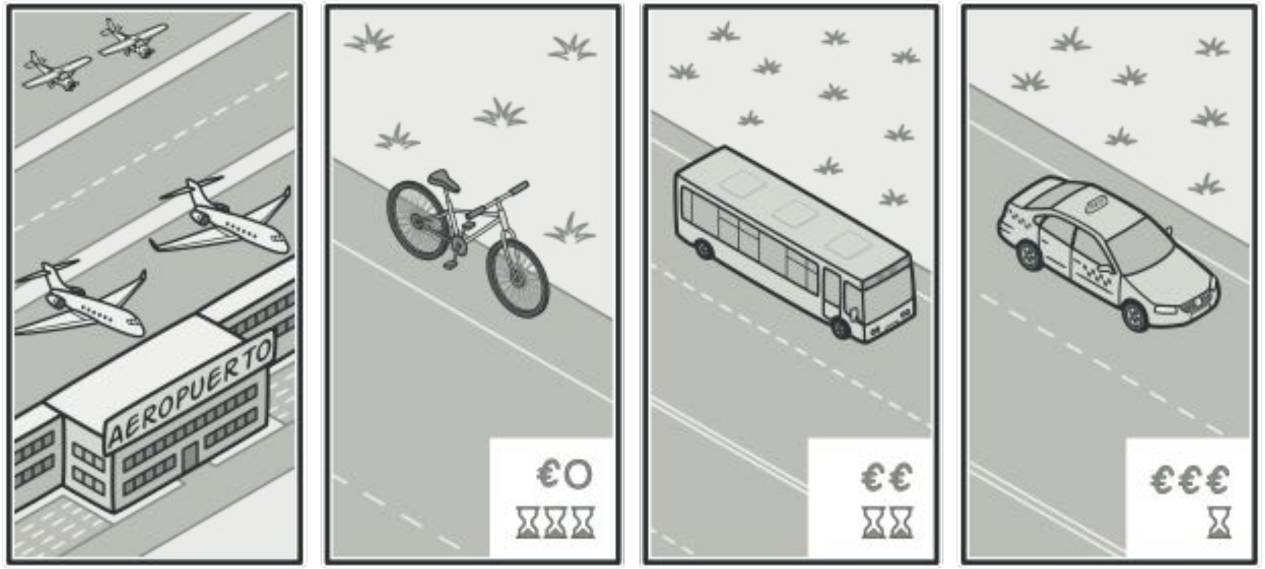
- Clients can only override certain parts of an algorithm, making them less affected by changes that happen to other parts of the algorithm.
- You can move the duplicate code into a superclass.

- Clients may be limited by the template provided.
- You might violate Liskov’s Substitution Principle.
- Template methods tend to be harder to maintain the more steps they have.

