SOFTWARE DESIGN & ARCHITECTURE
(Design Patterns)

USAMA MUSHARAF

MS-CS (Software Engineering)

LECTURER (Department of Computer Science)

FAST-NUCES PESHAWAR

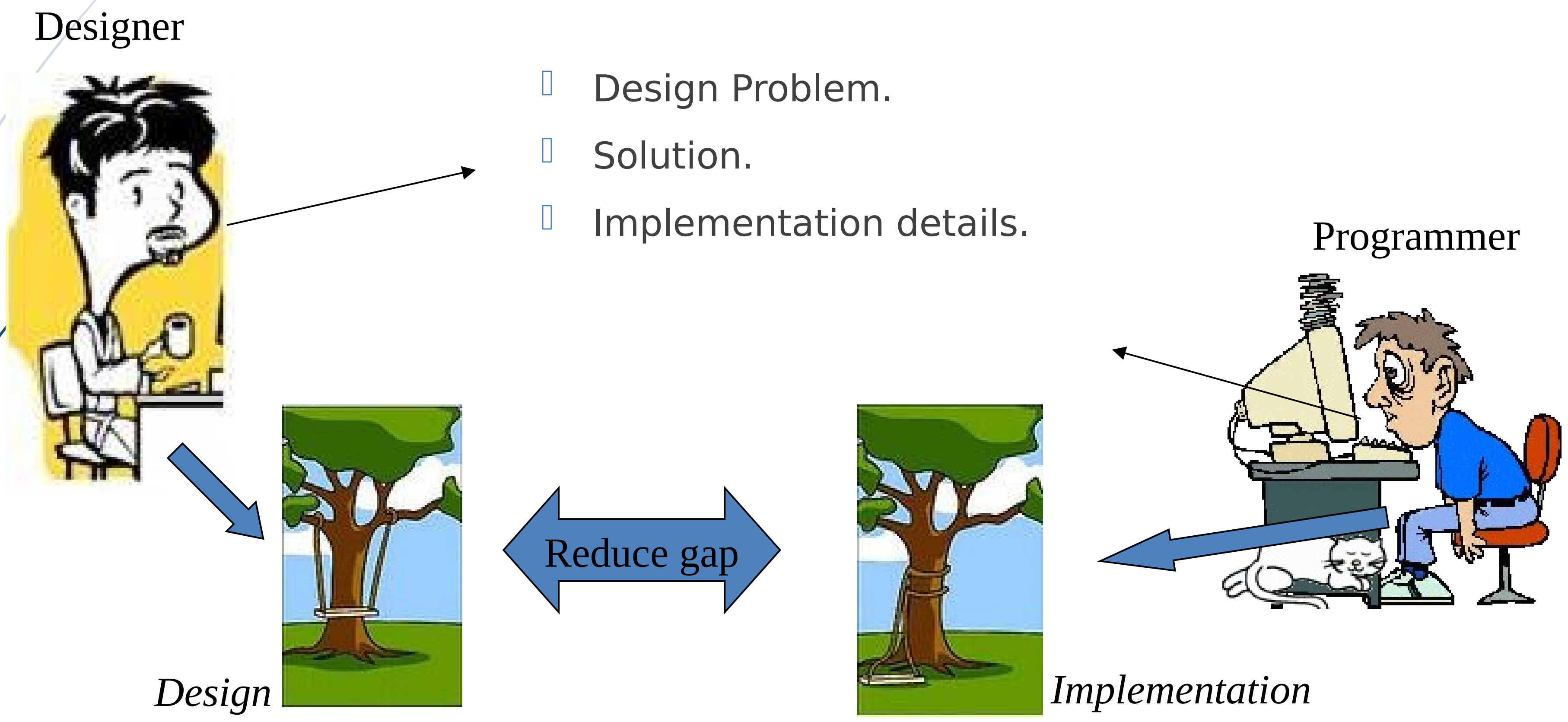
Software Design Patterns

Introduction to design patterns

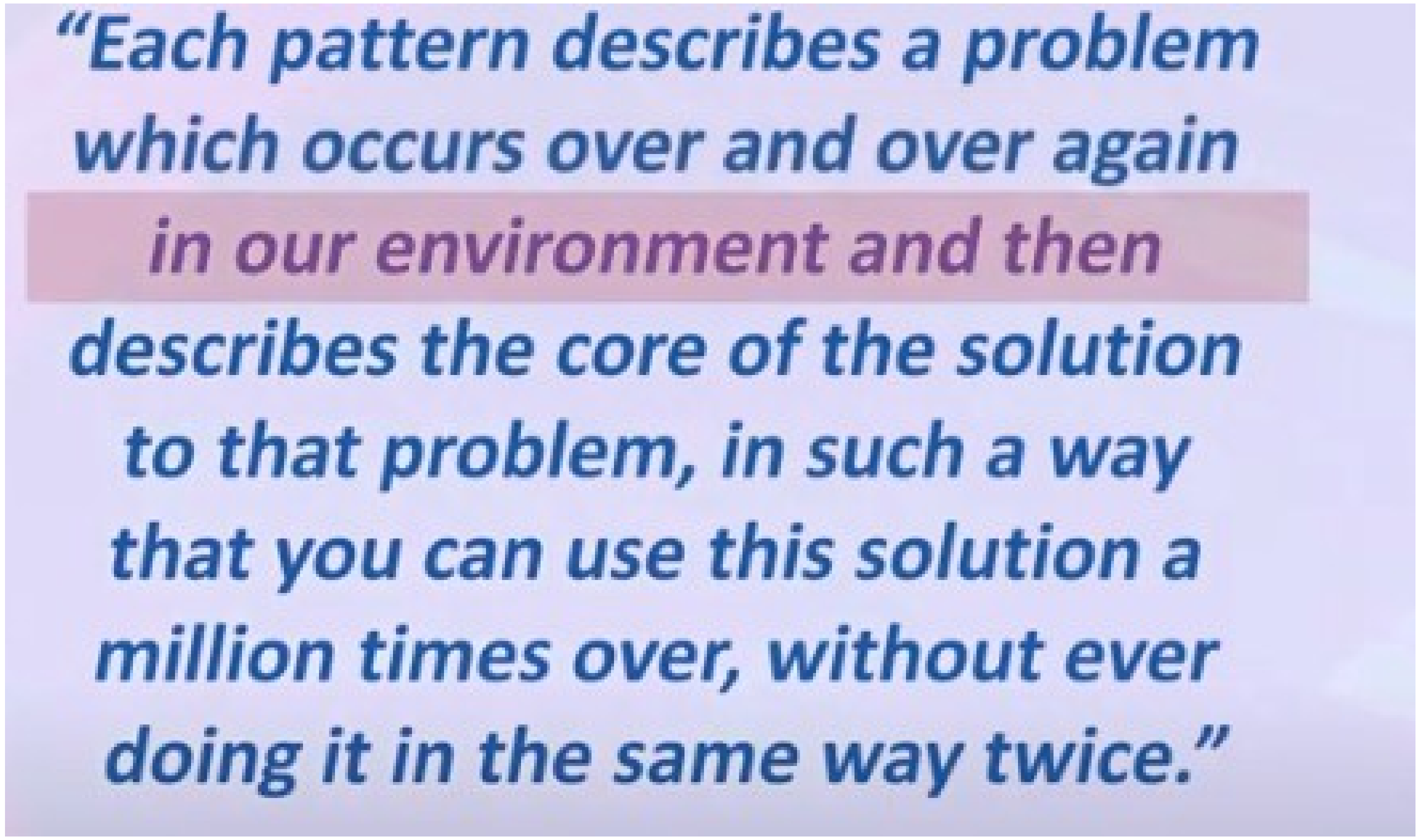
- A good design is more than just knowing and applying OO concepts like abstraction, inheritance, and polymorphism.
- Designers focuses on creating flexible designs that are more maintainable and that can cope with changes easily.

How Patterns are used

- Design Problem.
- Solution.
- Implementation details.



