Advanced Programming Methods

Iuliana Bocicor
maria.bocicor@ubbcluj.ro

Babes-Bolyai University

2024

Overview

References

The material in this lecture was inspired from:

- **Design Patterns: Elements of Reusable Object-Oriented Software**, by *Erich Gamma, Richard Helm, Ralph Johnson* and *John Vlissides*
- Refactoring guru: Design Patterns.
- Source making: Design Patterns
- Freeman, E., Robson, E., Bates, B., and Sierra, K. (2008). Head first design patterns. O'Reilly Media, Inc.

Design Patterns I

- When designing something new (a building, a novel, a computer program), designers make certain decisions.
- Experienced designers (architects, writers, software architects) know **not** to solve a problem from first principles, but to reuse good solutions that have worked in the past.
- Patterns are like templates that can be applied in many different situations.
- Software design patterns are recurring descriptions of classes and communicating objects that are customised to solve a general design problem in a particular context.

Design Patterns II

- They are general, flexible, reusable solutions to commonly occurring problems within a given context in software design.
- Object-oriented design patterns show relationships and interactions between classes or objects.
- Christopher Alexander, *" A Pattern Language"*: *"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"*.

Design Patterns III

- One of the most influential books for software engineering and object-oriented design theory and practice: **Design Patterns: Elements of Reusable Object-Oriented Software**, by *Erich Gamma, Richard Helm, Ralph Johnson* and *John Vlissides*.
- They are often referred to as the **Gang of Four (GoF)**.
- This book introduces the principles of design patterns and then offers a catalogue of such patterns (23 classic design patterns).

Essential elements of a pattern I

- **Pattern name**
  - a word or two which describe the desing problem, its solution and consequences;
  - it is a part of the software developer vocabulary;
  - "Finding good names has been one of the hardest parts of developing our catalog." (GoF)
- **Problem**
  - describes when to apply the pattern;
  - explains the problem and its context;
  - it might include a list of conditions that must be met before it makes sense to apply the pattern.

Essential elements of a pattern II

- **Solution**
  - describes the elements that make up the design, their relationships, responsibilities and collaborations;
  - provides an abstract description of a design problem and how the general arrangement of elements (classes and objects) solves it.
- **Consequences**
  - describe the results and trade-offs (space and time trade-offs) of applying the pattern;
  - may address language and implementation issues as well;
  - they include the pattern's impact on a system's flexibility, extensibility, or portability.

Patterns' purposes I

- The purpose of a pattern reflects what a pattern does.
- **Creational patterns**
  - concern the process of object creation;
  - E.g.: Abstract Factory, Builder, Factory Method, Prototype, Singleton.
- **Structural patterns**
  - are concerned with how classes are composed to form larger structures;
  - E.g.: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy.

Patterns' purposes II

- **Behavioural patterns**
  - are concerned with algorithms and the assignment of responsibilities between objects;
  - E.g.: Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.
- Some patterns are often used together (e.g. Composite is often used with Iterator).
- Some patterns are alternatives (e.g. Prototype is often an alternative to Abstract Factory).

Creational patterns

- Creational patterns deal with object creation mechanisms and abstract the instantiating process.
- The goal is to increase system flexibility and make the system independent of how its objects are created, composed, and represented.
- These patterns encapsulate knowledge about which concrete classes the system uses.
- They hide how instances of these classes are created.

Factory Method I

Problem

- Creation of a GUI application (first version is only for Windows).
- Classes for Windows controls are created and managed.
- What happens when a user requires the application to work cross-platform?
- We want to avoid situations in which we have many conditionals that switch the application's behaviour based on the platform it is running on.

Solution

The **Factory Method** design pattern.

Factory Method II

Intent

An interface is defined for creating an object, but subclasses will decide which class to instantiate.

Also known as

Virtual constructor

- Objects can be created without having to specify the exact class of the object that will be created.
- A *factory method* is used to create objects instead of directly using constructors.