Strategy

Example

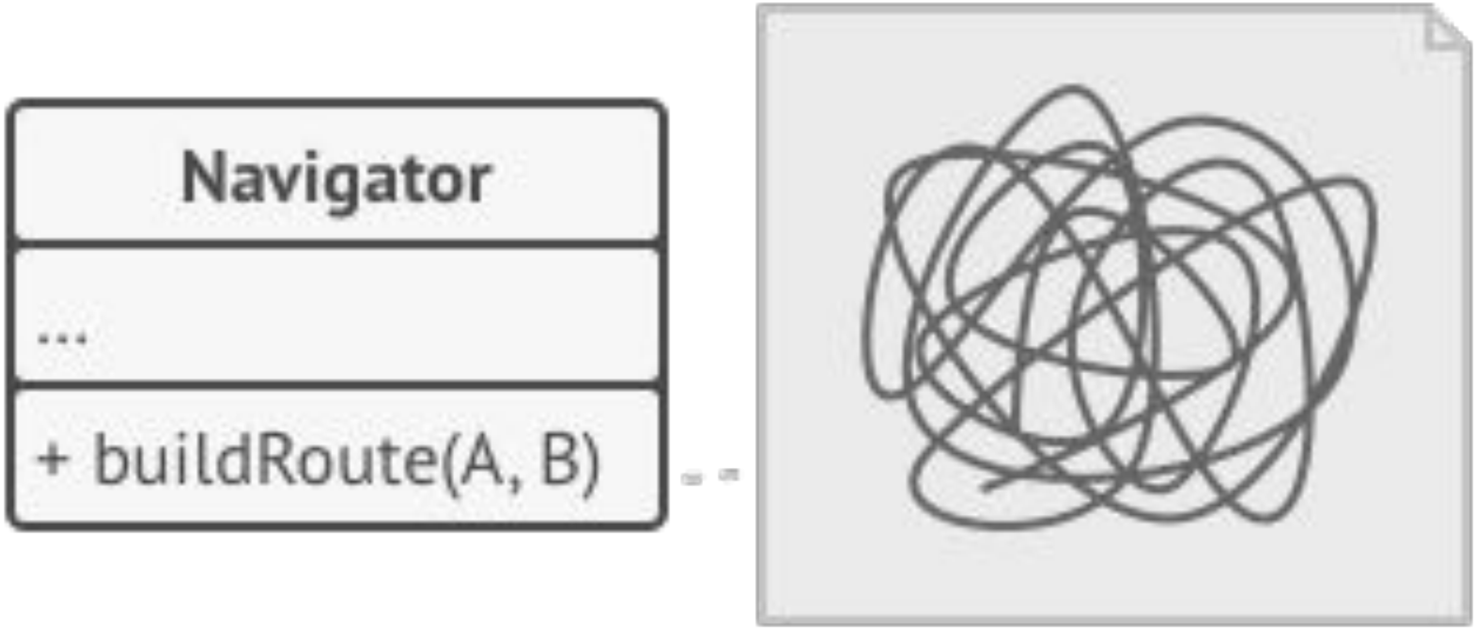
Navigation App



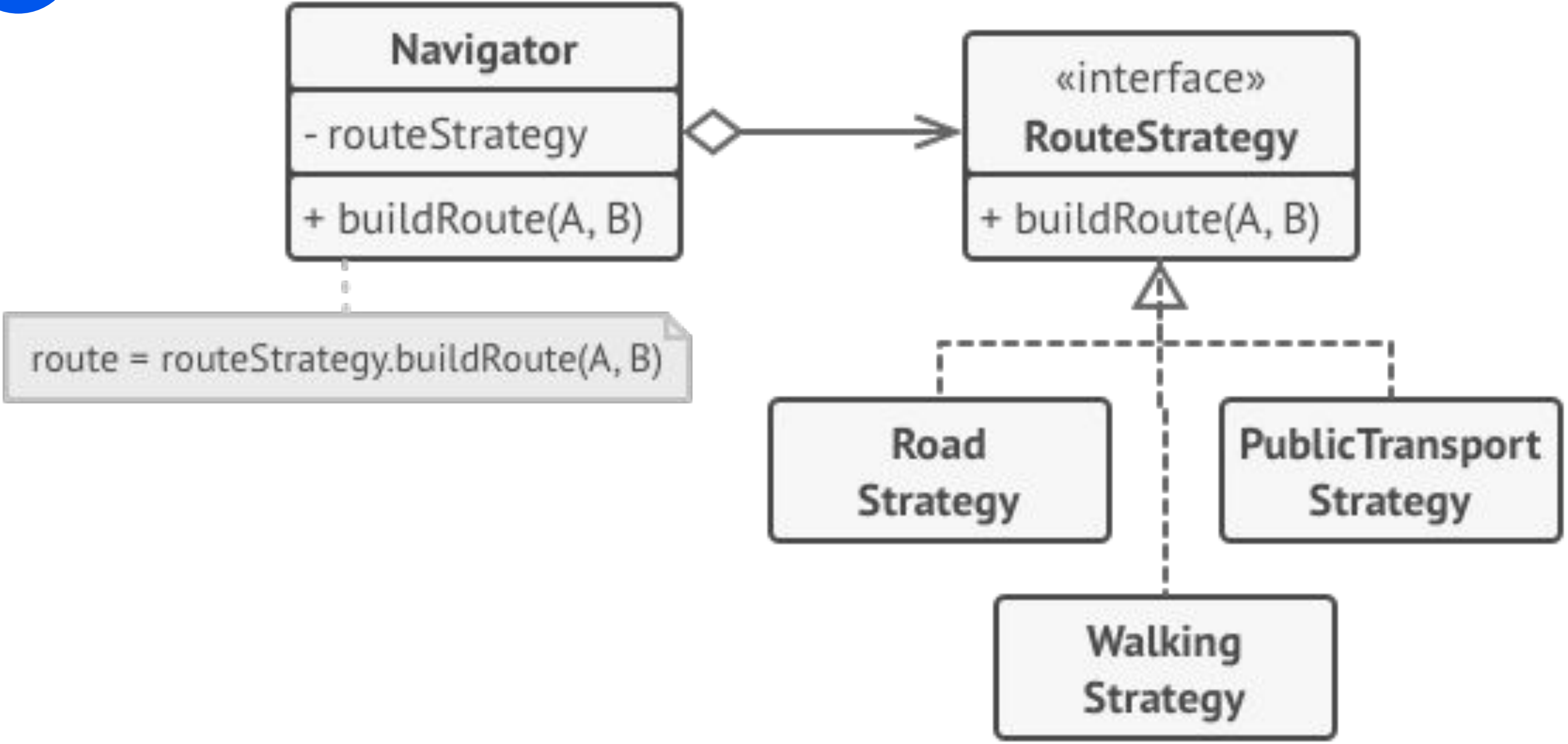
Problem

You are the creator of a navigation app for tourists:

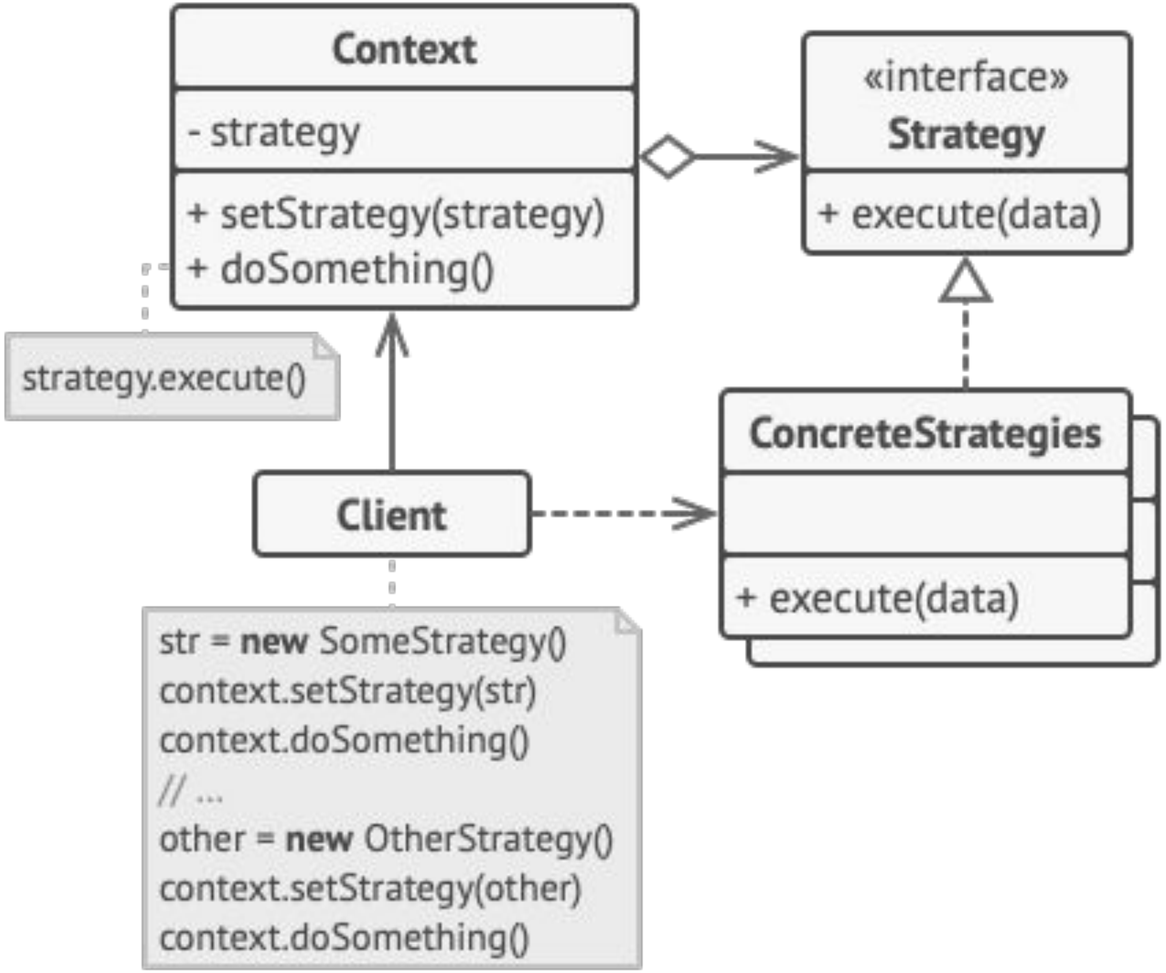
- You implemented a feature for automatic route planning, which only supported one type of navigation in the first version.
- Started implementing more and more types of navigation to the route planner, making the navigator very hard to maintain.
- Team work started to become inefficient as most of the time went to fixing integration conflicts.


Navigator
...
+ buildRoute(A, B)

