الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

المحاضرة الرابعة | كلية الهندسة المعلوماتية | مقرر تصميم نظم البرمجيات

Design Patterns: Template Method Pattern, State Pattern

د. رياض سنبل

Design Patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*

Factory Method	Abstract Factory	Singleton
Builder	Prototype	

- **Structural Patterns** *(how objects/classes can be combined)*

Adapter	Bridge	Composite
Decorator	Facade	Flyweight
Proxy		

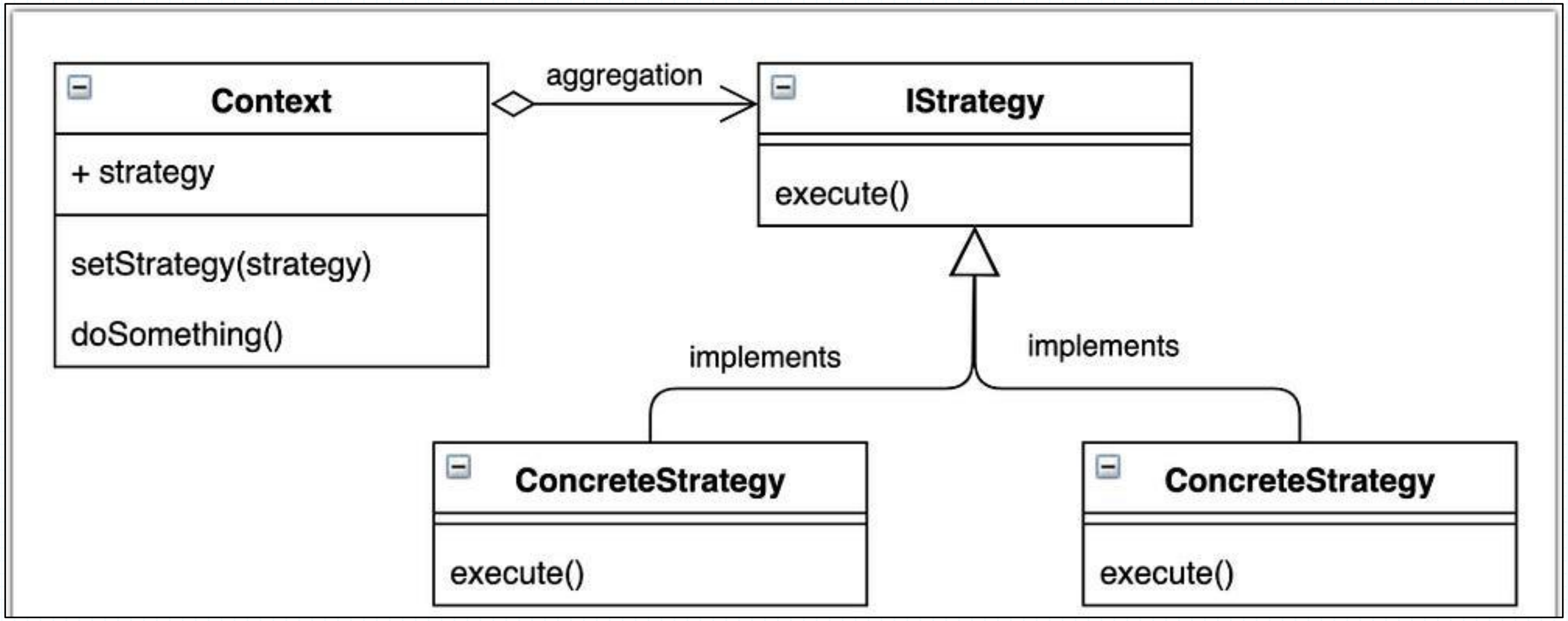
- **Behavioral Patterns** *(communication between objects)*

Command	Interpreter	Iterator
Mediator	Observer	State
Strategy	Chain of Responsibility	Visitor
$\boxed{\text{Template Method}}$		

