الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

مقرر تصميم نظم البرمجيات | كلية الهندسة المعلوماتية | المحاضرة الثالثة

Design Patterns 1:
Strategy Pattern, Template Method Pattern

د. رياض سنبل

Design Patterns

- Designing object-oriented software is hard, and designing reusable object-oriented software is even harder.
- You also want to avoid redesign, or at least minimize it
- **design pattern**:
  A standard **solution** to a common software **problem** in a context.
  - describes a recurring software structure or idiom
  - is abstract from any particular programming language
  - identifies classes and their roles in the solution to a problem

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
  - Template Method

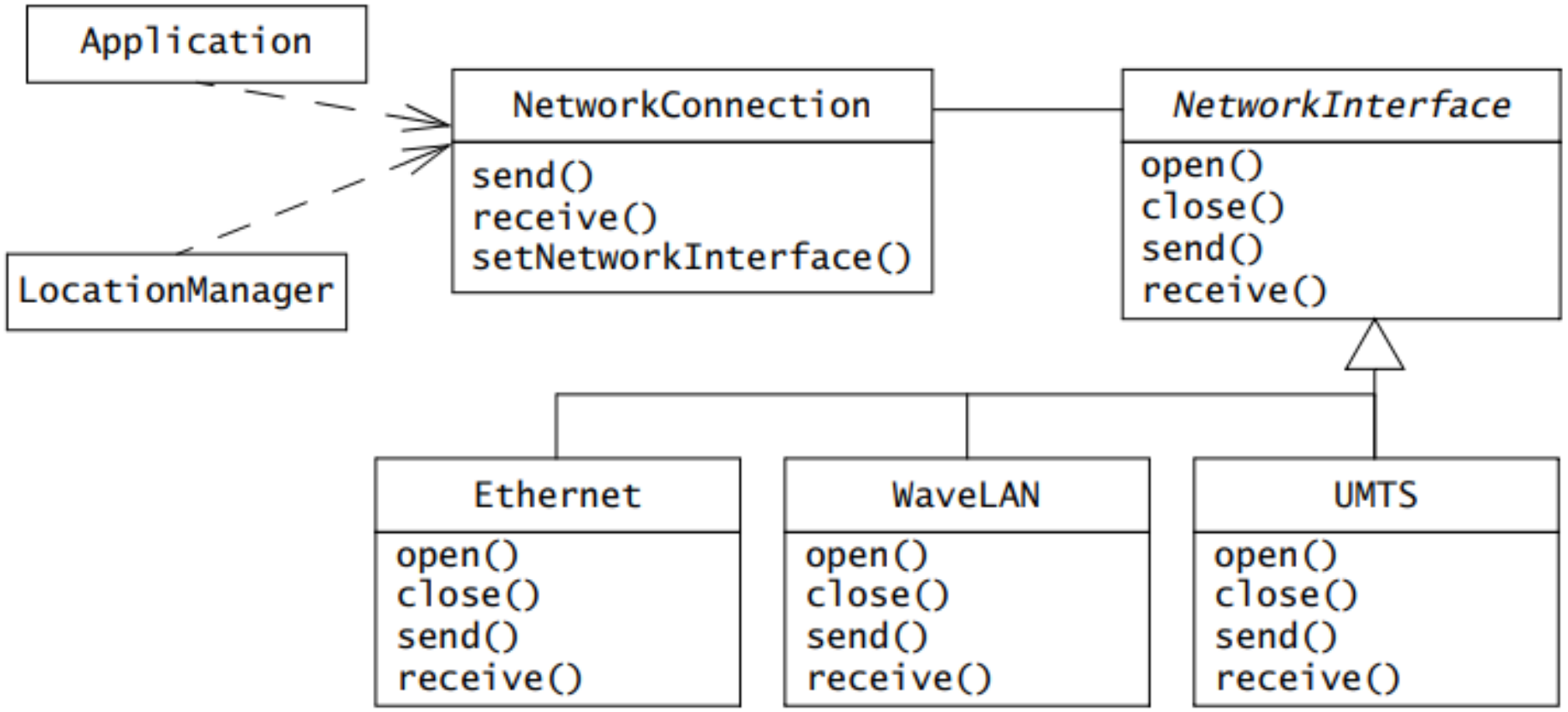
Strategy (or Policy) Pattern

Example

- Consider a mobile application running on a wearable computer that uses different networks :
  - *A car mechanic using the wearable computer to access repair manuals and maintenance records for the vehicle under repair.*
  - *The wearable computer should operate in the shop with access to a local wireless network as well as on the roadside using a third-generation mobile phone network, such as UMTS.*
  - *When updating or configuring the mobile application, a system administrator should be able to use the wearable computer with access to a wired network such as Ethernet.*

- This means that the mobile application needs to deal with different types of networks as it **switches between networks dynamically**, based on factors such as location and network costs.
- Assume that during the system design of this application, we identify the dynamic switching between wired and wireless networks as a critical design goal.
- Furthermore we want to be able to deal with future network protocols without having to recompile the application