Factory Method III

Structure

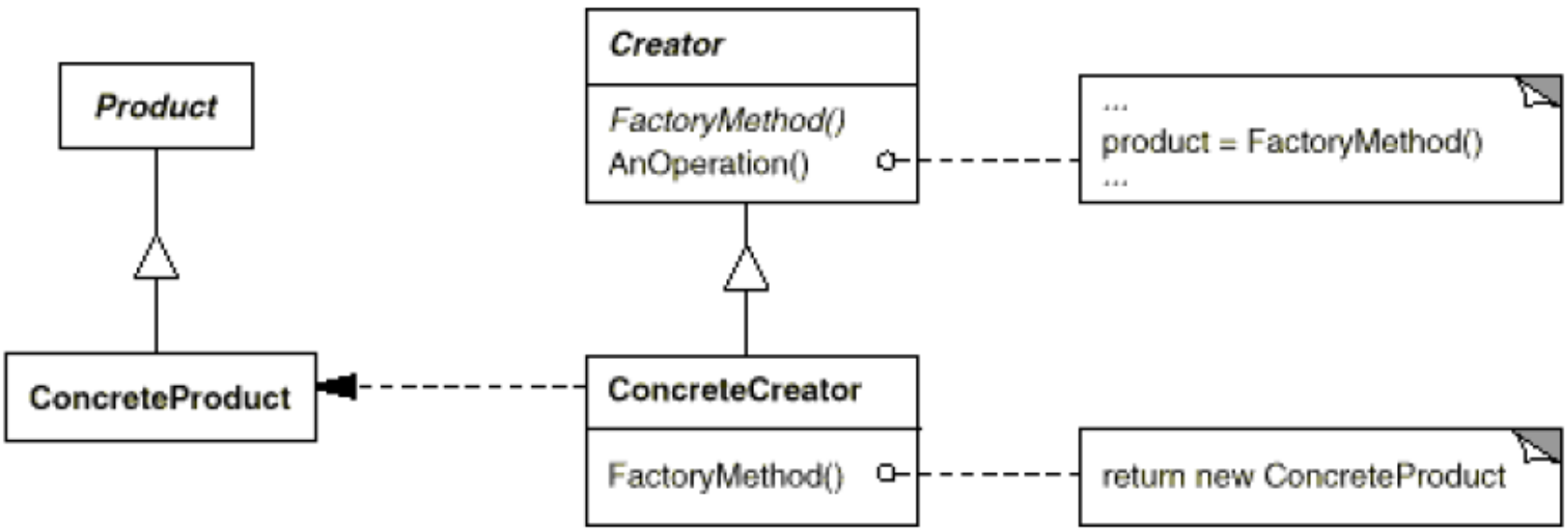


Figure: Figure source: GoF book

Factory Method IV

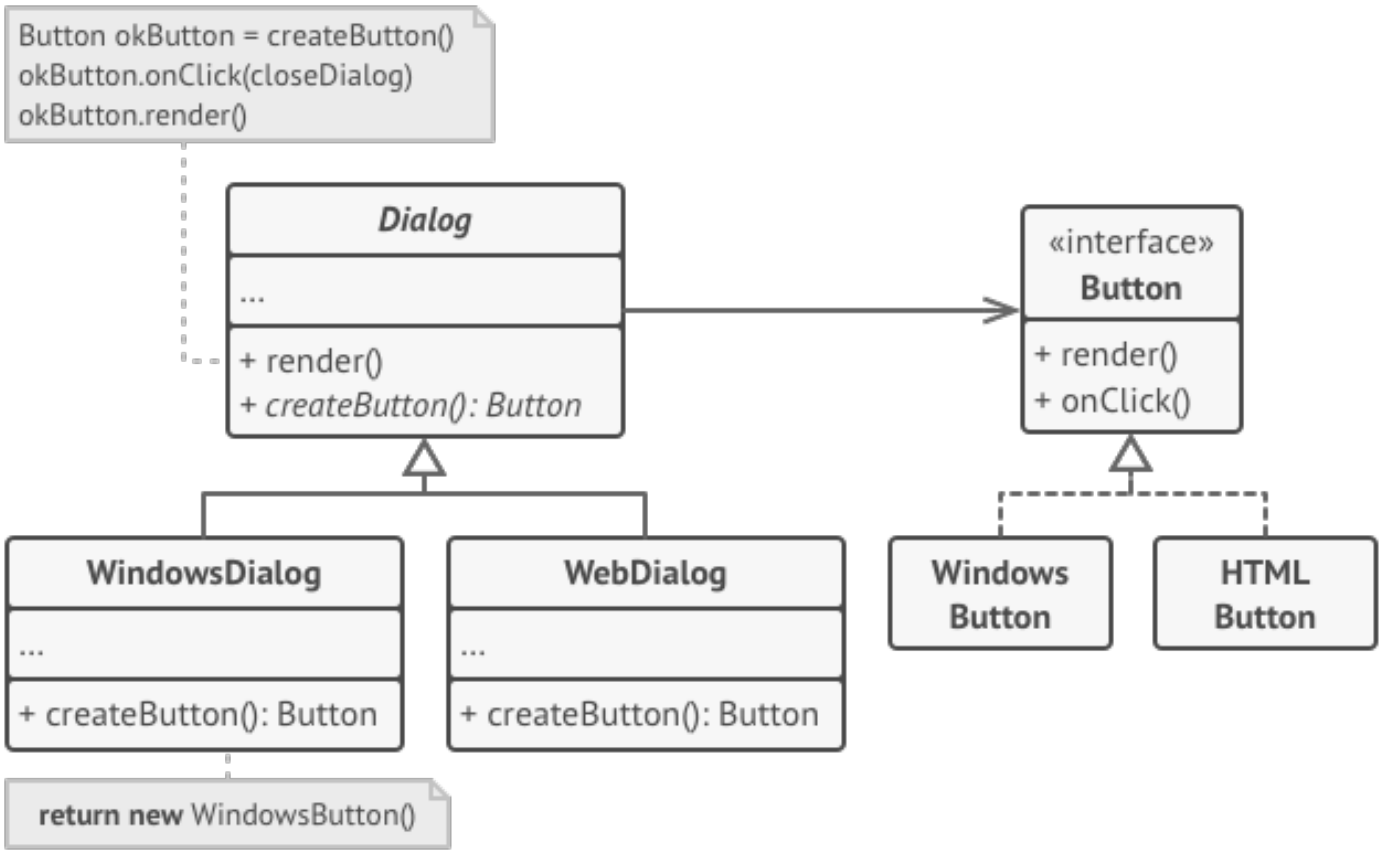


Figure: Figure source: Refactoring guru: Factory Method

Factory Method V

Factory method

FactoryMethod.Main

Factory Method VI

Applicability

Use the *Factory Method* when:

- a class cannot anticipate the class of objects it must create;
- we don't know beforehand the exact types and dependencies of the objects your code should work with;
- we want to allow the framework we create to easily extend its internal components.

Factory Method VII

Consequences

- The code only deals with the *Product* interface and therefore it can work with any *ConcreteProduct* class.
- Tight coupling between the creator and the concrete products is avoided.
- *Open/Closed Principle*: new types of products can be introduced into the program without breaking existing client code.
- *Single Responsibility Principle*: all product creation code can be moved into one place in the program, making the code easier to maintain.
- Introducing new subclasses can also make the code more complicated.

Abstract Factory I

Problem

- Furniture shop: provides several types of furniture items and different styles.
- We should be able to create objects such that they match the style.

Abstract Factory II

Figure: Figure source: Refactoring guru: Abstract Factory

Abstract Factory III

Solution

The **Abstract Factory** design pattern.

Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also known as

Kit

Abstract Factory IV

- The idea is to create an interface for each piece of furniture (Chair, Sofa) and then the concrete classes will all implement the interface.
- Further, we create an **abstract factory**, which can create all sorts of pieces of furniture (Chair, Sofa) and we inherit a concrete factory for each style.
- In this way, each factory will only create furniture of that style.

Abstract Factory V

Structure

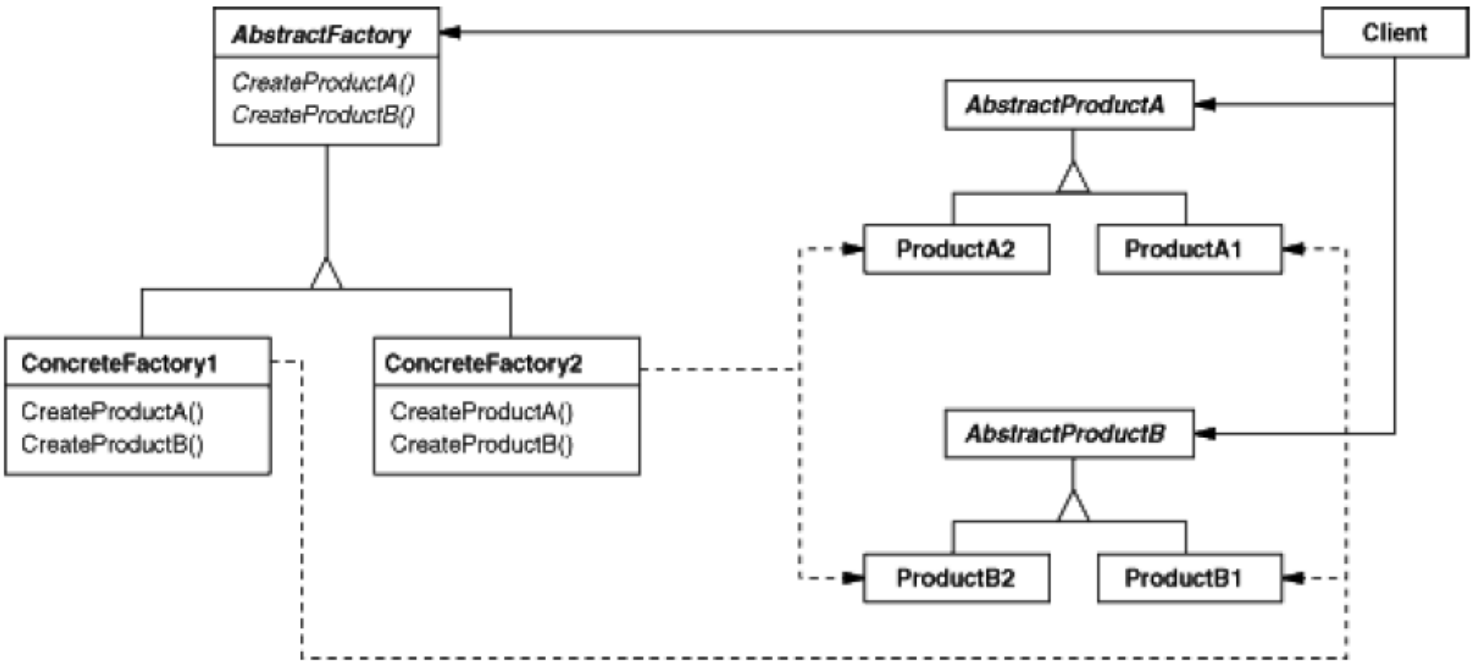


Figure: Figure source: GoF book

Abstract Factory VI

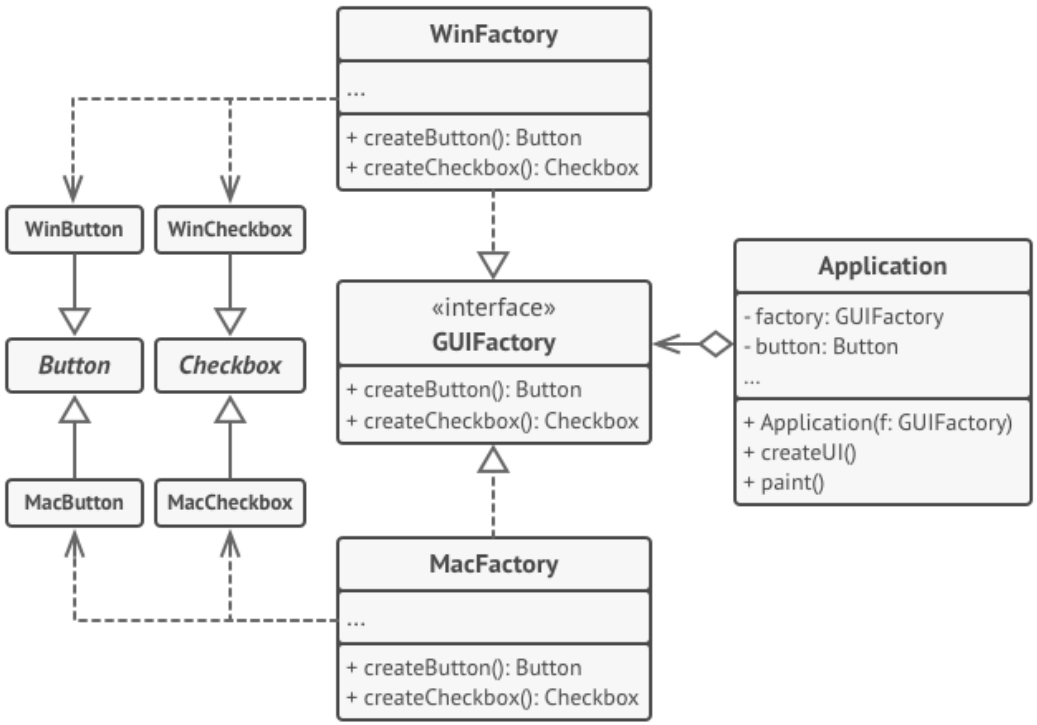


Figure: Figure source: Refactoring guru

Abstract Factory VII

Abstract Factory

AbstractFactory.Main

Abstract Factory VIII

Applicability

Use the *Abstract Factory* when:

- a system should be independent of how its products are created, composed, and represented;
- a system should be configured with one of multiple families of products;
- a family of related product objects is designed to be used together, and we need to enforce this constraint.