Solution

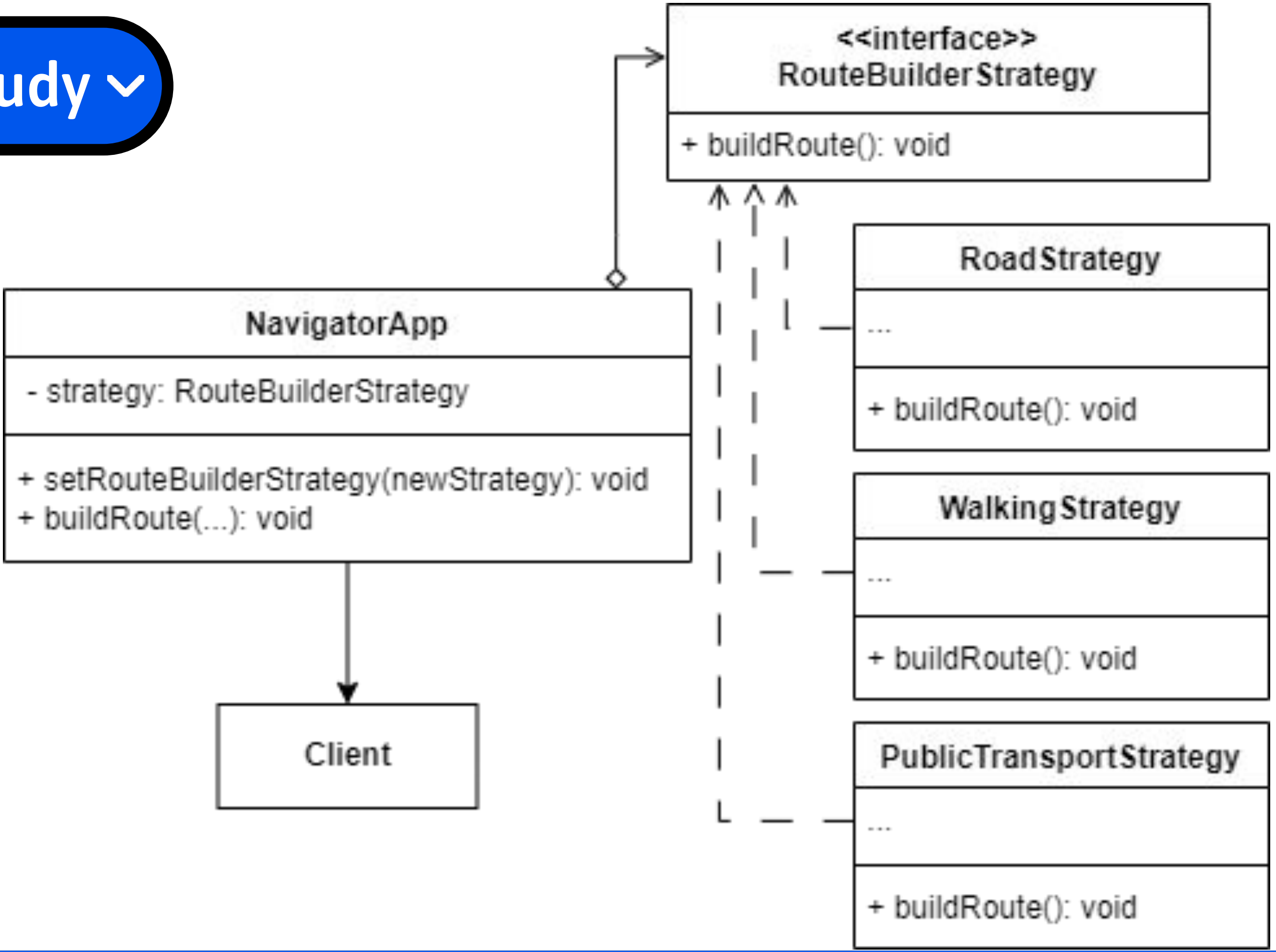

Navigator
- routeStrategy
+ buildRoute(A, B)
route = routeStrategy.buildRoute(A, B)
«interface»
RouteStrategy
+ buildRoute(A, B)
Road Strategy
PublicTransport Strategy
Walking Strategy