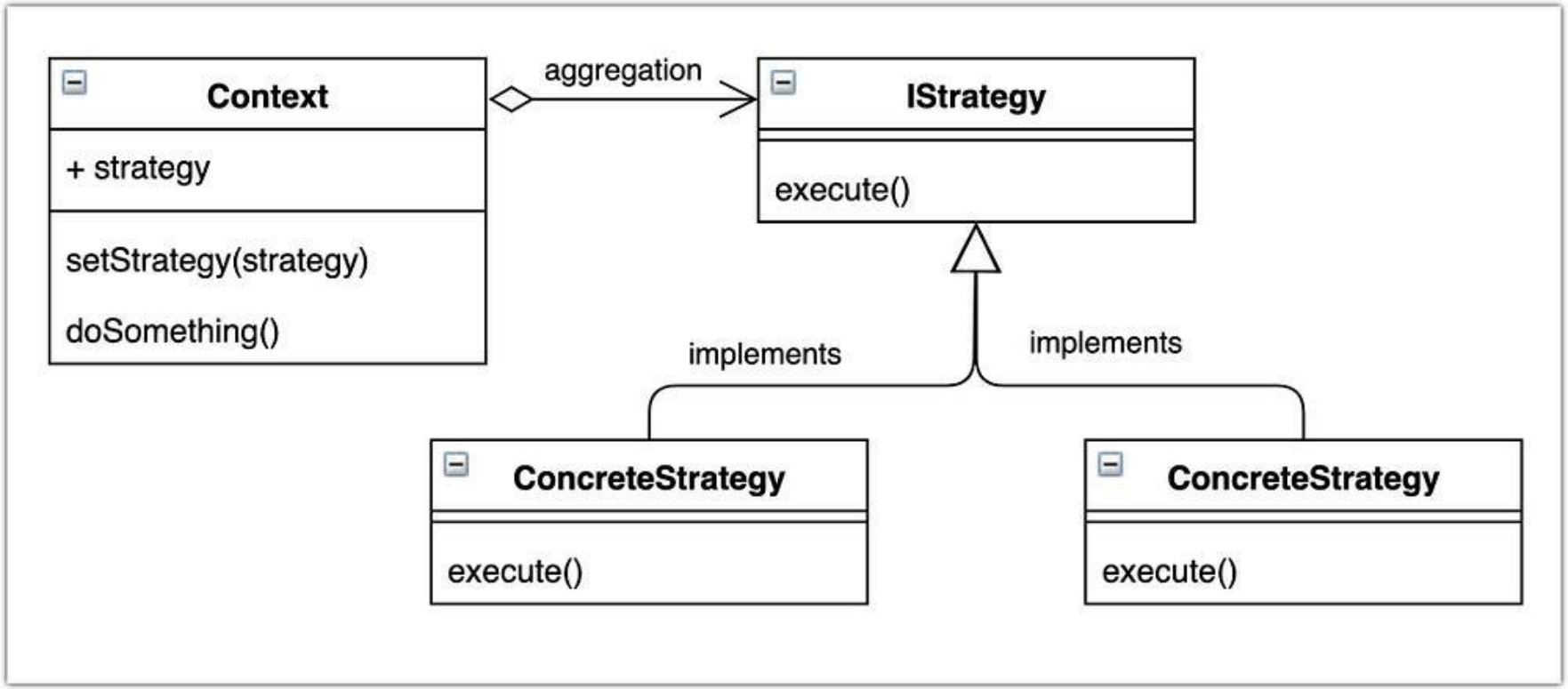
Final Solution



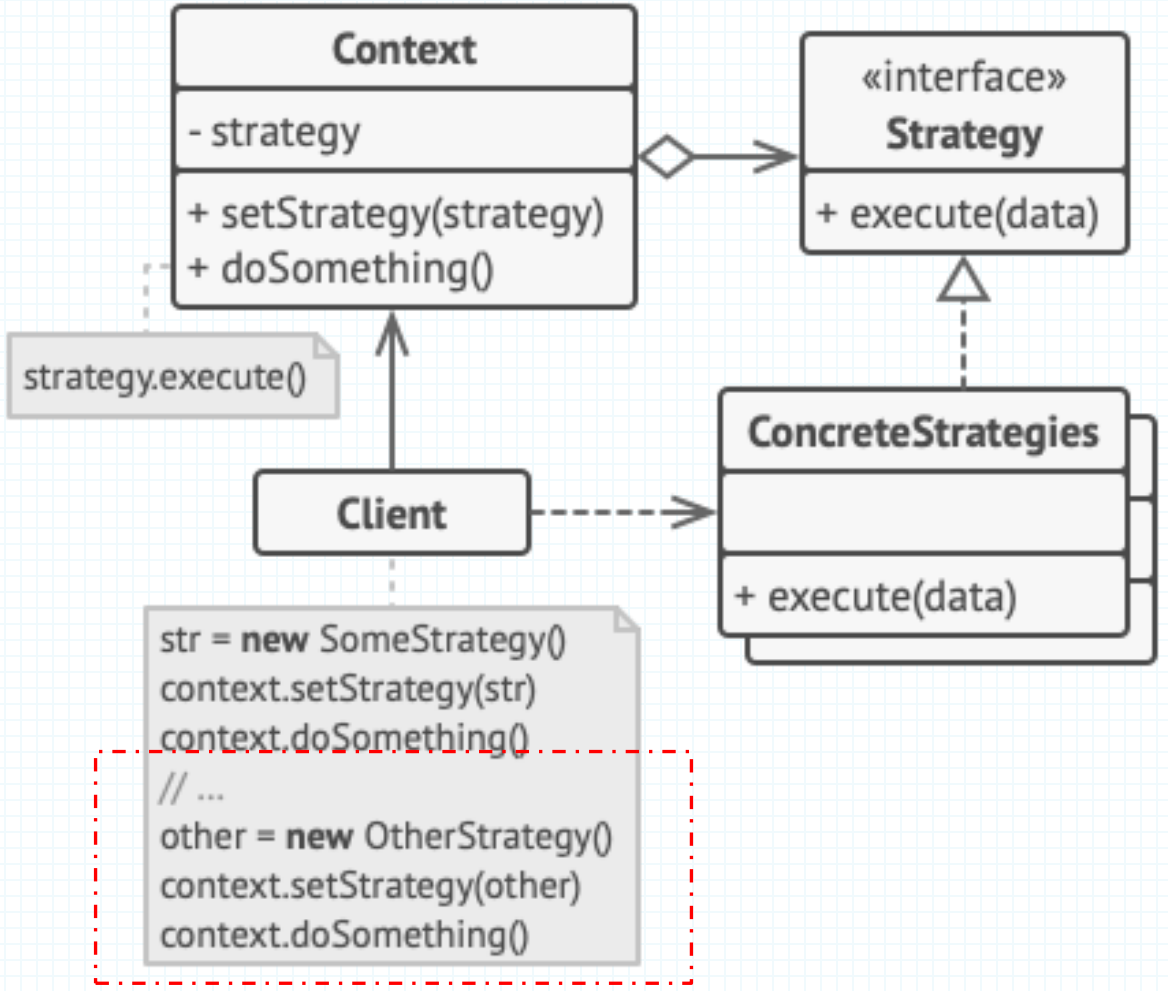
Strategy Pattern

- The intent of the Strategy Pattern is to:
  - (1) Define a family of algorithms, policies, logics, etc
  - (2) Encapsulate each one,
  - (3) Make them interchangeable within that family
- Strategy Pattern lets the algorithm **vary independently from the clients** that use it
- The Strategy Pattern enables an algorithm's behavior to be **selected at runtime**