Abstract Factory IX

Consequences

- All products that we get from a factory are compatible with each other.
- Tight coupling between the concrete products and the client code is avoided.
- *Open/Closed Principle*: new variants of products can be introduced into the program without breaking existing client code.
- *Single Responsibility Principle*: all product creation code can be moved into one place in the program, making the code easier to support.
- Introducing new subclasses can also make the code more complicated.

Structural patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- These structures are being maintained flexible and efficient.
- They describe ways to compose objects to realise new functionality.
- The added flexibility of object composition comes from the ability to change the composition at run-time.

Adapter I

Problem

- You have an application that uses PayPal as payment service.
- As the application advances, you need to integrate other payment services, with completely different interfaces (payment functionality).

Solution

The **Adapter** design pattern.

- We create an adapter that converts the interfaces of the new services to something our program knows how to work with.

Adapter II

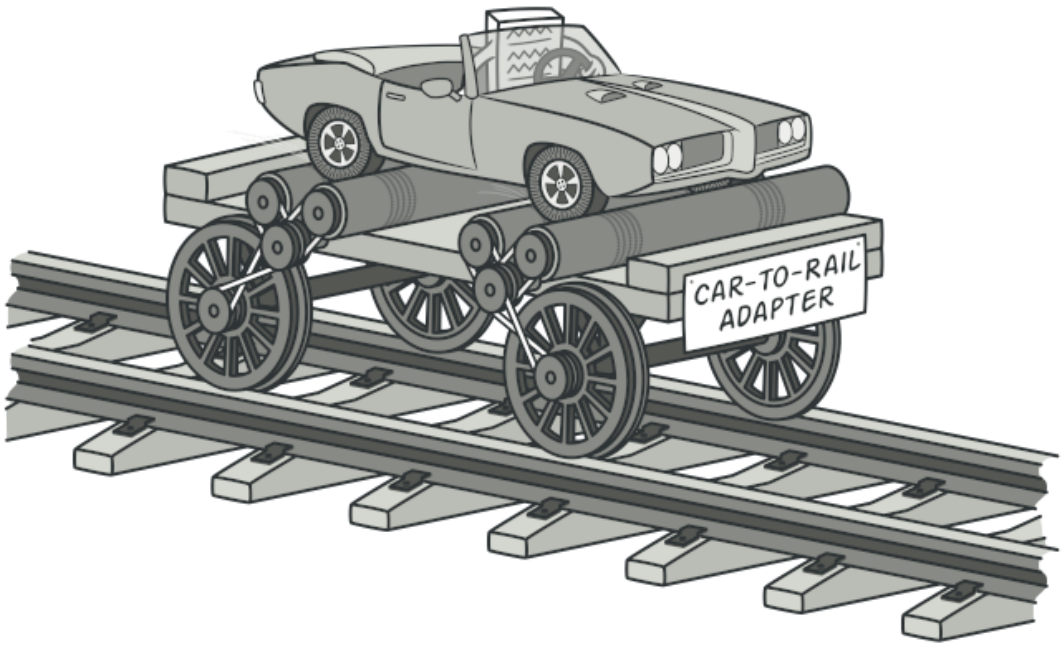


Figure: Figure source: Refactoring guru: Adapter

Adapter III

Intent

- Convert the interface of a class to another interface that a client expects.
- Is used to allow classes that could not communicate because of incompatible interfaces to work together.

Also Known As

- Wrapper.

Adapter IV

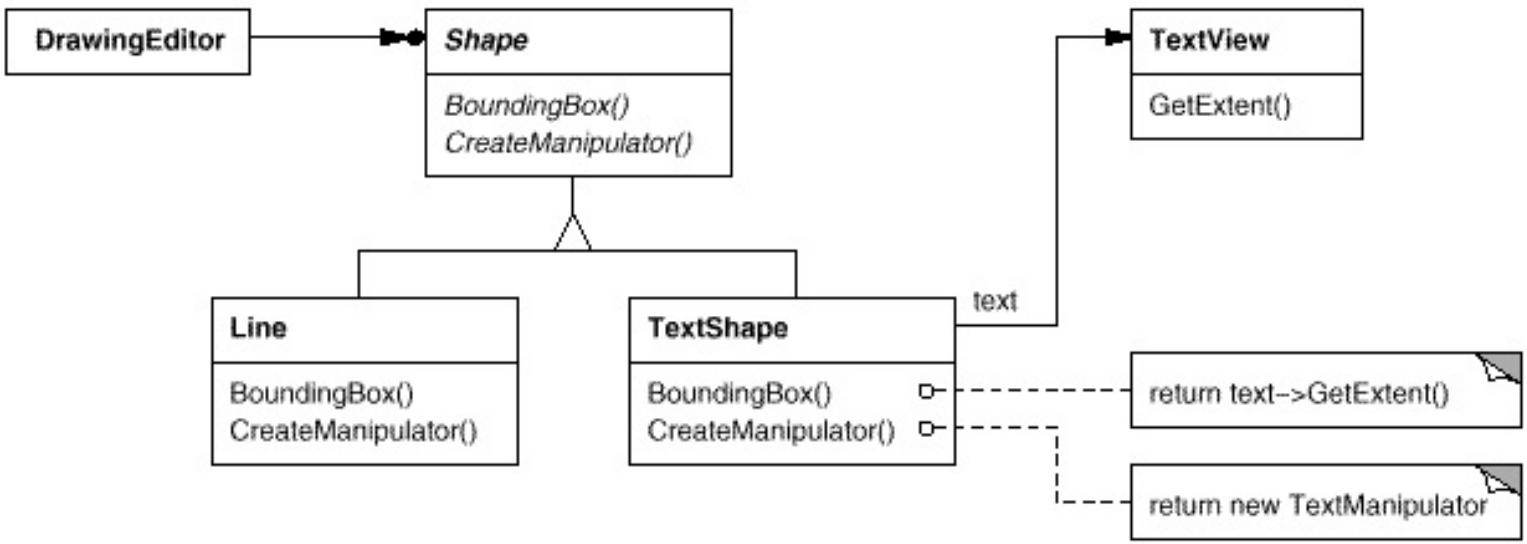


Figure: Figure source: GoF book

Adapter V

Applicability

Use *Adapter* when:

- an existing class could be used, but its interface does not match the one we need.
- we want to create a reusable class that cooperates with unrelated classes (classes that don't necessarily have compatible interfaces).

Adapter VI

Structure

A class adapter uses multiple inheritance to adapt one interface to another.

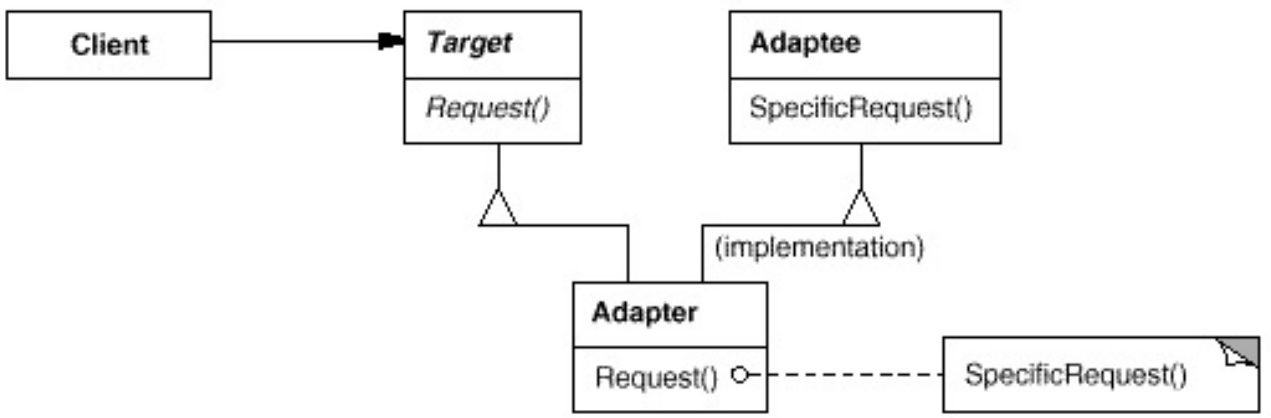


Figure: Figure source: GoF book

Adapter VII

An object adapter relies on object composition.