Template Method Pattern

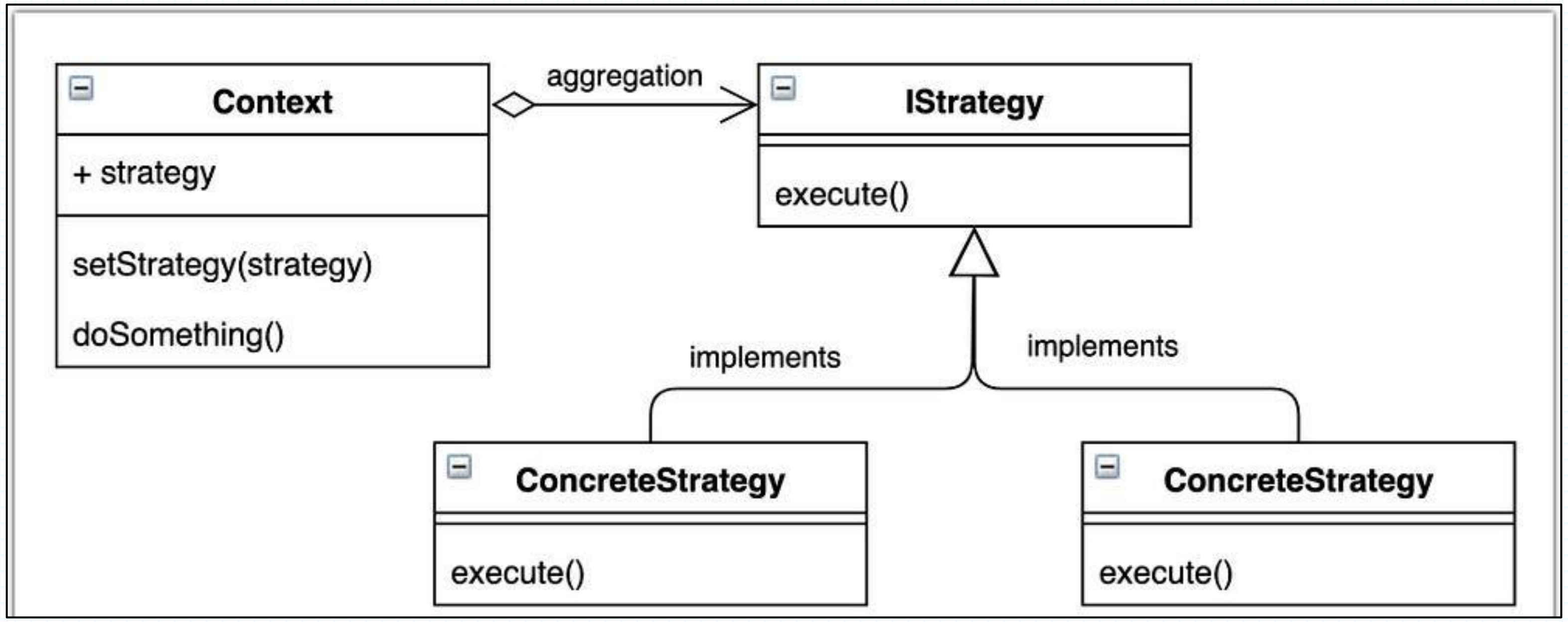
TEMPLATE METHOD PATTERN

Strategy Pattern



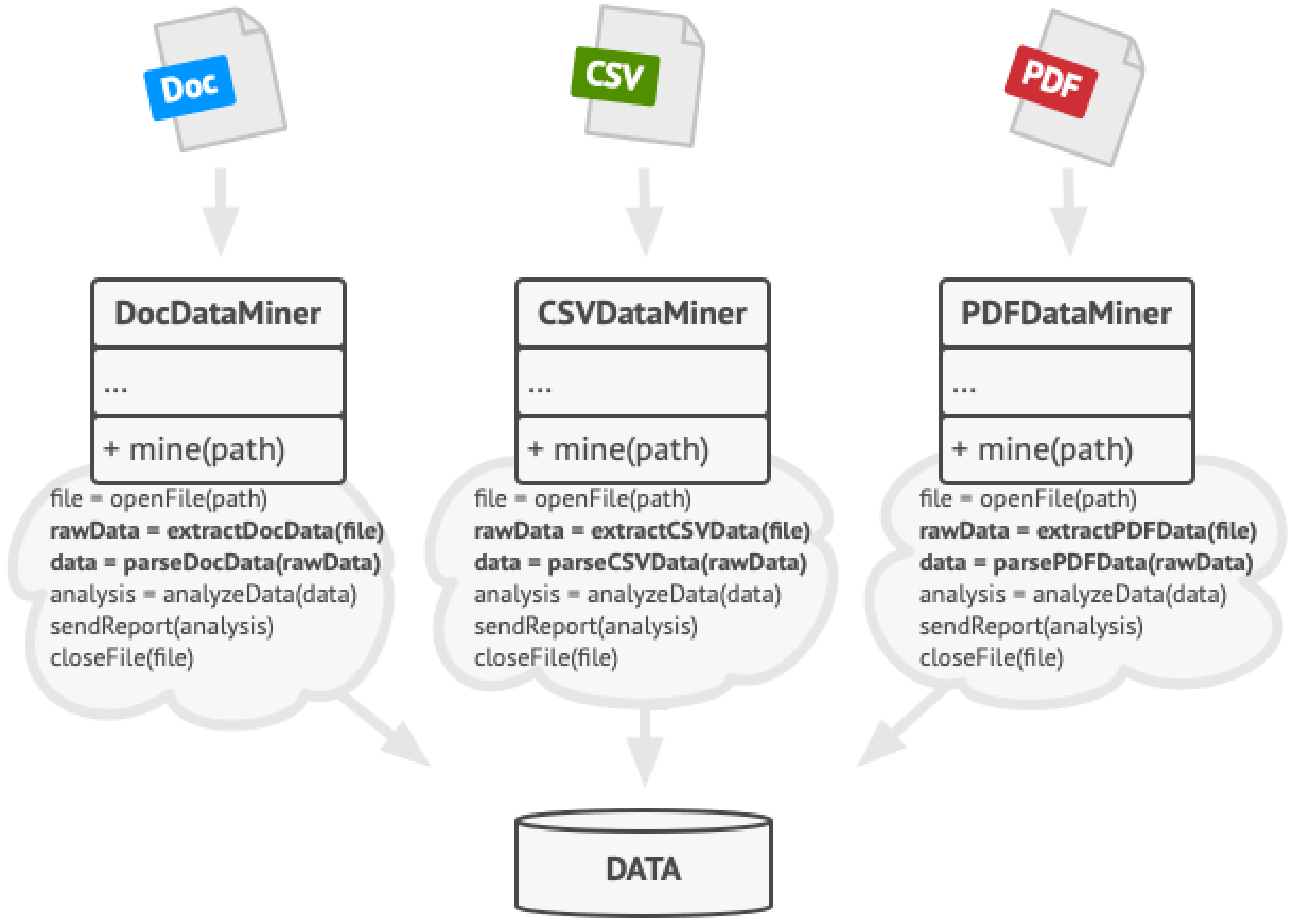
Strategy Pattern.. What if?