The structure

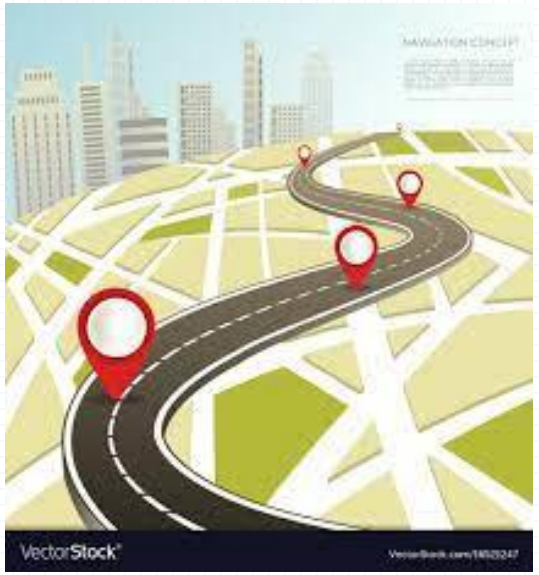


The structure

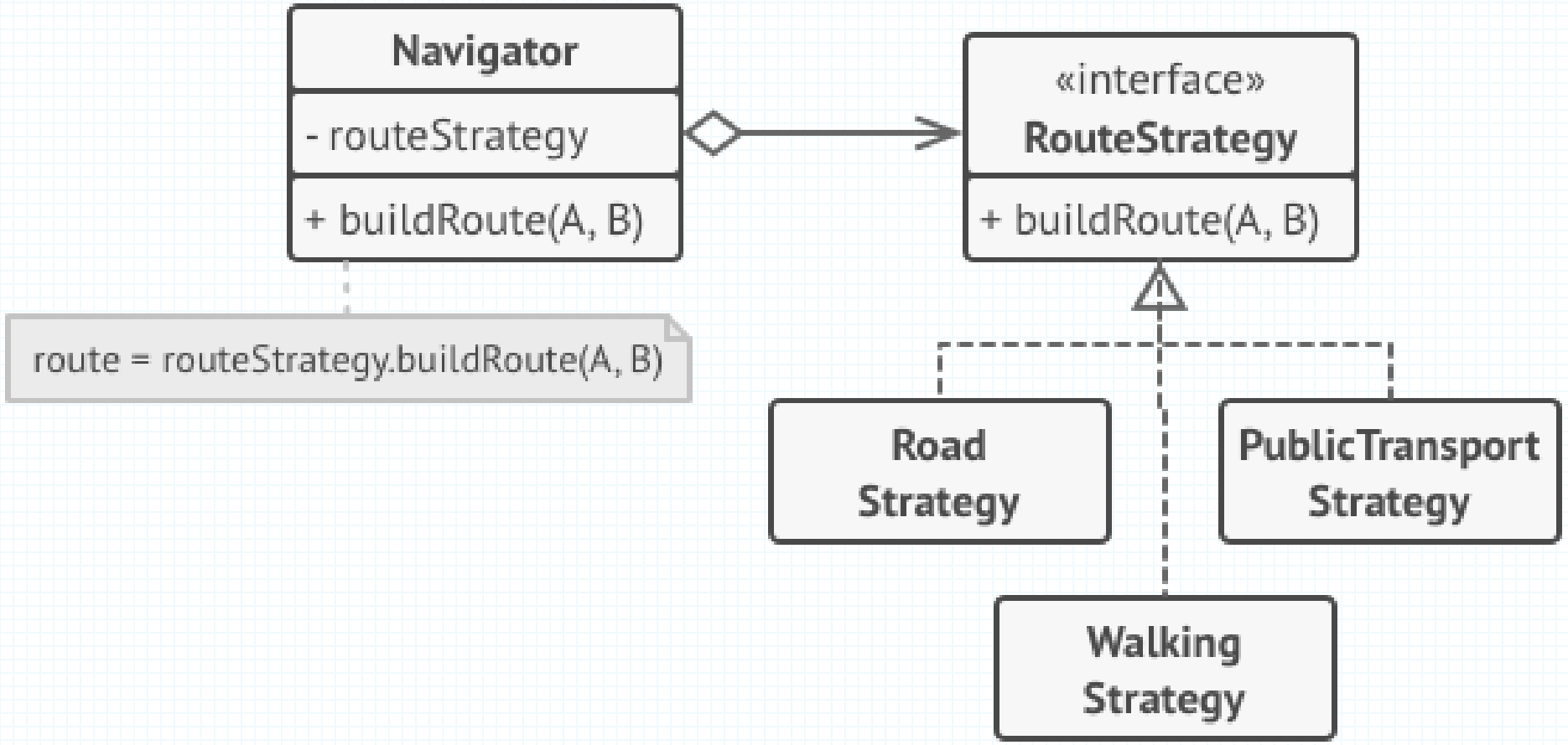


Practical Example: Navigation App

- 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 first version of the app could only build the routes over roads.
- The next update, you added an option to 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