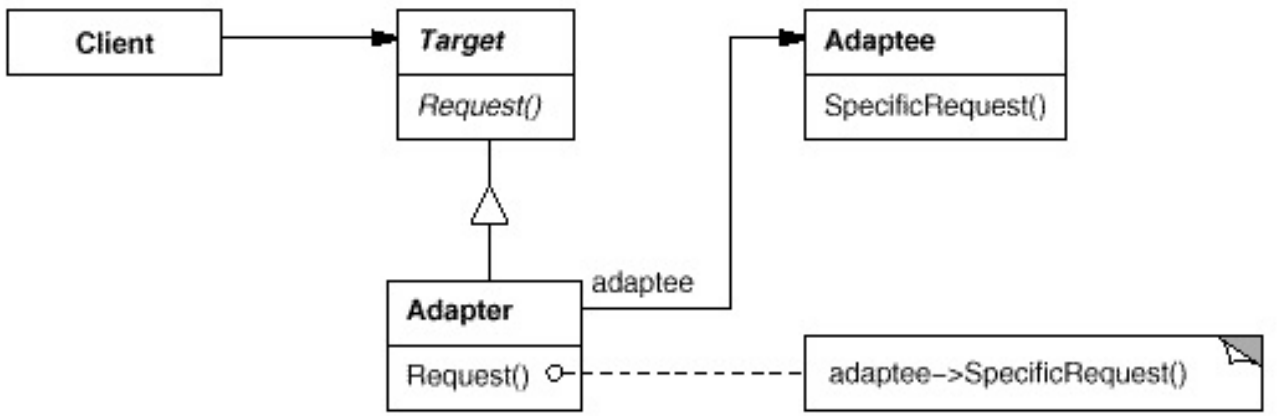
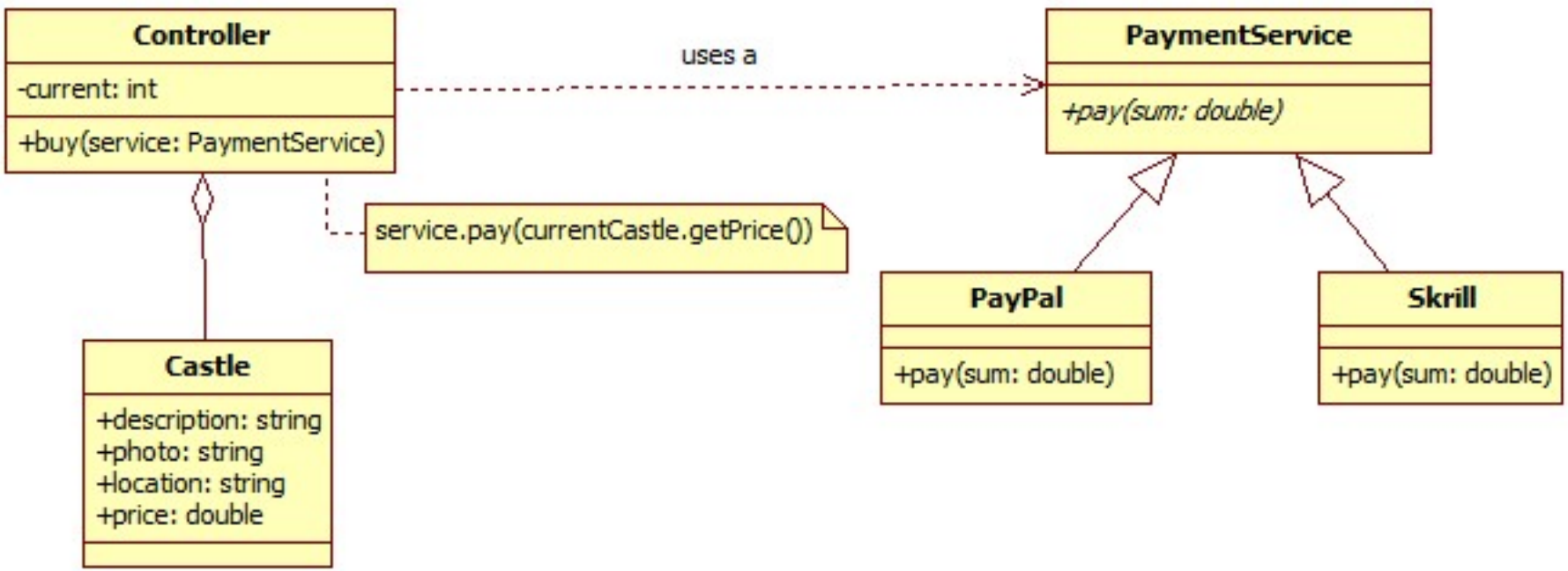
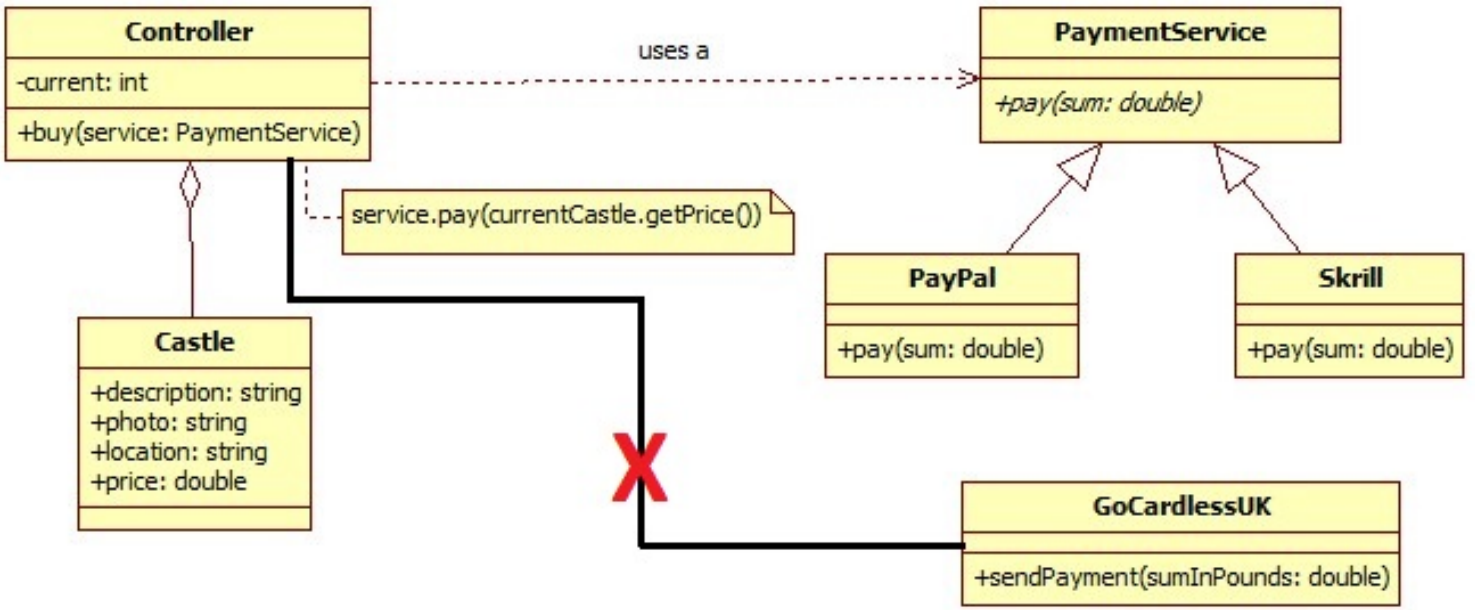