Pattern

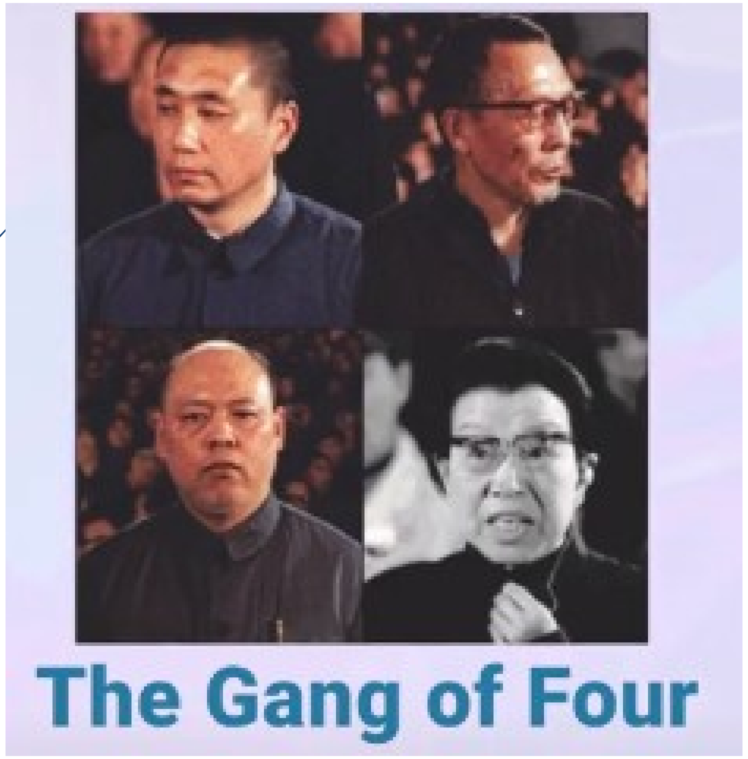

"Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice."

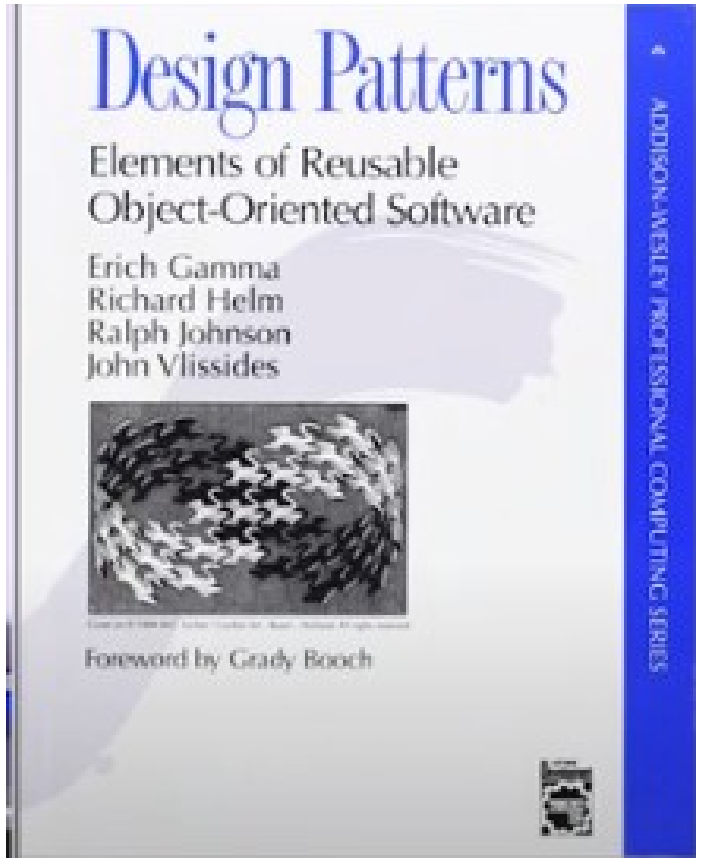
Pattern

A Pattern must:

- Solve a problem and be useful.
- Have a context and can describe where the solution can be used.
- Recur in relevant situations.
- Provide sufficient understanding to tailor the solution.