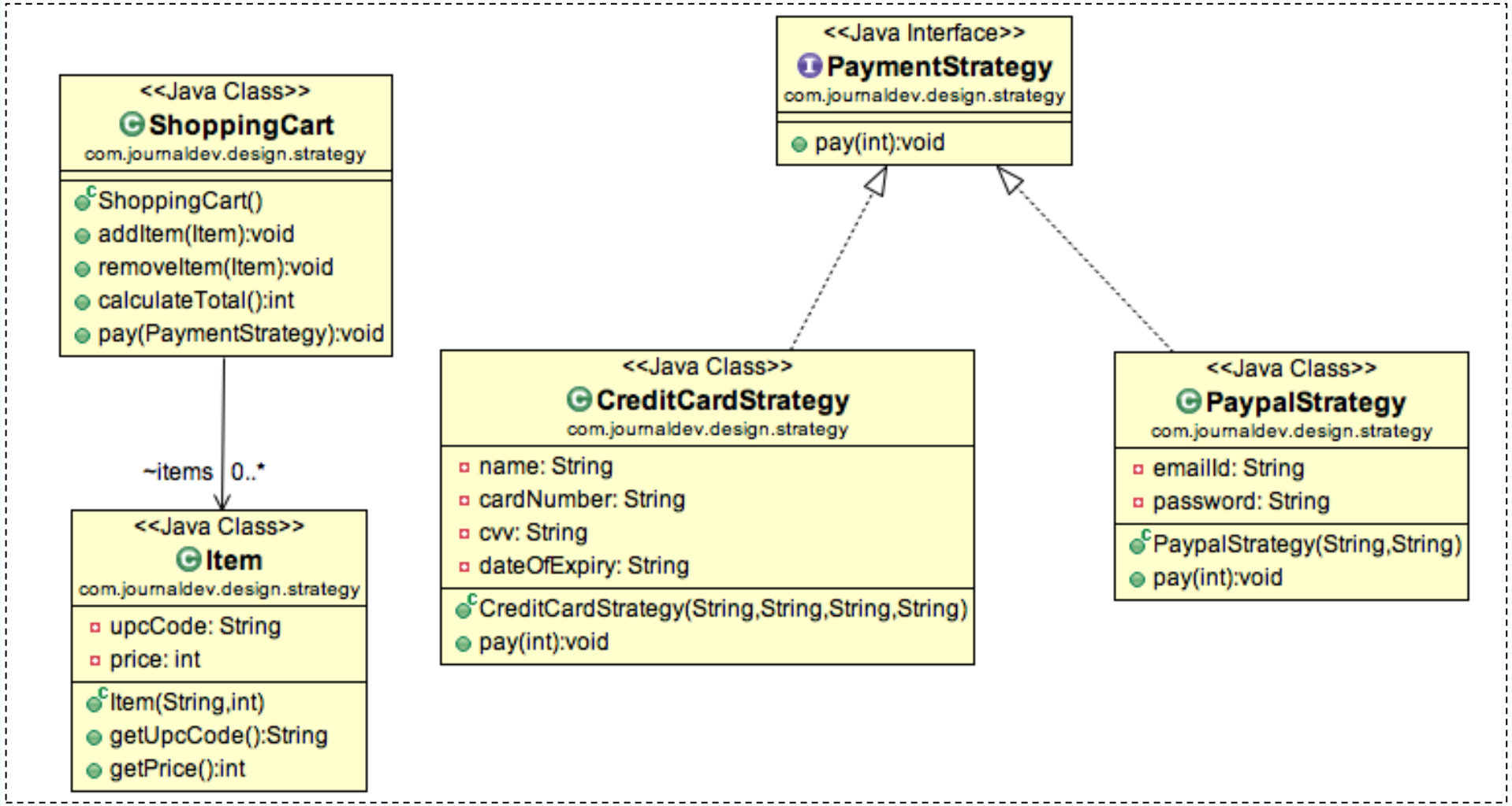
Solution


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

Example: E-Commerce

- Implement a simple Shopping Cart where we have two payment strategies - using Credit Card or using PayPal.

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;
	}
}
```

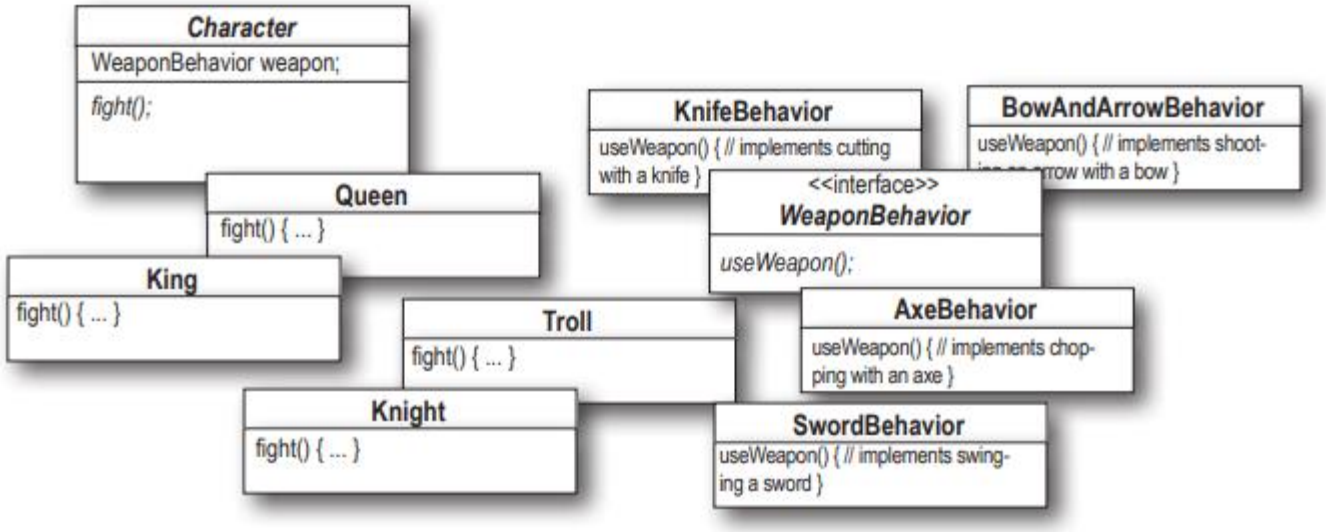
```
public class ShoppingCart {
	List<Item> items;
	public ShoppingCart(){
		this.items=new ArrayList<Item>();
	}
	public void addItem(Item item){
		this.items.add(item);
	}
	public void removeItem(Item item){
		this.items.remove(item);
	}
	public int calculateTotal(){
		int sum = 0;
		for(Item item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