Theory

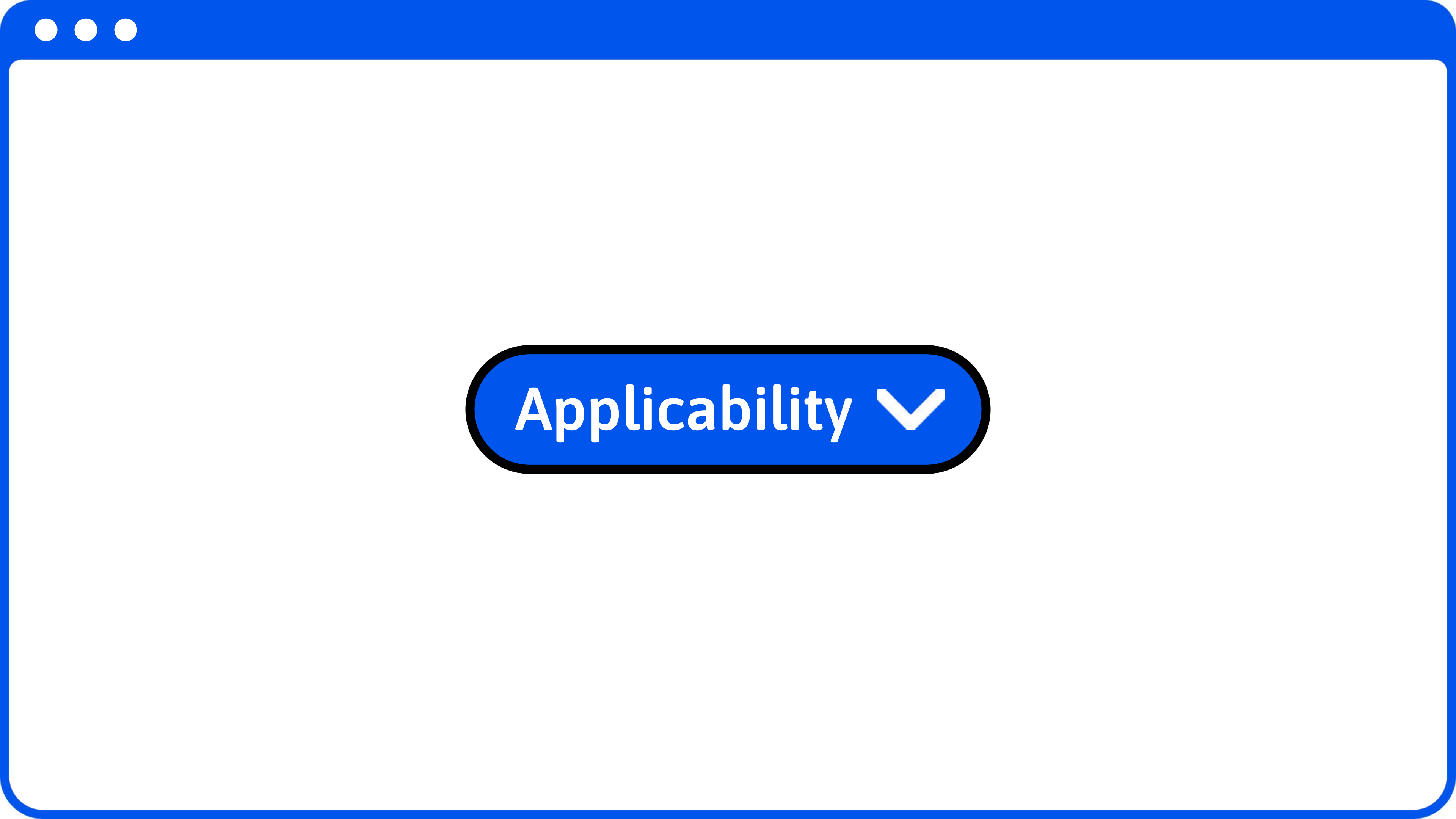


```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

Case Study



<<interface>>
RouteBuilderStrategy
+ buildRoute(): void
NavigatorApp
- strategy: RouteBuilderStrategy
+ setRouteBuilderStrategy(newStrategy): void
+ buildRoute(...): void
Client
Road Strategy
...
+ buildRoute(): void
Walking Strategy
...
+ buildRoute(): void
PublicTransportStrategy
...
+ buildRoute(): void


Applicability

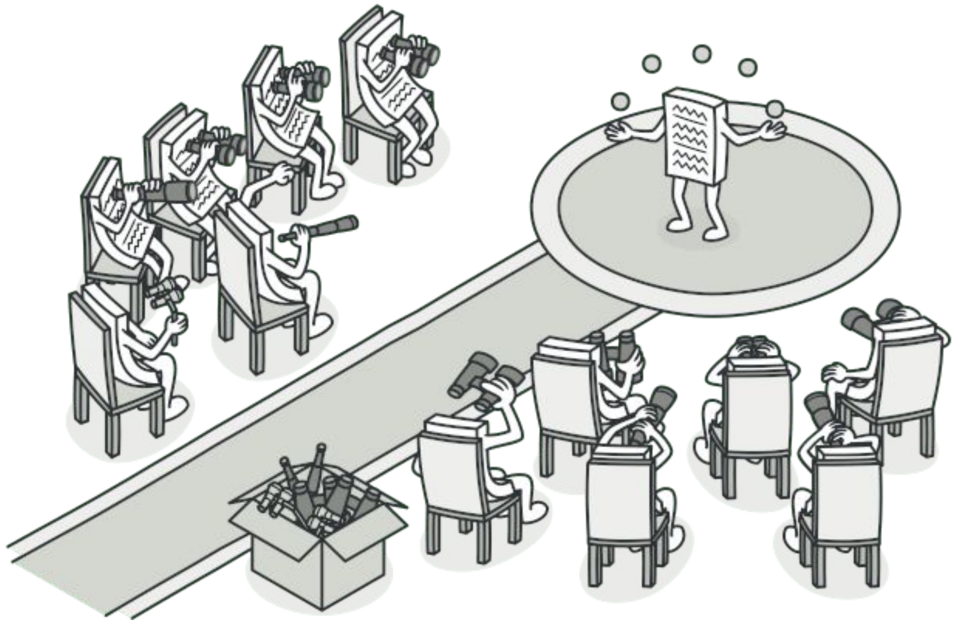
- Use different variants of an algorithm within an object and be able to switch from one algorithm to another during runtime.
- When you have a lot of similar classes that only differ in the way they execute some behavior.