Gang of four




Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Elements of Design Pattern

Elements of design pattern

Design patterns have four essential elements:

- Pattern name
- Problem
- Solution
- Consequences

Design pattern vs Framework

Pattern	Framework
Design patterns are recurring solutions to the problems that arise during the life of a software application in a particular context.	A frame work is a group of components that cooperate with each other to provide a reusable architecture.
Primary Goal	Primary Goal
• Improves quality of the software in terms of the software being reusable, maintainable, extensible etc. • Reduces development time	• Improves quality of the software in terms of the software being reusable, maintainable, extensible etc. • Reduces development time

Design pattern vs Framework

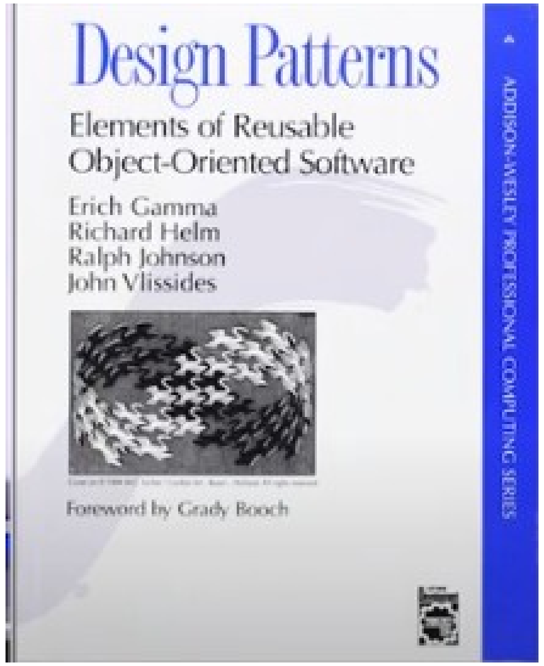
Pattern	Framework
Patterns are logical in nature.	Frameworks are more physical in nature as they exist in the form of software.
Independent of programming language and implementation.	Implementation Specific.
Patterns provide a way to do good design and are used to help design frameworks.	Design patterns may be used in the design and implementation of a framework.

Categories of Design Pattern

Categories of design pattern

This book defined 23 patterns in three categories

- *Creational patterns* deal with the process of initializing and configuring of classes and objects (5)
- *Structural patterns,* deal primarily with the static composition and structure of classes and objects (7)
- *Behavioral patterns,* which deal primarily with dynamic interaction among classes and objects (11)
  - How they distribute responsibility


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

GoF Patterns

- *Creational Patterns*
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- *Structural Patterns*
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- *Behavioral Patterns*
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Limitations of design pattern

1. *Patterns* do not lead to direct code reuse.
2. Patterns are deceptively simple.
3. Patterns are validated by experienced rather than by automated testing.
4. Integrating patterns into a software development process is a human intensive activity.

Singleton Design Pattern

(creational pattern)

Singleton Design Pattern

Sometimes there may be a need to have one and only one instance of a given class during the lifetime of an application.

Eg. Database Connection

Singleton Design Pattern ensures that there is only one instance of a class and provides global point of access to it.

Code for Singleton Design Pattern

```
public class Singleton
{
private static Singleton instance;
private Singleton()     // Private Constructor
     {   }
public static Singleton getInstance()
         {
         if (instance == null)
         { instance = new Singleton(); }
     return instance;
          }
}
```

```
public class testsingleton
 {
public static void main (String args[])
      {
      Singleton.getInstance();  // call to static
method
      }
 }
```

Singleton design pattern

Problem Statement:

In Chocolate manufacturing industry, there are computer controlled chocolate boilers. The job of boiler is to take in milk and chocolate, bring them to boil and then pass it on to the next phase of chocolate manufacturing process.

We have to make sure that bad things don’t happen like filling the filled boiler or boiling empty boiler or draining out unboiled mixture.

Code

```
public class ChocolateBoiler {
private boolean empty;
private boolean boiled;
private static ChocolateBoiler
uniqueins;
private ChocolateBoiler()
{
empty=true;
boiled=false;
}
```

```
public static ChocolateBoiler getInstance()
{
     if(uniqueins==null)
     {
      uniqueins=new ChocolateBoiler();
      getInstance().fill();
      getInstance().boil();
      getInstance().drain();
     }
 return uniqueins;
}
```

Factory Design Pattern

(creational pattern)

Factory Design Pattern

Factory Pattern defines an interface for creating the object but let the subclass decide which class to instantiate. Factory pattern let the class defer instantiation to the sub class.