Detailed Solution

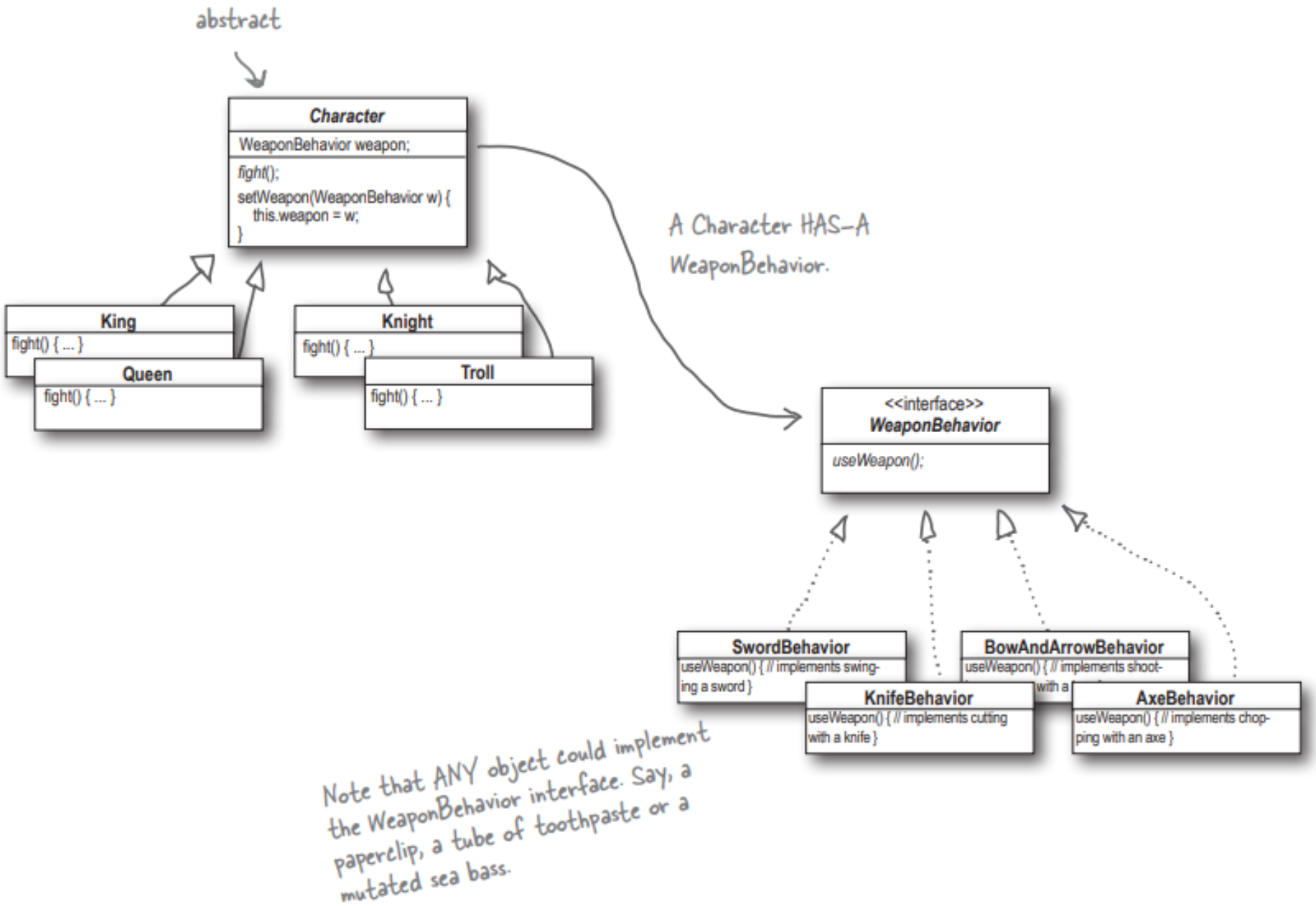
```
public class ShoppingCartTest {
	public static void main(String[] args) {
		ShoppingCart cart = new ShoppingCart();

		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: Adventure Game

- Below you'll find a mess of classes and interfaces for an action adventure game. You'll find classes for game characters along with classes for weapon behaviors the characters can use in the game. Each character can make use of one weapon at a time, but