- To isolate algorithms from the rest of the code.
  The clients will interact with them through an interface.
- When your class has a massive conditional statement that switches between different variants of the same algorithm.

Advantages and Disadvantages

- Swap algorithms used at runtime.
- Isolate the implementation details of an algorithm from the code that uses it.
- Replace inheritance with composition.
- Open/Closed Principle.

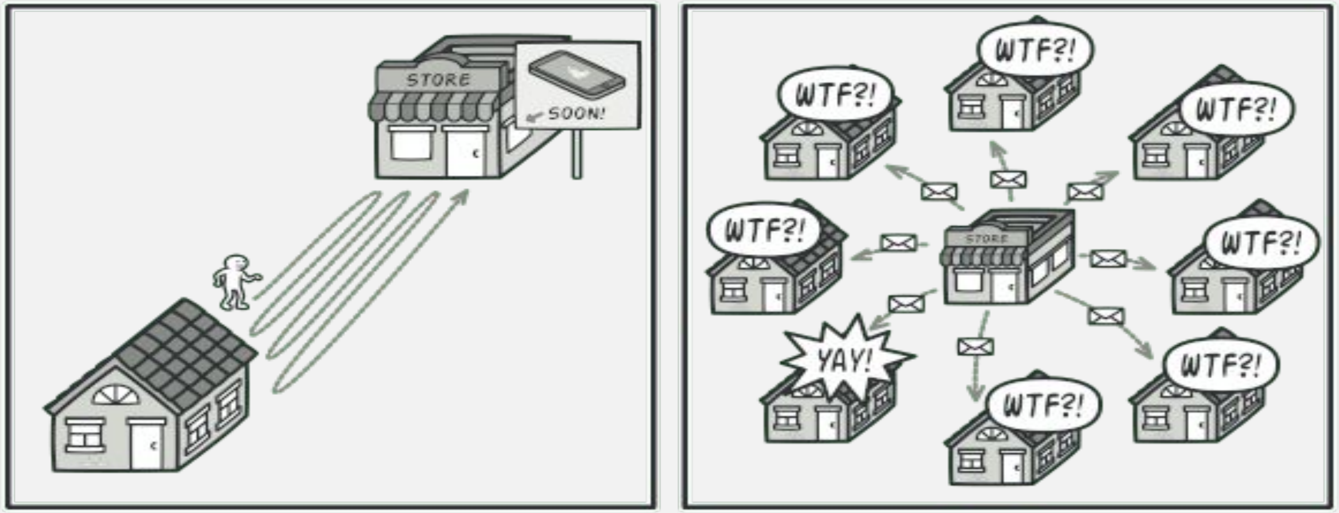
- Overcomplicates the program if you only have a couple of rarely changing algorithms.
- Clients must be aware of the differences between strategies.