Factory Design Pattern

Problem Statement:

If there exist class hierarchies i-e super / sub classes then client object usually know which class /sub class to instantiate but at times client object know that it needs to instantiate the object but of which class it does not know

it may be due to many factors

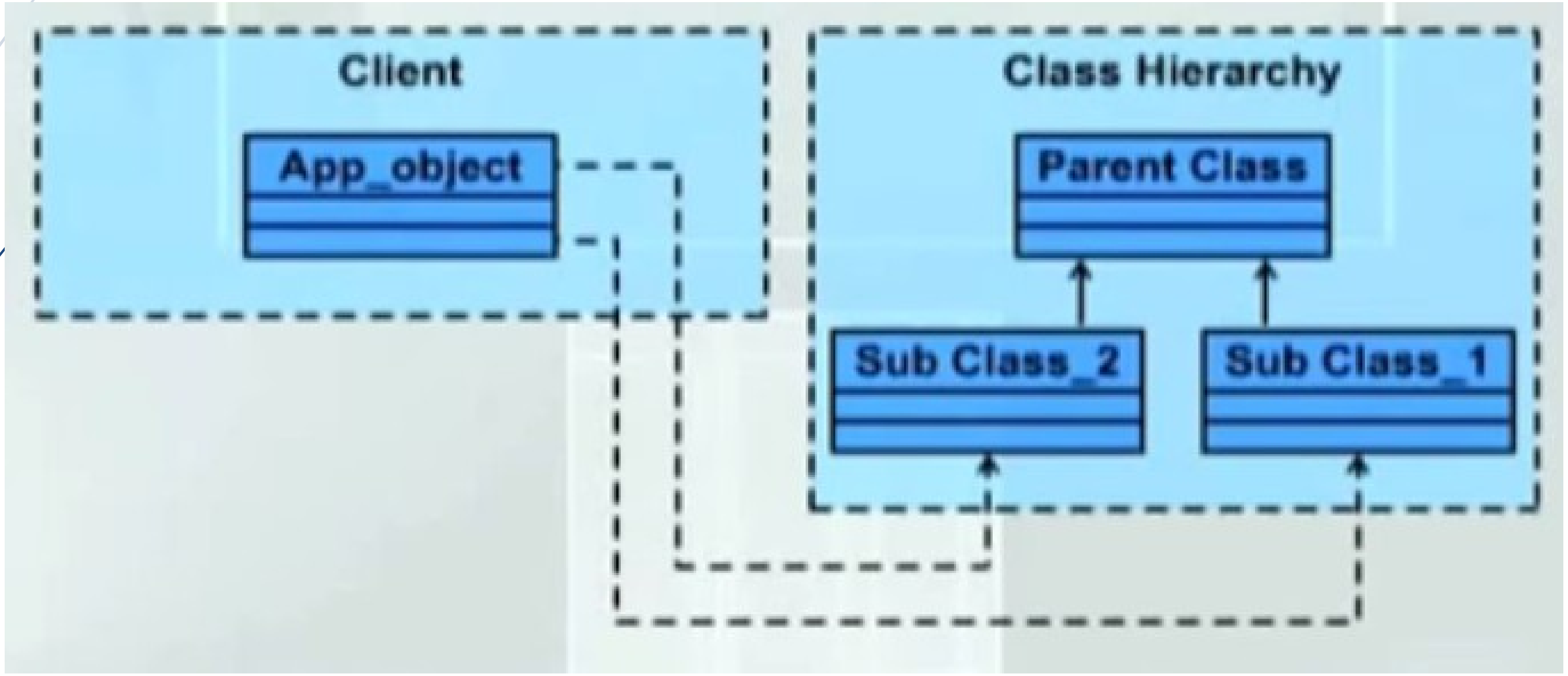
Factory Design Pattern

In such cases, an application object needs to implement the class selection criteria to instantiate an appropriate class from the hierarchy to access its services and that selection criteria will be considered as a part of the client code to access the concrete class from hierarchies of classes.