Figure: Figure source: GoF book

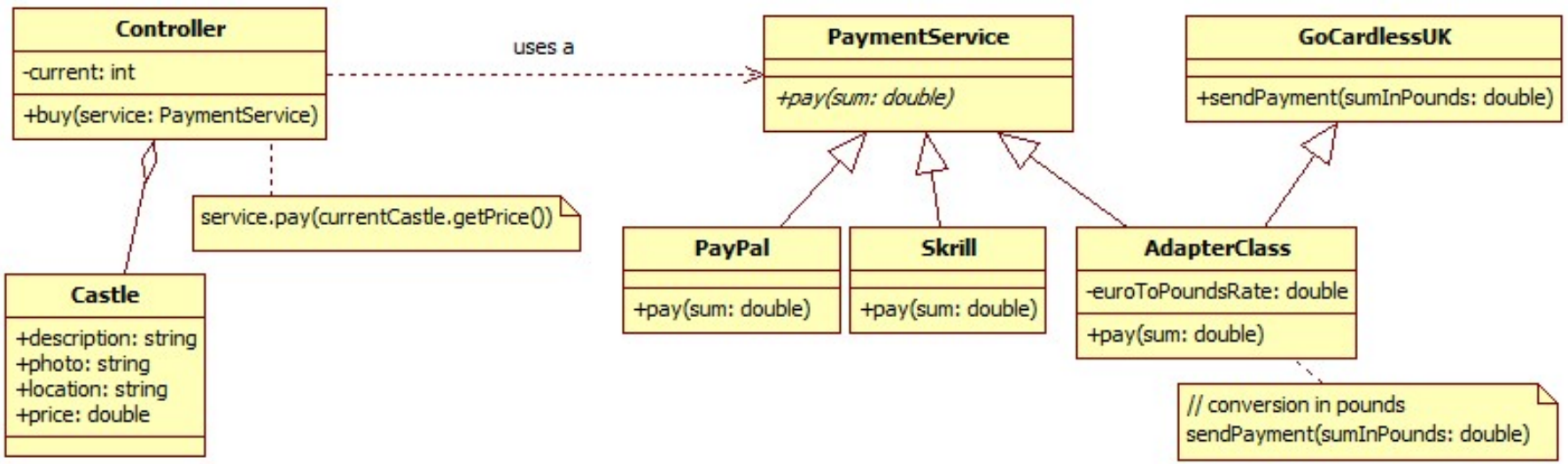
Adapter VIII



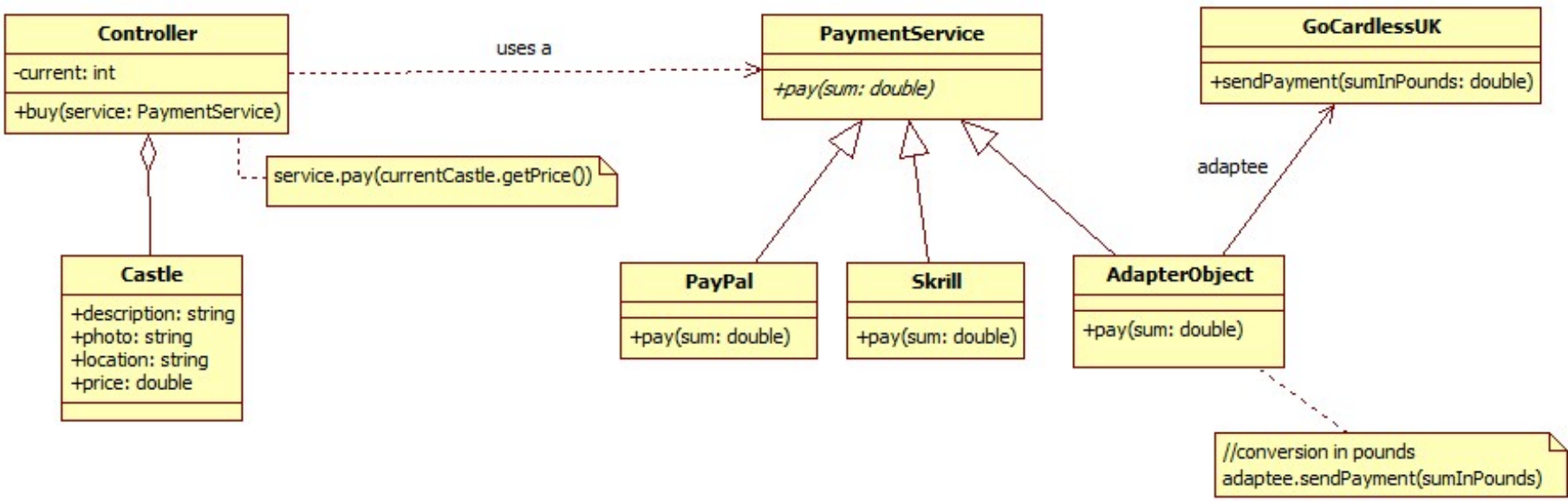
Adapter IX


Controller
-current: int
+buy(service: PaymentService)
uses a
PaymentService
+pay(sum: double)
service.pay(currentCastle.getPrice())
PayPal
+pay(sum: double)
Skrill
+pay(sum: double)
Castle
+description: string
+photo: string
+location: string
+price: double
X
GoCardlessUK
+sendPayment(sumInPounds: double)

Adapter X


Controller
-current: int
+buy(service: PaymentService)
uses a
PaymentService
+pay(sum: double)
GoCardlessUK
+sendPayment(sumInPounds: double)
service.pay(currentCastle.getPrice())
Castle
+description: string
+photo: string
+location: string
+price: double
PayPal
+pay(sum: double)
Skrill
+pay(sum: double)
AdapterClass
-euroToPoundsRate: double
+pay(sum: double)
// conversion in pounds
sendPayment(sumInPounds: double)

Adapter XI


Controller
-current: int
+buy(service: PaymentService)
uses a
PaymentService
+pay(sum: double)
GoCardlessUK
+sendPayment(sumInPounds: double)
service.pay(currentCastle.getPrice())
adaptee
Castle
+description: string
+photo: string
+location: string
+price: double
PayPal
+pay(sum: double)
Skrill
+pay(sum: double)
AdapterObject
+pay(sum: double)
//conversion in pounds
adaptee.sendPayment(sumInPounds)

Adapter XII

Adapter

Adapter.Main

Adapter XIII

Consequences

- A class adapter:
  - because it extends the *Adaptee* class, it will not work when we want to adapt a class *and all its subclasses*.
  - the *Adapter* can override some of the *Adaptee*'s behaviour.
- An object adapter:
  - lets a single *Adapter* work with many *Adaptees*.
  - makes it harder to override *Adaptee* behavior (will require subclassing *Adaptee* and making Adapter refer to the subclass).

Composite I

Problem

- We have an application in which we can have simple (primitive) objects and more complex objects.
- A simple object is processed differently than a more complex one, however we want to be able to process them in a unitary manner.

Solution

The **Composite** design pattern.

Composite II

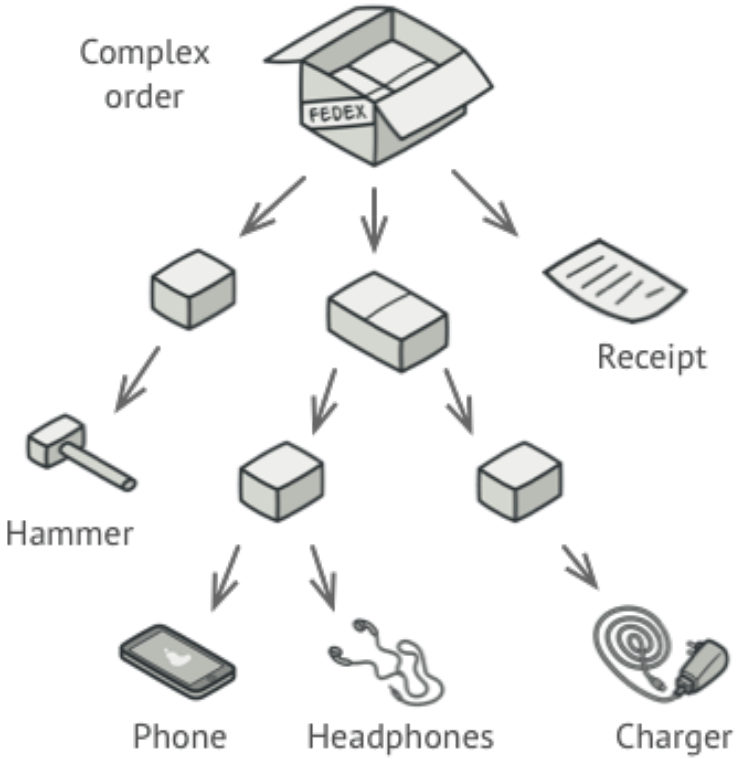


Figure: Figure source: Refactoring guru: Composite

Composite III

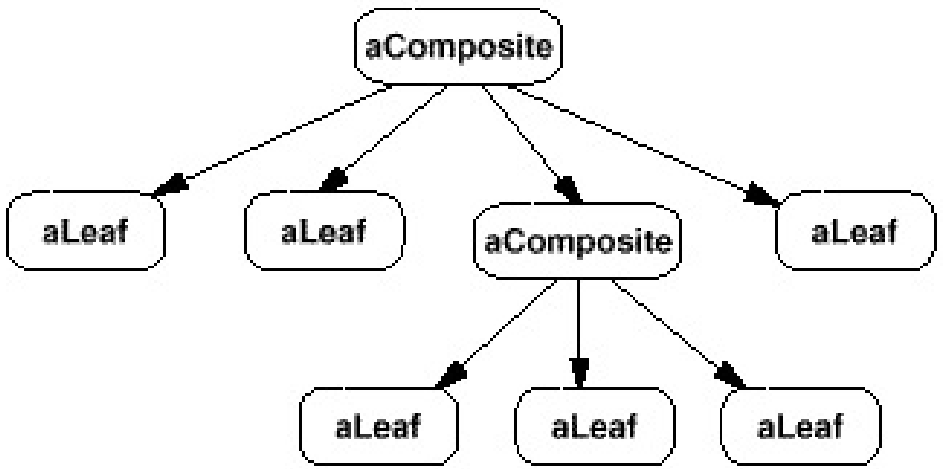


Figure: Figure source: GoF book

Composite IV

Intent

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

Composite V

Applicability

Use *Composite* when:

- we want to represent part-whole hierarchies of objects.
- we want client code to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite VI