- Can be replaced by a set of functions, which doesn’t bloat the code.

Observer



Example

Store & Spam

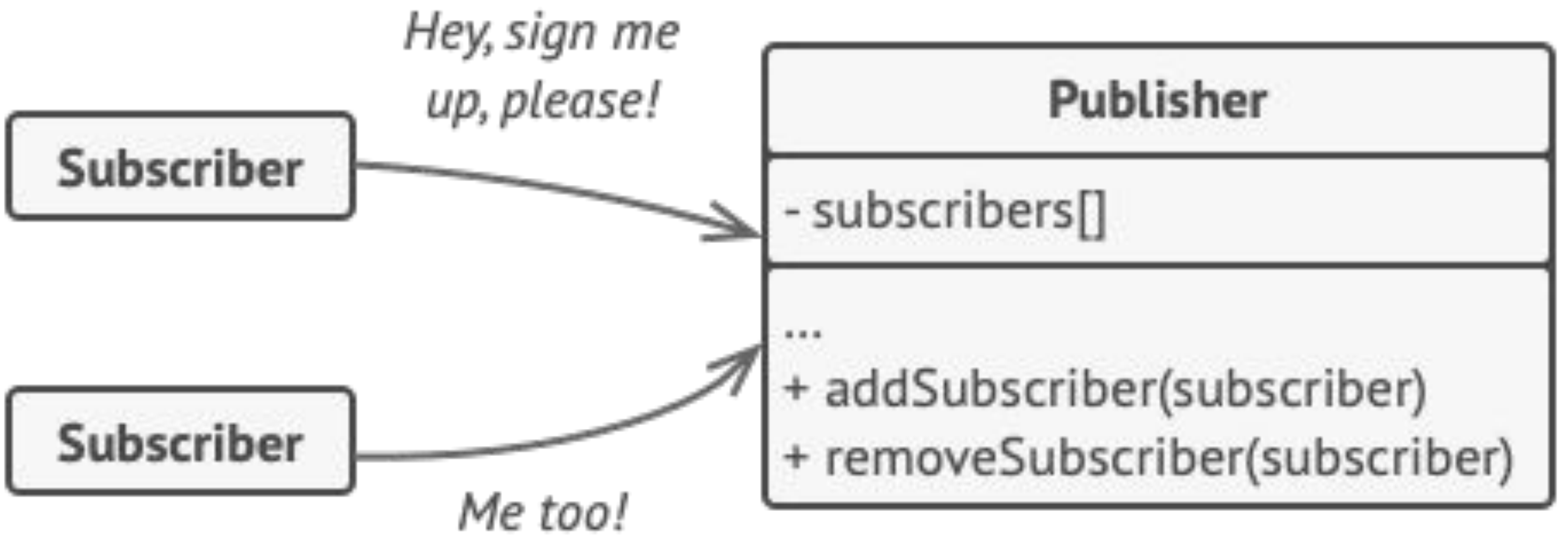


Problem

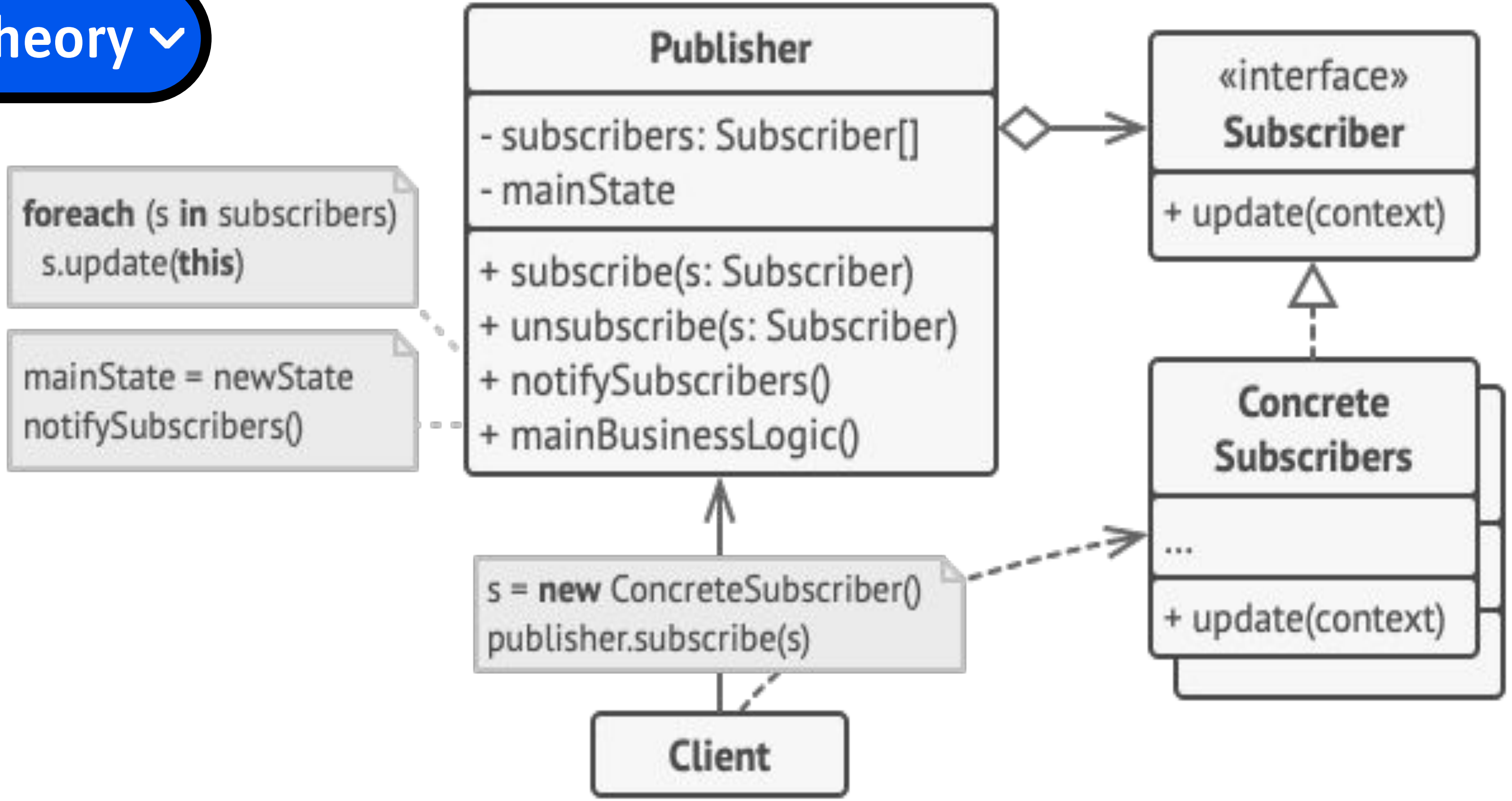
A customer is interested in buying a product from the store, which leaves us with two possible options:

- 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).

Solution


Hey, sign me up, please!
Subscriber
Subscriber
Me too!
Publisher
- subscribers[]
...
+ addSubscriber(subscriber)
+ removeSubscriber(subscriber)

Theory

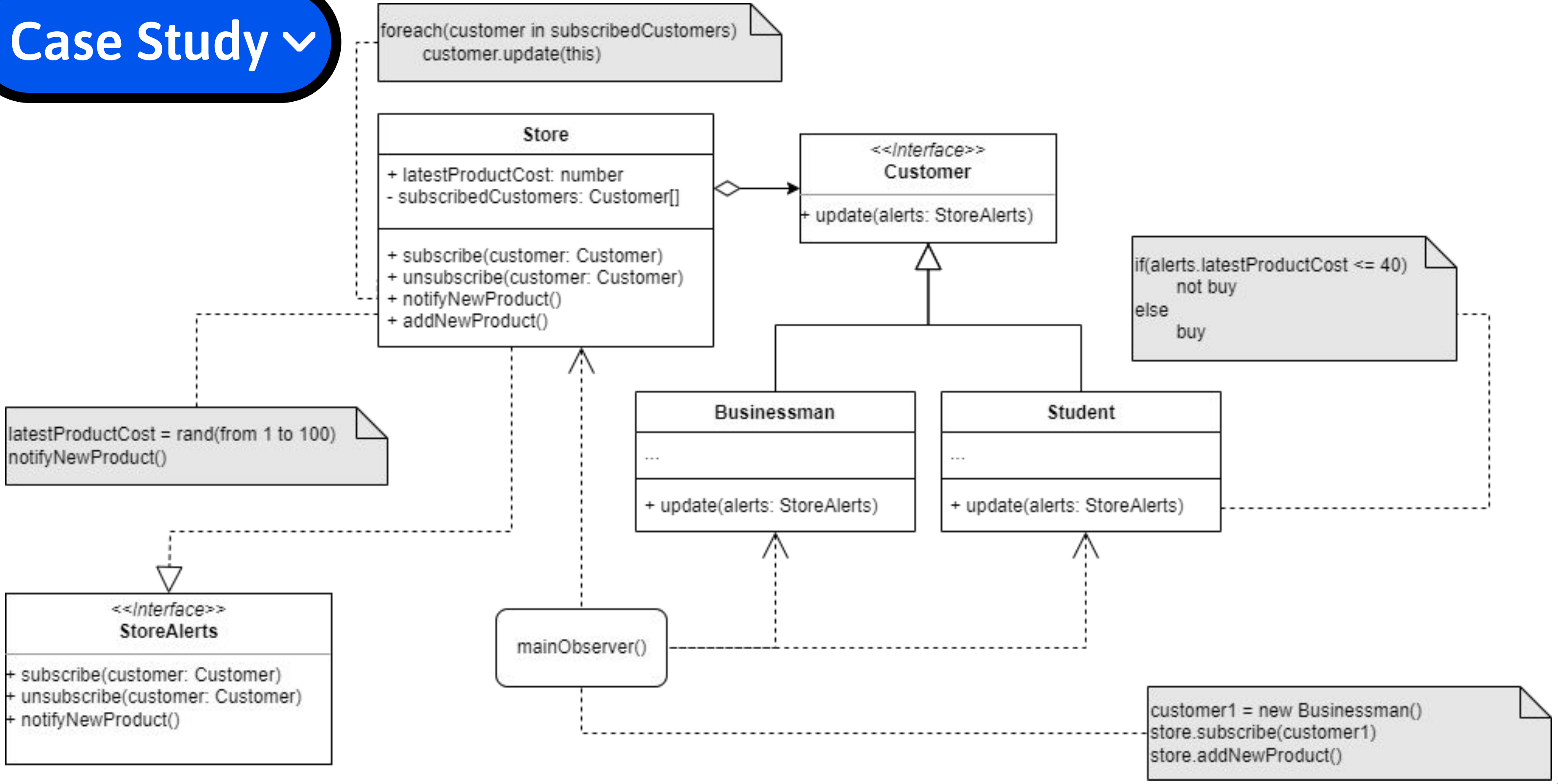


```
foreach (s in subscribers)
  s.update(this)
```

```
mainState = newState
notifySubscribers()
```

```
s = new ConcreteSubscriber()
publisher.subscribe(s)
```

Case Study


foreach(customer in subscribedCustomers)
customer.update(this)
Store
+ latestProductCost: number
- subscribedCustomers: Customer[]
+ subscribe(customer: Customer)
+ unsubscribe(customer: Customer)
+ notifyNewProduct()
+ addNewProduct()
<<Interface>>
Customer
+ update(alerts: StoreAlerts)
if(alerts.latestProductCost <= 40)
not buy
else
buy
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 and Disadvantages

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

- 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
- Design Patterns in C#

Any questions?