- In Strategy patter, IStrategy is an Interface
- i.e. each concreate Stategy has its own implementation..
- What if these strategies have some **common** ("shared") steps!



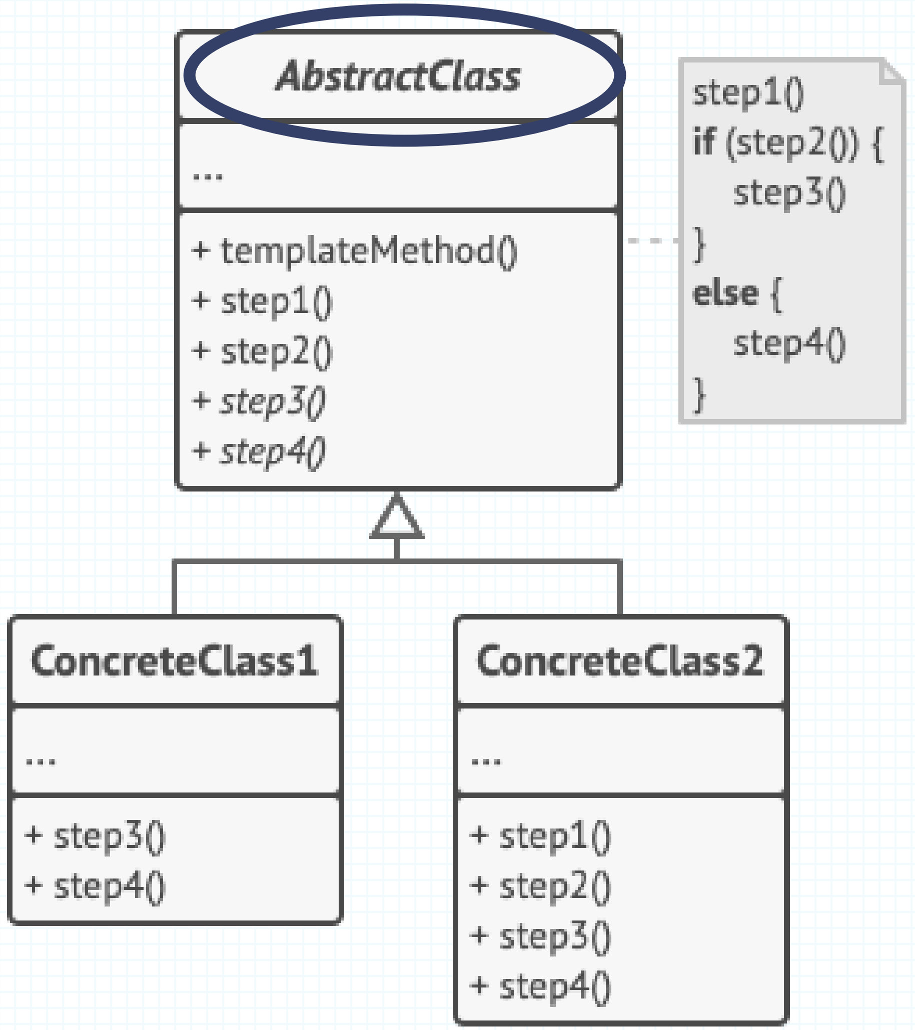
Example

- Imagine that you’re creating a data mining application that analyzes corporate documents. Users feed the app documents in various formats (PDF, DOC, CSV), and it tries to extract meaningful data from these docs in a uniform format.



Template Method Pattern

- **Template Method** is a behavioral design pattern that defines the skeleton of an algorithm in the superclass but lets **subclasses override** specific steps of the algorithm without changing its structure.