can change weapons at any time during the game. Your job is to sort it all out...



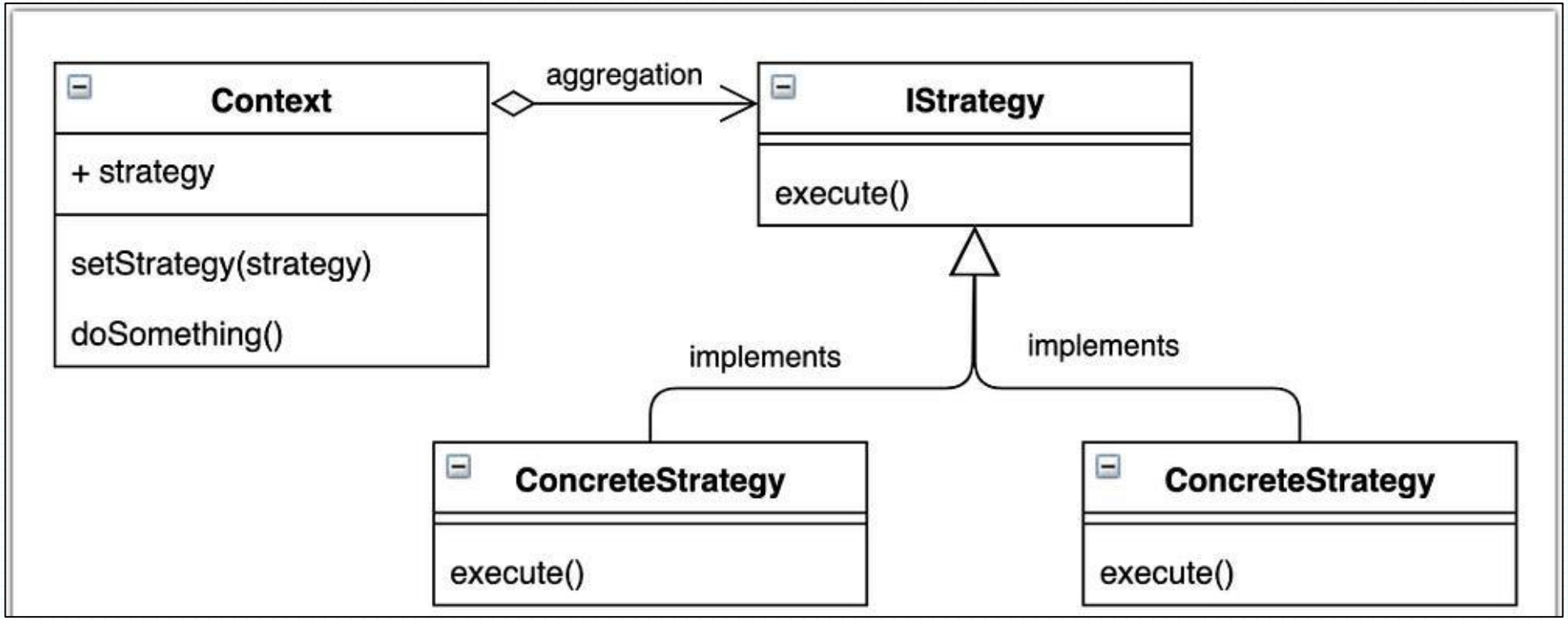
Solution


abstract
Character
WeaponBehavior weapon;
fight();
setWeapon(WeaponBehavior w) {
this.weapon = w;
}
A Character HAS-A WeaponBehavior.
King
fight() { ... }
Queen
fight() { ... }
Knight
fight() { ... }
Troll
fight() { ... }
<<interface>>
WeaponBehavior
useWeapon();
SwordBehavior
useWeapon() { // implements swinging a sword }
BowAndArrowBehavior
useWeapon() { // implements shooting with a