Structure

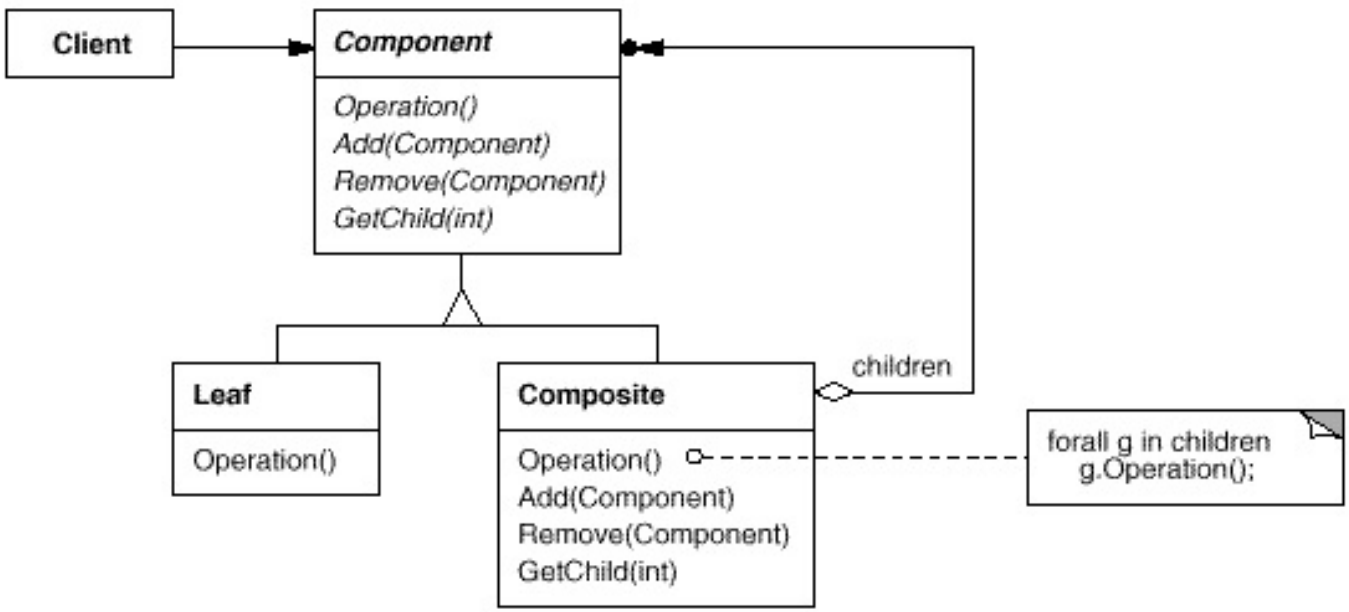


Figure: Figure source: GoF book

Composite VII

Example

- Graphic application for building complex diagrams out of simple components.
- The user can group components to form larger components, which in turn can be grouped to form still larger components.
- A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
- Code must treat primitive and container objects differently and this makes the application more complex.

Composite VIII

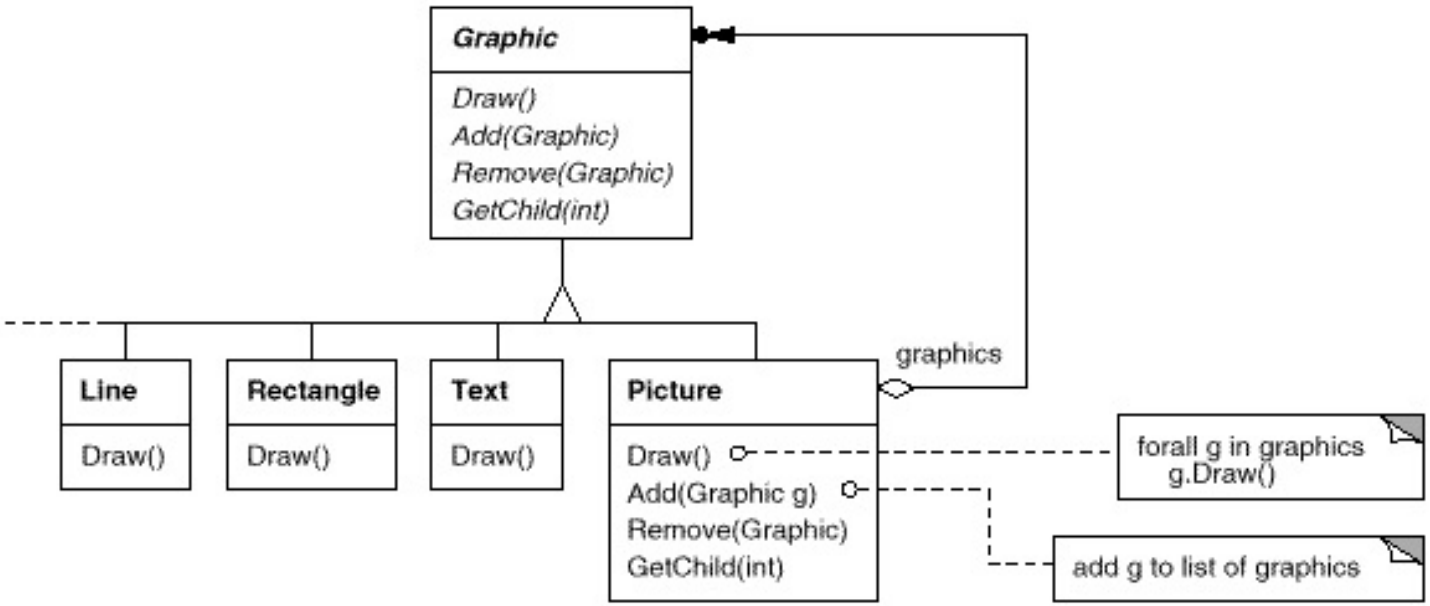


Figure: Figure source: GoF book

Composite IX

- The key to the Composite pattern is an abstract class that represents *both* primitives and their containers - class **Graphic**.
- This class will define operations that are specific to all graphical objects (e.g. *Draw()* and implement child-related operations).
- The "primitives" (Line, Rectangle, Text) will know how to draw themselves, but they do not need to manage any children.
- More complex objects (e.g. a Picture), which contain more Graphic objects will call *Draw()* on its children.

Composite X

How it works

- **Component** (Graphic)
  - declares the interface for objects in the composition.
  - declares an interface for accessing and managing its child components.
- **Leaf** (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behavior for primitive objects in the composition.

Composite XI

- **Composite** (Picture)
  - defines behaviour for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client**
  - manipulates objects in the composition through the Component interface.

Composite

Composite.Main

Composite XII

Consequences

- Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
- Makes the client simple. Clients can treat composite structures and individual objects uniformly.
- Makes it easier to add new kinds of components. Clients don't have to be changed for new Component classes.

Decorator I

Problem

- In a coffee shop we have different types of coffee and several types of condiments that can be added, each having different prices.
- How can we create an application in which the classes can be easily extensible and incorporate behaviour without modifying existing code?

Decorator II

Types of coffee

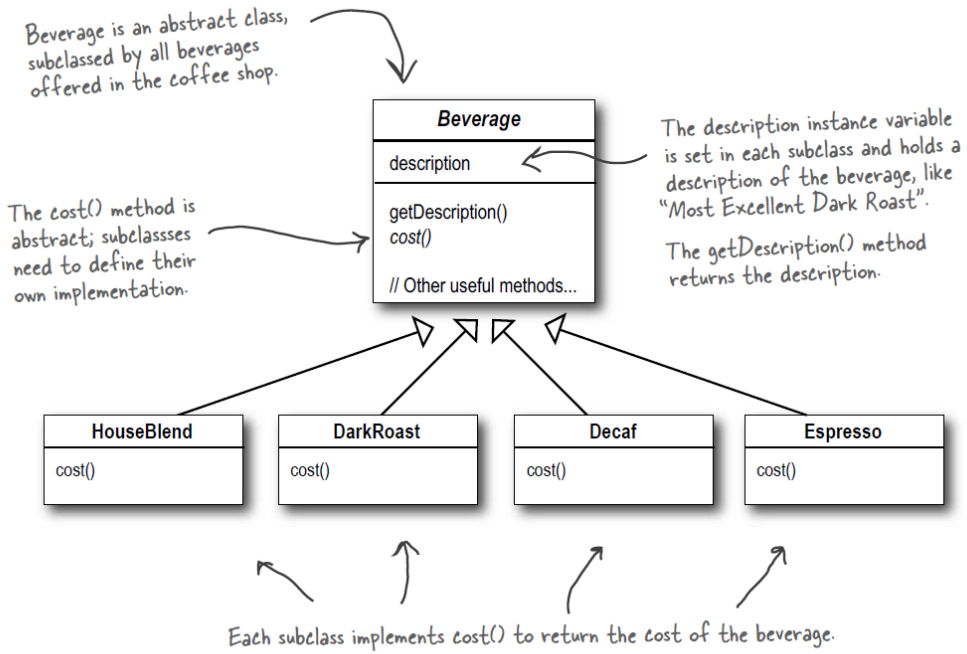


Figure: Figure source: Freeman, E., Robson, E., Bates, B., and Sierra, K. (2008). Head first design patterns. O'Reilly Media, Inc.

Decorator III

Types of condiments

- steamed milk, chocolate (mocha), hazelnuts, whipped milk.