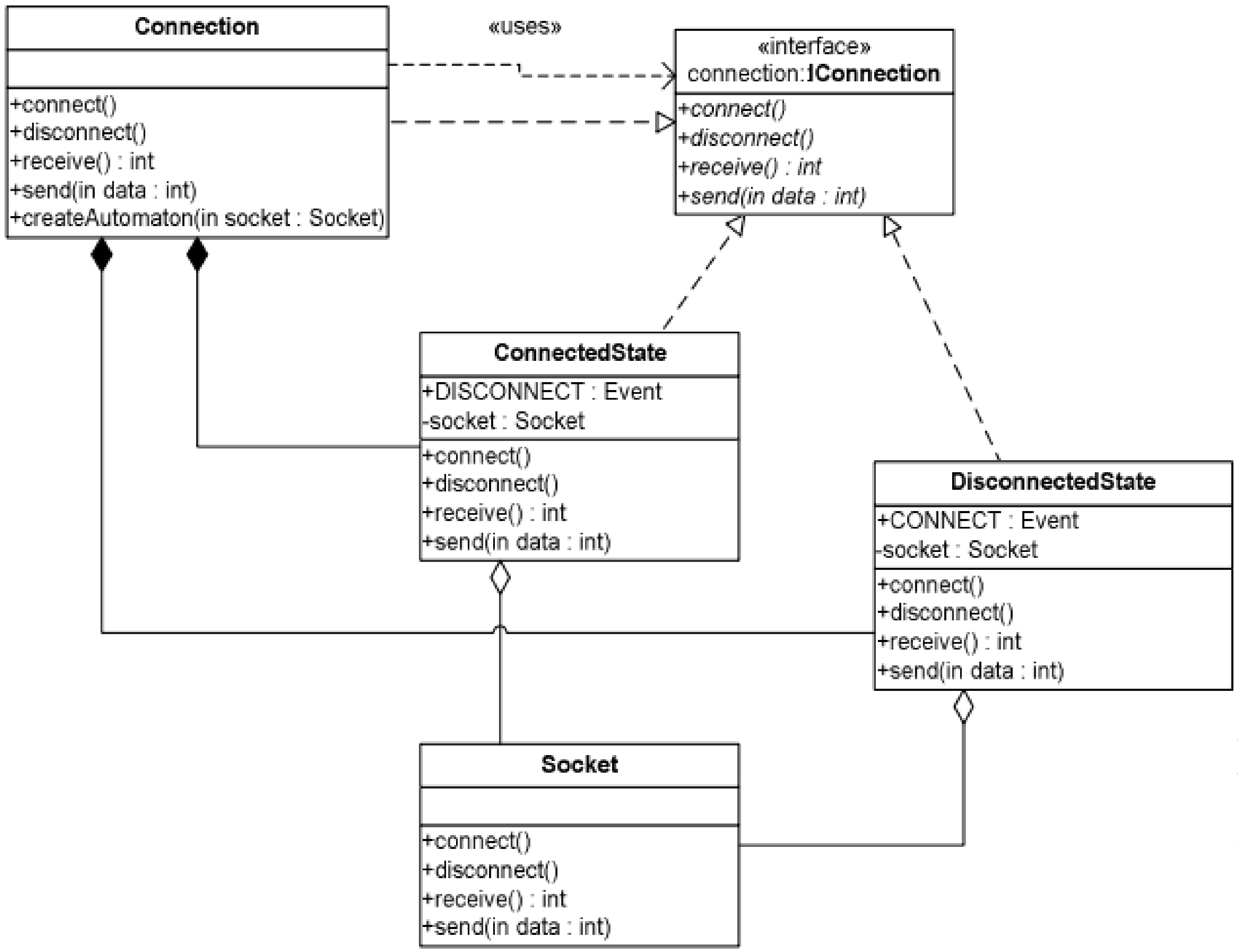


State Pattern

The Problem

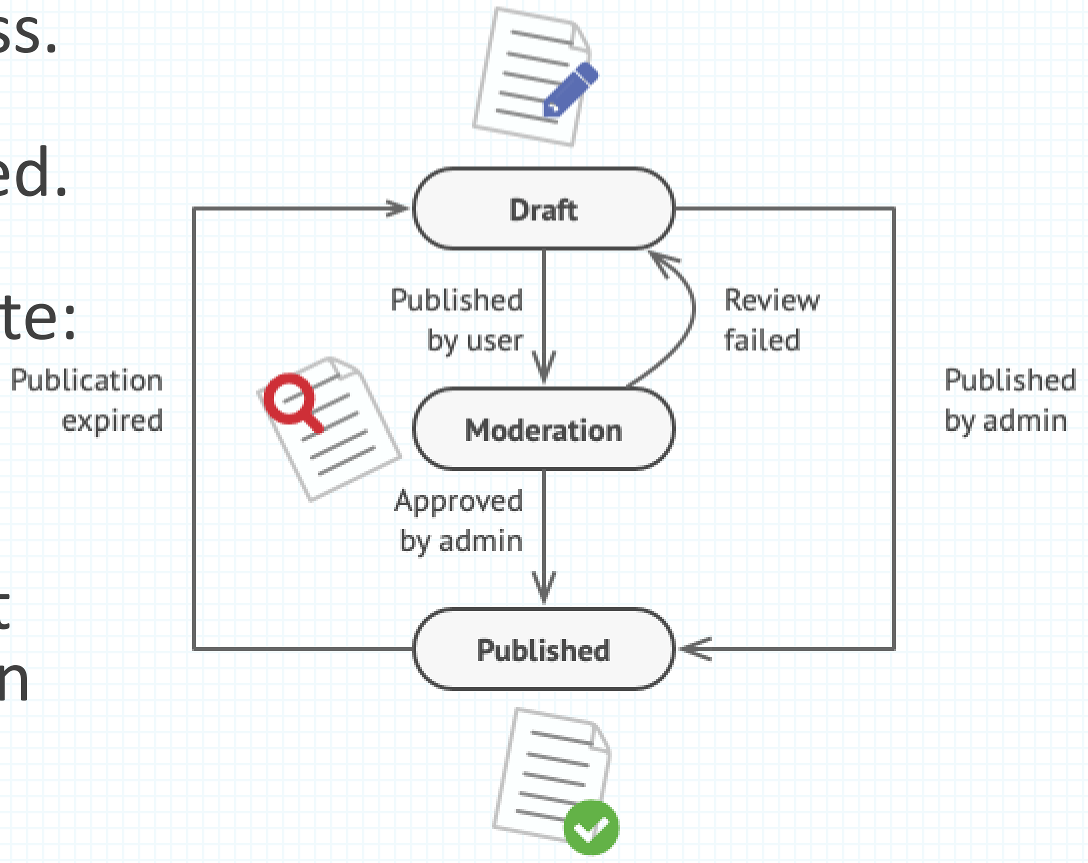
- An object's behavior changes based on its internal state!
- Operations have large, multipart conditional statements that depend on the object’s state.
- This state is usually represented by one or more enumerated constants.
- Often, several operations will contain this same conditional structure.
- The State pattern puts each branch of the conditional in a separate class