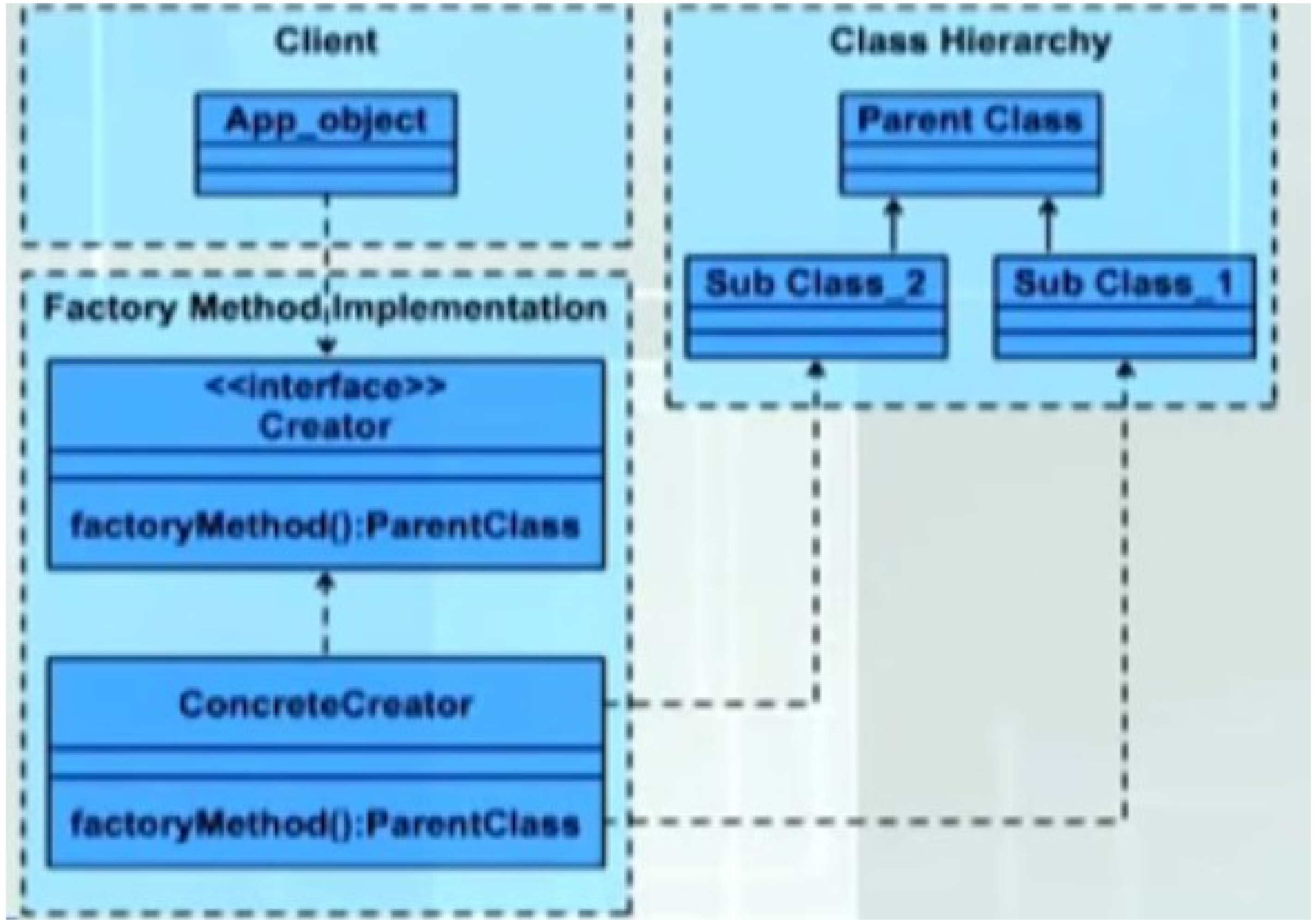
Disadvantage of this approach:

It results in high coupling.

High Degree of Coupling



Proposed Solution



Proposed solution

The solution has the build violation of principle of software design i-e “Loose coupling”; as opposite to the principle above solution is having high degree of coupling between client and classes in hierarchies.

Problem Statement

We want the user to enter the name in either “first name last name or last name, first name” format.