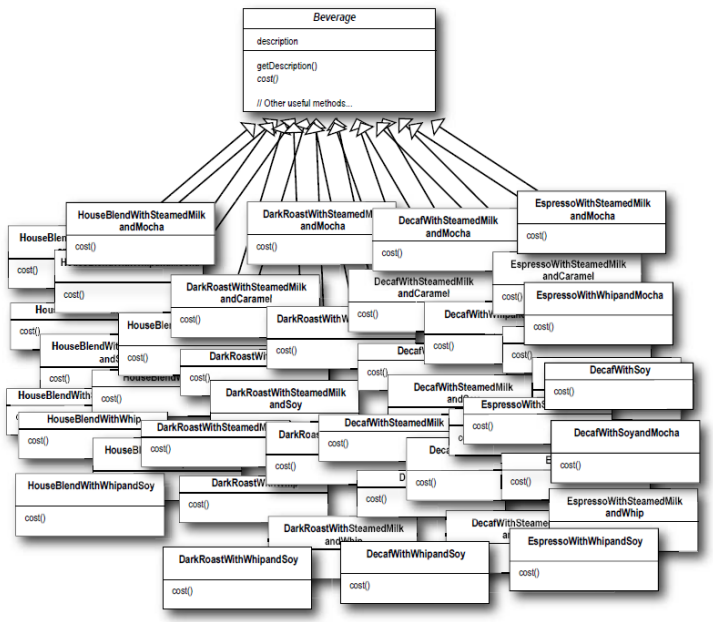


Figure: Figure source: Freeman, E., Robson, E., Bates, B., and Sierra, K. (2008). Head first design patterns. O'Reilly Media, Inc.

Decorator IV

Intent

- Attach additional responsibilities to an object dynamically.
- Provide a flexible alternative to subclassing for extending functionality.

Also known as

Wrapper

Decorator V

Applicability

Use *Decorator* when:

- we want to add or withdraw responsibilities to/from individual objects dynamically and without affecting other objects.
- extension by subclassing is impractical (see example above).

Decorator VI

Structure

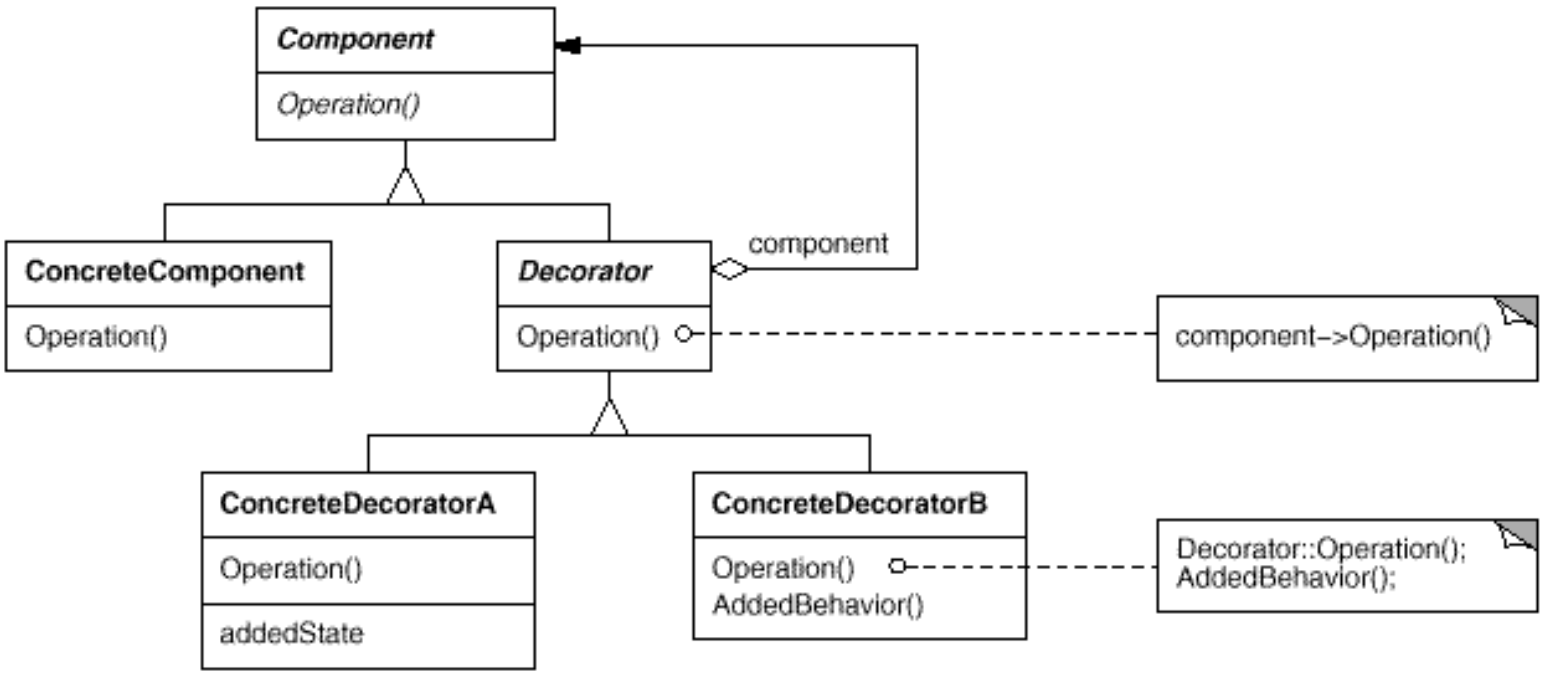


Figure: Figure source: GoF book

Decorator VII

Solution for the coffee shop

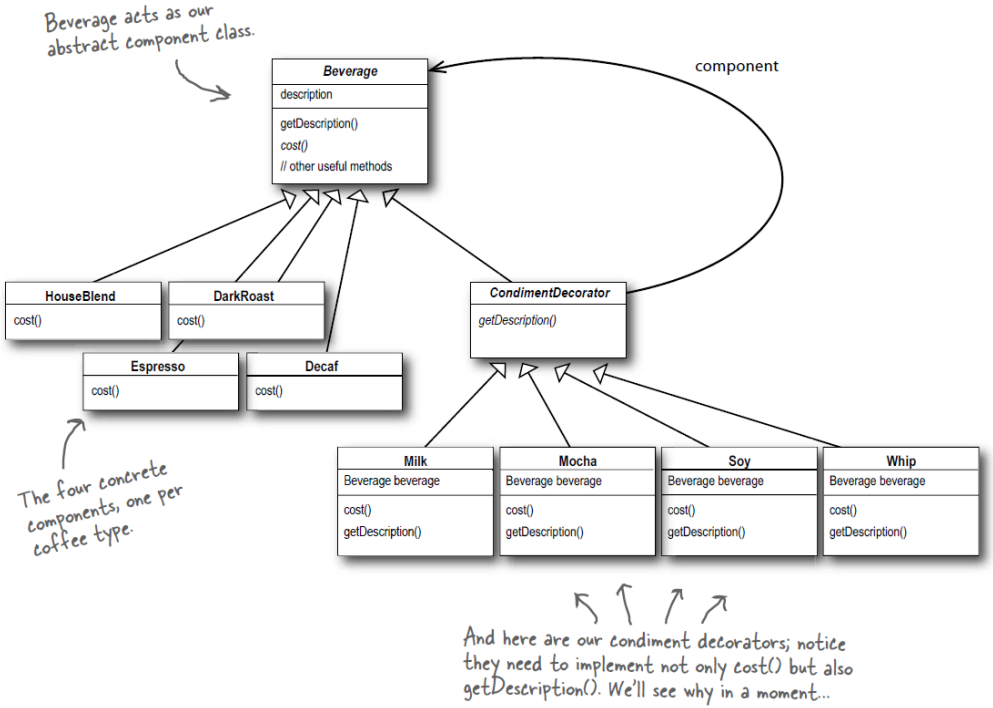


Figure: Figure source: Freeman, E., Robson, E., Bates, B., and Sierra, K. (2008). Head first design patterns. O'Reilly Media, Inc.

Decorator VIII

- The key to the Decorator pattern is an interface that represents objects that have responsibilities that can be added to them dynamically.
- The Decorator maintains a reference to a Component object and defines an interface that conforms to Component's interface, while a concrete Decorator actually adds responsibilities to the component.

Decorator IX

Consequences

- The patterns provides more flexibility than static inheritance.
- Responsibilities can be added and removed at runtime by simply attaching or detaching them.
- The complexity of the system is decreased (compared to the case when using just inheritance - class explosion).
- Decorators allow adding the same property twice (e.g. coffee with double chocolate).

Behavioural patterns

- Behavioural patterns are concerned with algorithms and the assignment of responsibilities between objects.
- They describe not just patterns between objects, but patterns of communications between them and the way objects are interconnected.
- They characterise complex control flow that's difficult to follow at run-time.