Example

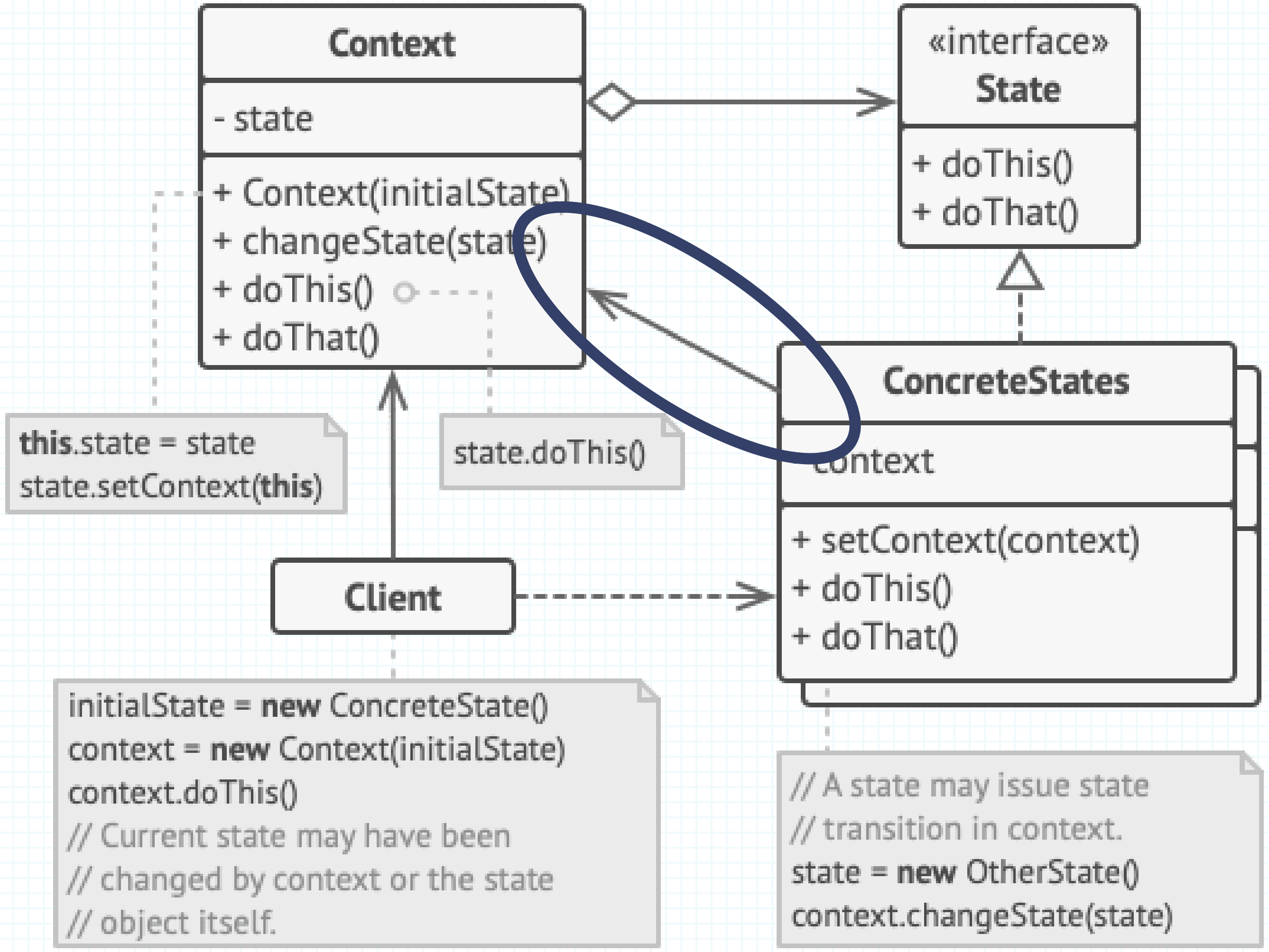

Connection
+connect()
+disconnect()
+receive() : int
+send(in data : int)
+createAutomaton(in socket : Socket)
«uses»
«interface»
connection::IConnection
+connect()
+disconnect()
+receive() : int
+send(in data : int)
ConnectedState
+DISCONNECT : Event
-socket : Socket
+connect()
+disconnect()
+receive() : int
+send(in data : int)
DisconnectedState
+CONNECT : Event
-socket : Socket
+connect()
+disconnect()
+receive() : int
+send(in data : int)
Socket
+connect()
+disconnect()
+receive() : int
+send(in data : int)

Example

- Imagine that we have a Document class. A document can be in one of three states: Draft, Moderation and Published. The publish method of the document works a little bit differently in each state:
- In Draft, it moves the document to moderation.
- In Moderation, it makes the document public, but only if the current user is an administrator.
- In Published, it doesn't do anything at all.



State Pattern



Comparison

- **Template** varies <u>specific</u> parts of the algorithm by subclasses.
- **Template Method** is based on <u>inheritance</u>

- **State** encapsulates state-specific behavior into separate objects
- **State pattern** is based on <u>composition.</u>

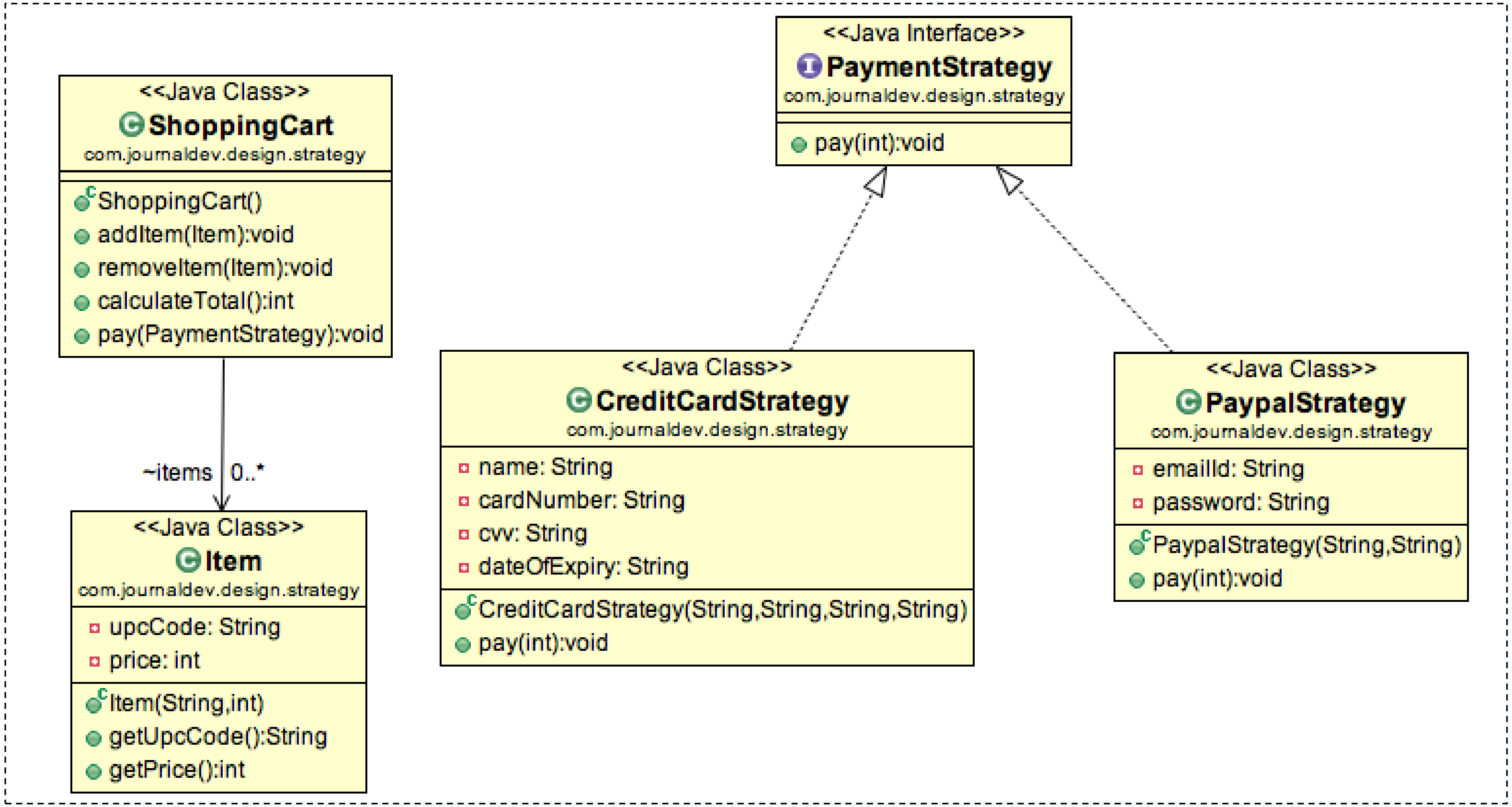
- **Strategy** varies the <u>entire</u> algorithm via its strategies.
- **Strategy** is based on <u>composition</u>