KnifeBehavior
useWeapon() { // implements cutting with a knife }
AxeBehavior
useWeapon() { // implements chopping with an axe }
Note that ANY object could implement the WeaponBehavior interface. Say, a paperclip, a tube of toothpaste or a mutated sea bass.

Template Method Pattern

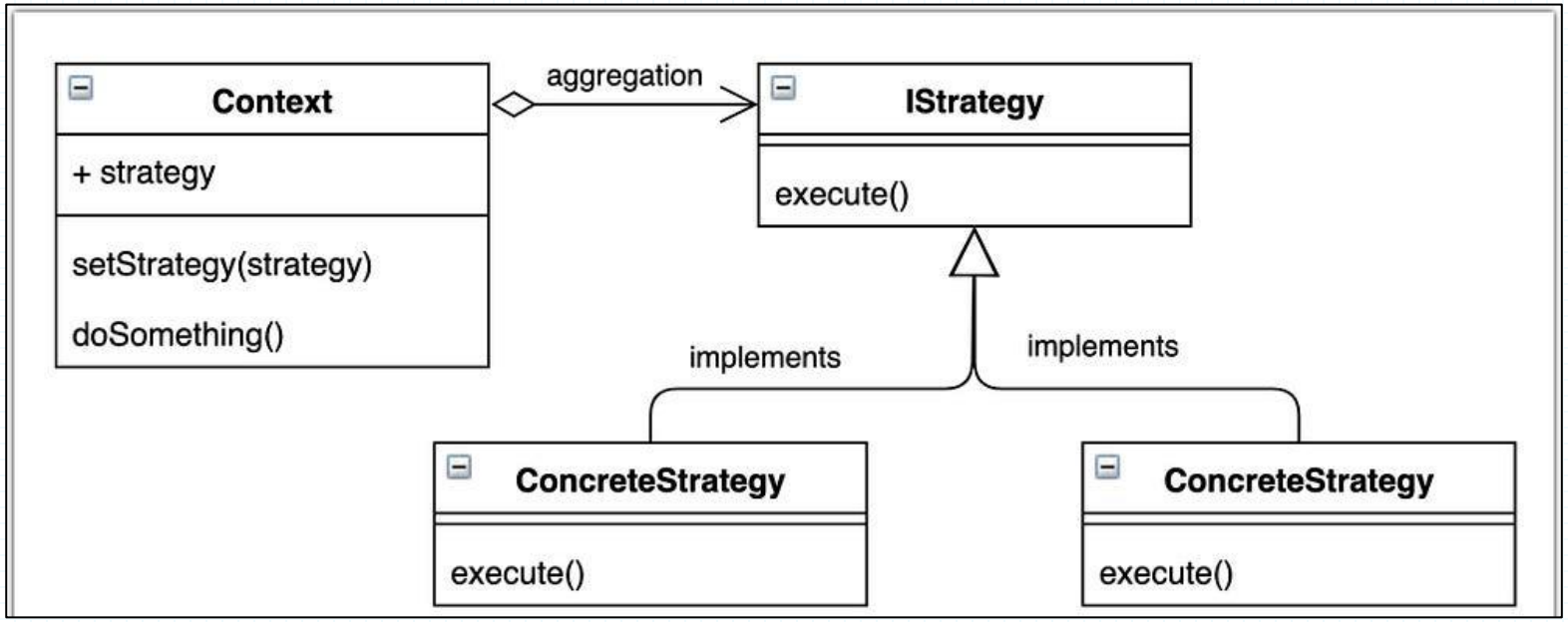
TEMPLATE METHOD PATTERN

Strategy Pattern



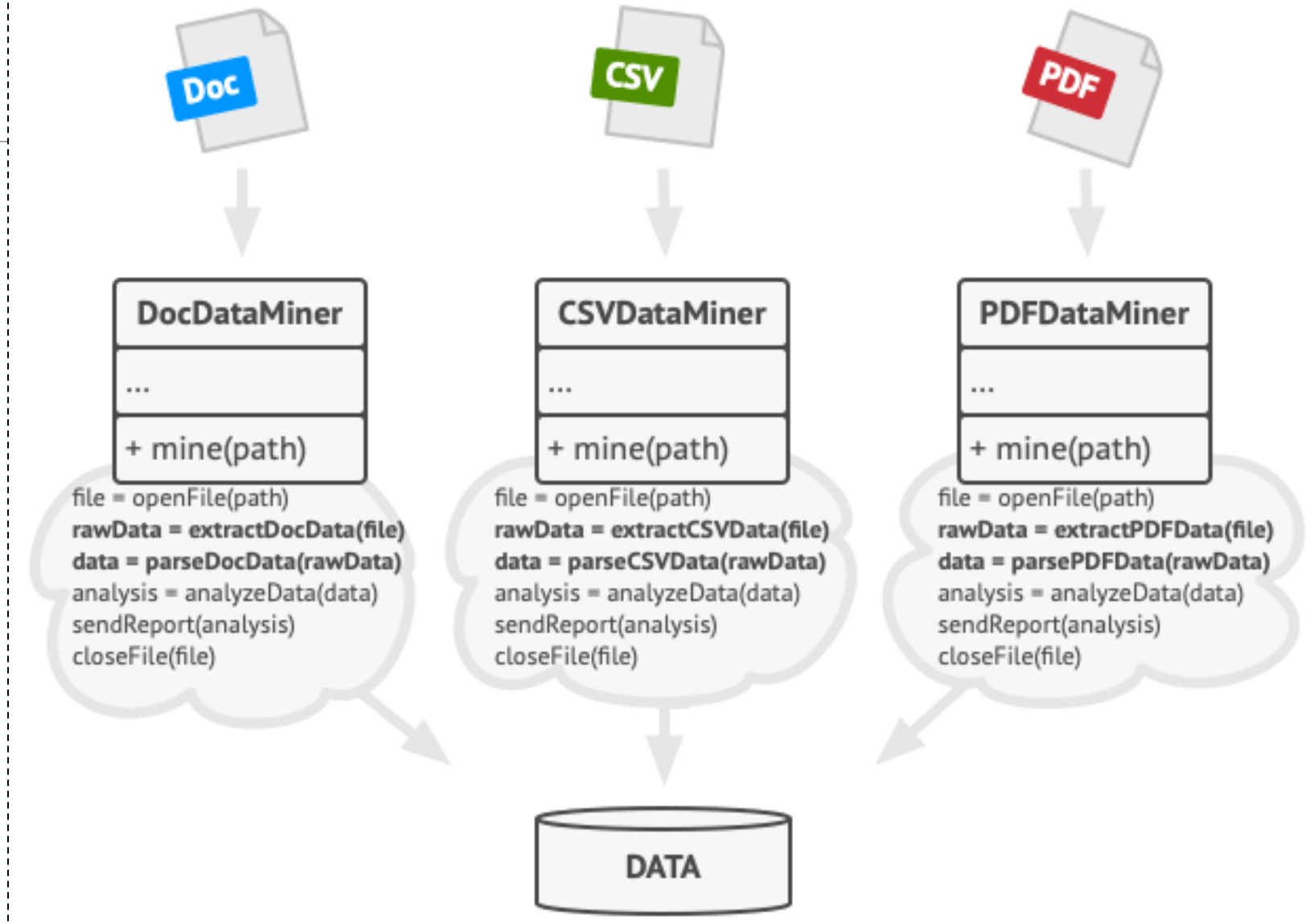
Strategy Pattern.. What if?

- 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.