Observer I

Problem

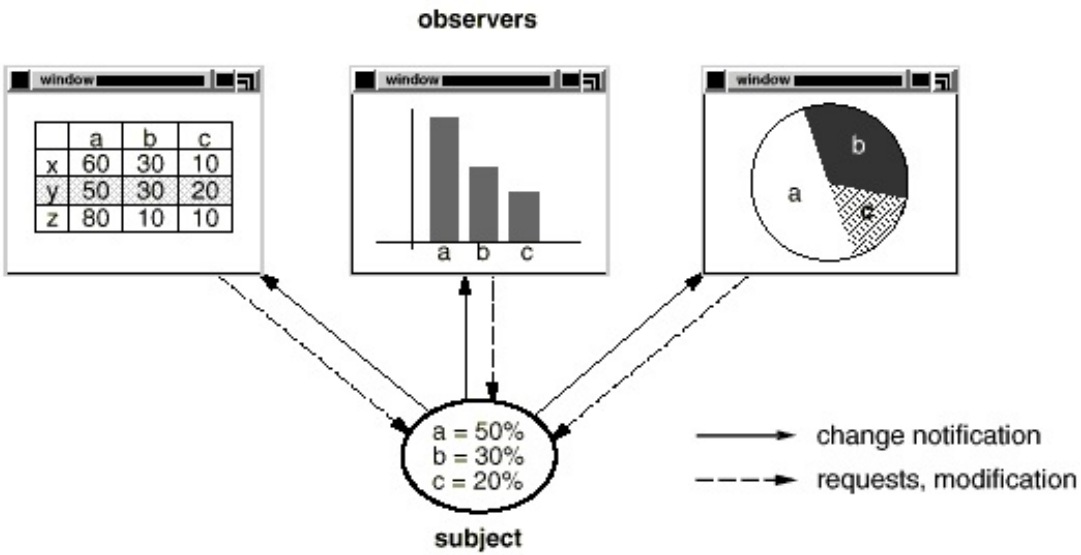


Figure: Figure source: GoF book

Observer II

Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also Known As

- Dependents, Publish-Subscribe.

Observer III

How it works

- Key objects: **subject** and **observer**.
- A subject may have many observers.
- All observers are notified when the subject is changed.
- Each observer will query the subject to synchronize its state with the subject's state.

Observer IV

Applicability

Use *Observer* when:

- a change to one object requires changing others, and you don't know how many objects need to be changed.
- an object should be able to notify other objects without making assumptions about who these objects are.

Observer V

Structure

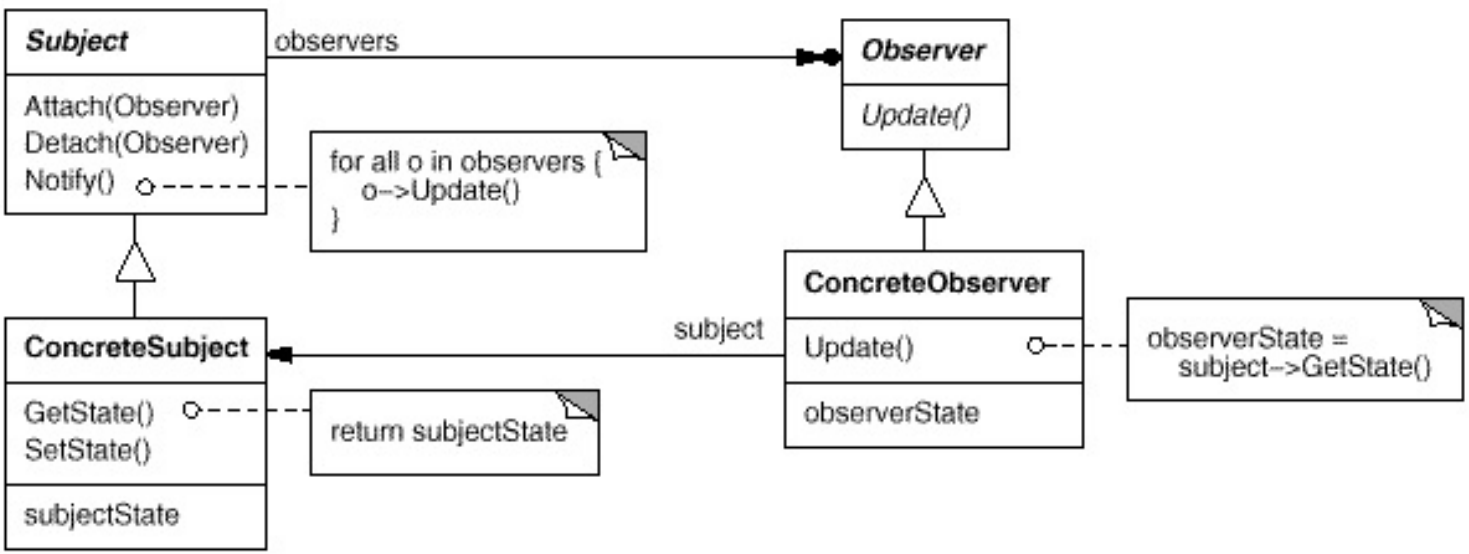


Figure: Figure source: GoF book

Observer VI

Consequences

- The subject and the observer are *loosely coupled* (the subject knows that it has a list of observers, but does not know their concrete classes).
- Support for broadcast communication; observers can be added or removed at any time.

Observer VII

Observer in Java

- Observer (interface) and Observable (class) had been implemented in Java, but have been deprecated since Java 9.
- Observers could be added/removed to/from the observable object using the function addObserver()/deleteObserver().
- Whenever the Observable changed, it had to bemarked as having been changed (method setChanged()) and it could notify observers via the method notifyObservers.

Example

Please see Seminar7 (Order and OrderController).

Strategy I

Problem

- Route building application: builds routes between sources and destination, by car.
- Later, we decide to add walking routes.
- Later, routes using public transport.
- Later, routes for cyclists.
- The class that computes the navigation routes becomes bigger and bigger.

Strategy II

Intent

- Algorithms from a class that do something specific, each one differently, are extracted into separate classes called *strategies*.
- Using Strategy, we can define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.

Also Known As

- Policy

Strategy III

Applicability

Use *Strategy* when:

- many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours.
- you need different variants of an algorithm.

Strategy IV

Structure

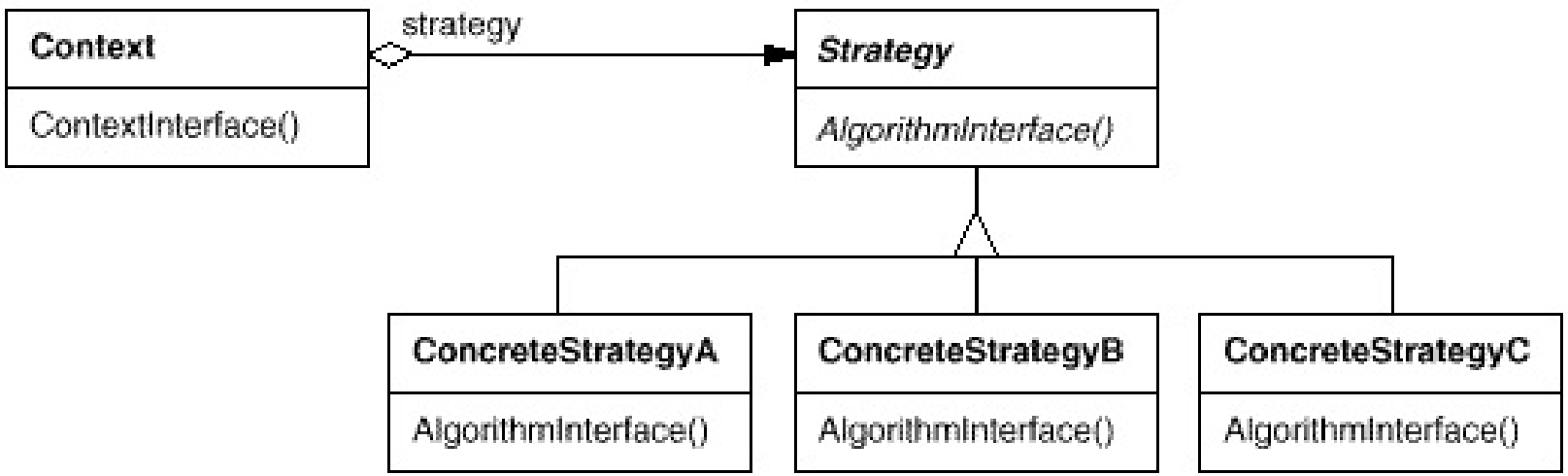


Figure: Figure source: GoF book

Strategy V

Consequences

- We can create families of related algorithms and using inheritance we can factor out common functionality of these algorithms.
- Strategies eliminate conditional statements.
- Strategies can also provide different implementations of the same behaviour.
- A drawback is that the client must be aware of different existing strategies.
- Another drawback is that the number of objects in the application is increased.

Strategy VI

Example

Please see Seminar7 (PaymentStrategy).

Summary I

- Software design patterns: recurring descriptions of classes and communicating objects that are customised to solve a general design problem in a particular context.
- Three categories: creational (factory method, abstract factory), structural (adapter, composite, decorator), behavioural (observer, strategy, command).

Summary II

- Knowing what pattern to apply in which situation (applicability) and the consequences it has allows one to design better, reusable architectures.
- *"Design patterns help a designer get a design "right" faster"*. (GoF book).
- *Next week:*
  - Introduction in C#.