Examples

Example 1

- You are designing a system for processing online orders in an e-commerce platform. The system needs to handle various payment methods (e.g., credit card, PayPal, bank transfer) with specific validation and transaction processing logic. How would you design the payment processing system to accommodate different payment methods while ensuring flexibility and maintainability?

Solution



<<Java Interface>>
PaymentStrategy
com.journaldev.design.strategy
pay(int):void
<<Java Class>>
ShoppingCart
com.journaldev.design.strategy
ShoppingCart()
addItem(Item):void
removeItem(Item):void
calculateTotal():int
pay(PaymentStrategy):void
~items
0..*
<<Java Class>>
Item
com.journaldev.design.strategy
upcCode: String
price: int
Item(String,int)
getUpcCode():String
getPrice():int
<<Java Class>>
CreditCardStrategy
com.journaldev.design.strategy
name: String
cardNumber: String
cvv: String
dateOfExpiry: String
CreditCardStrategy(String,String,String,String)
pay(int):void
<<Java Class>>
PaypalStrategy
com.journaldev.design.strategy
emailId: String
password: String
PaypalStrategy(String,String)
pay(int):void

Detailed Solution

```
public interface PaymentStrategy {
	public void pay(int amount);
}
```

```
public class CreditCardStrategy implements PaymentStrategy {
	private String name;
	private String cardNumber;
	private String cvv;
	private String dateOfExpiry;

	public CreditCardStrategy(String nm, String ccNum, String cvv, String 
expiryDate){
		this.name=nm;
		this.cardNumber=ccNum;
		this.cvv=cvv;
		this.dateOfExpiry=expiryDate;
	}
	@Override
	public void pay(int amount) {
		System.out.println(amount +" paid with credit/debit card");
	}
}
```

Detailed Solution

```
public interface PaymentStrategy {
        public void pay(int amount);
}
```

```
public class PaypalStrategy implements PaymentStrategy {

        private String emailId;
        private String password;

        public PaypalStrategy(String email, String pwd){
                this.emailId=email;
                this.password=pwd;
        }

        @Override
        public void pay(int amount) {
                System.out.println(amount + " paid using Paypal.");
        }

}
```

Detailed Solution

```
public class Item {

	private String upcCode;
	private int price;

	public Item(String upc, int cost){
		this.upcCode=upc;
		this.price=cost;
	}
	public String getUpcCode() {
		return upcCode;
	}

	public int getPrice() {
		return price;
	}
}
```

```
public class ShoppingCart {
	List<Item> items;
	public ShoppingCart(){
		this.items=new ArrayList<Item>();
	}
	public void addItem(Item item){
		this.items.add(item);
	}
	public void removeItem(Item item){
		this.items.remove(item);
	}
	public int calculateTotal(){
		int sum = 0;
		for(Item item : items){
			sum += item.getPrice();
		}
		return sum;
	}
	public void pay(PaymentStrategy paymentMethod){
		int amount = calculateTotal();
		paymentMethod.pay(amount);
	}
```

Detailed Solution

```
public class ShoppingCartTest {
	public static void main(String[] args) {
		ShoppingCart cart = new ShoppingCart();

		Item item1 = new Item("1234",10);
		Item item2 = new Item("5678",40);

		cart.addItem(item1);
		cart.addItem(item2);

		//pay by paypal
		cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

		//pay by credit card
		cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/15"));
	}
}
```

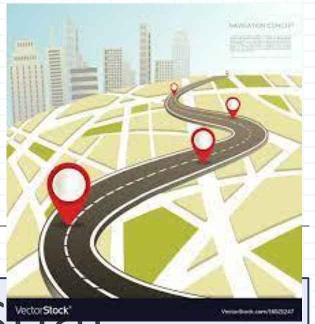
Example 2

- You are developing a video editing application where users can apply different filters and effects to their videos. Each filter or effect may have its own unique implementation, but they all follow a common sequence of steps for processing the video frames. How would you design the application to support the addition of new filters and effects while maintaining a consistent processing workflow?

Solution

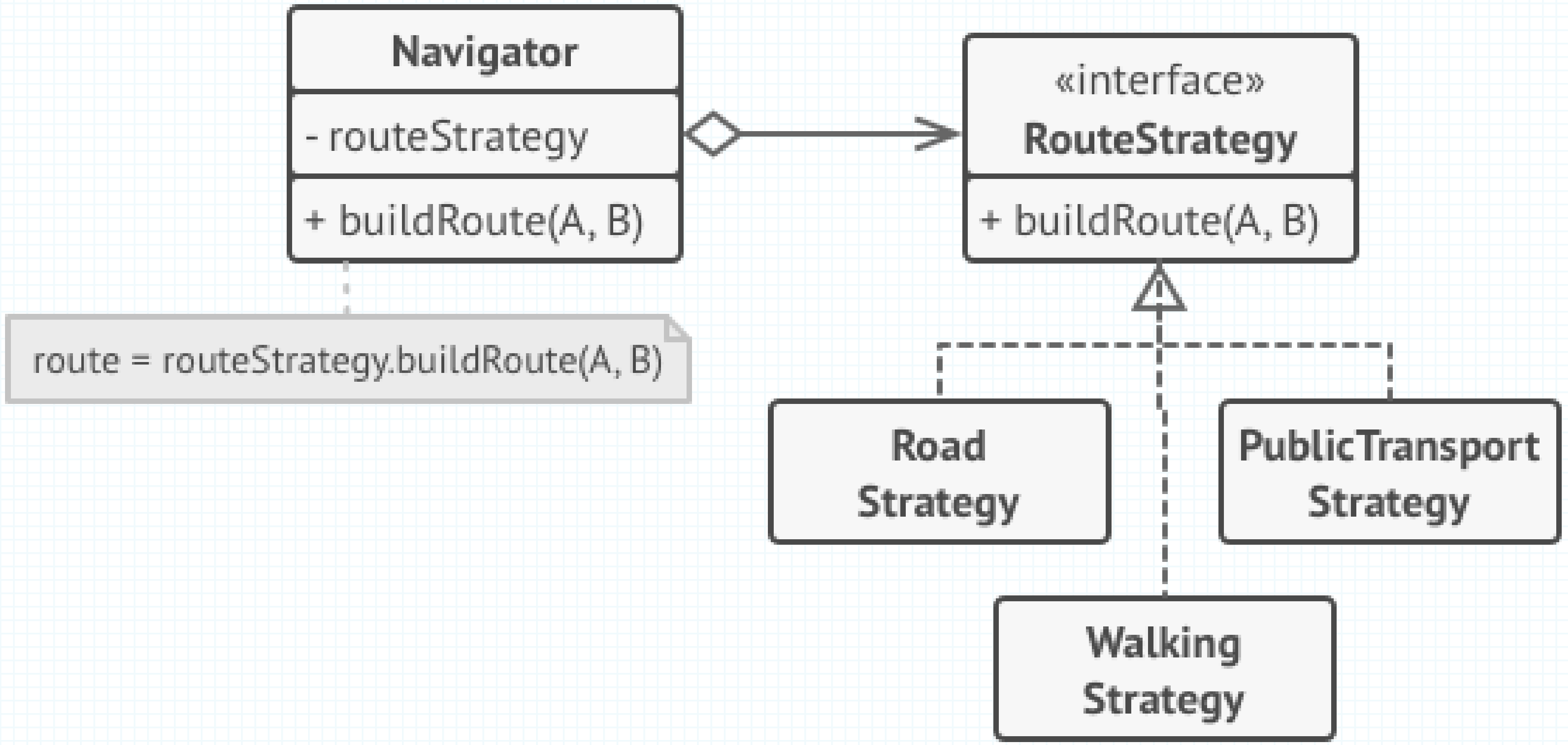
- You can define a template method in a base class that outlines the common sequence of steps for processing video frames, with specific methods that subclasses can override to implement individual filters or effects.
- This allows for the addition of new filters and effects while ensuring consistency in the processing workflow across different implementations.

Example 3



- One day you decided to create a navigation app for casual travelers. The app was centered around a beautiful map which helped users quickly orient themselves in any city.
- One of the most requested features for the app was automatic route planning. A user should be able to enter an address and see the fastest route to that destination displayed on the map.
- The app could build the routes over roads, build walking routes.
- Right after that, you added another option to let people use public transport in their routes.
- And even later, another option for building routes through all of a city’s tourist attractions.   etc

Solution


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

Example 4

- You are designing a game where characters can have different behaviors based on their current state (e.g., idle, attacking, defending). The behavior of each character may change dynamically based on game events and player actions. How would you design the game system to handle character behaviors while ensuring flexibility and extensibility?

Solution

- The State pattern would be an appropriate choice for designing the character behavior system. Each behavior of the character (e.g., idle, attacking, defending) could be implemented as a separate state object, encapsulating its specific logic. The system can transition between states dynamically based on game events and player actions, allowing for flexibility and extensibility in managing character behaviors.