We have made the assumption that there will always be a comma between last name and first name and space between first name last names.

The client does not need to be worried about which class is to access when it is entering the name in either of the format. Independent of the format of the data to be entered, system will display first name and last name.

Builder Design Pattern

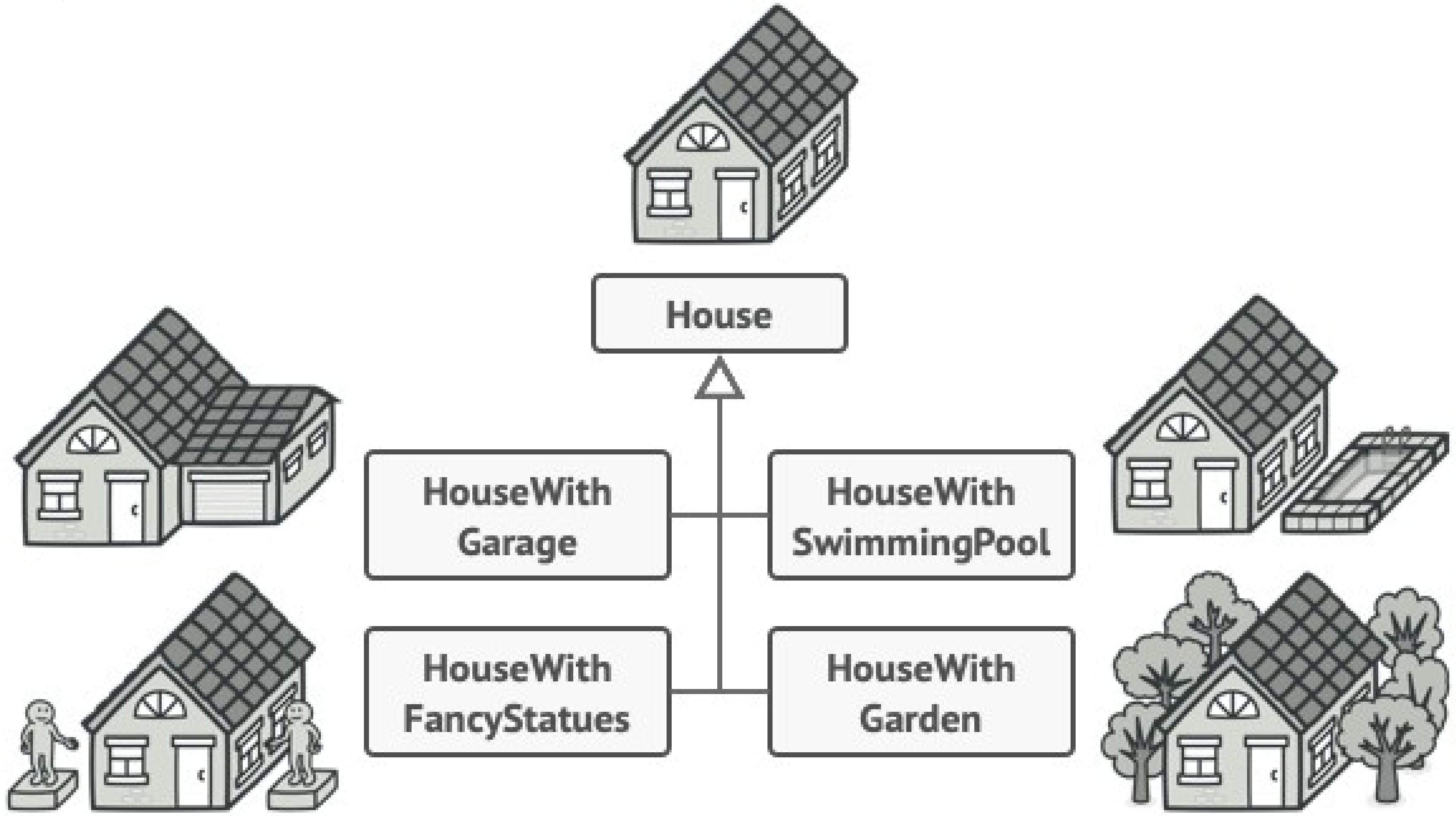
(creational pattern)

Builder Design Pattern

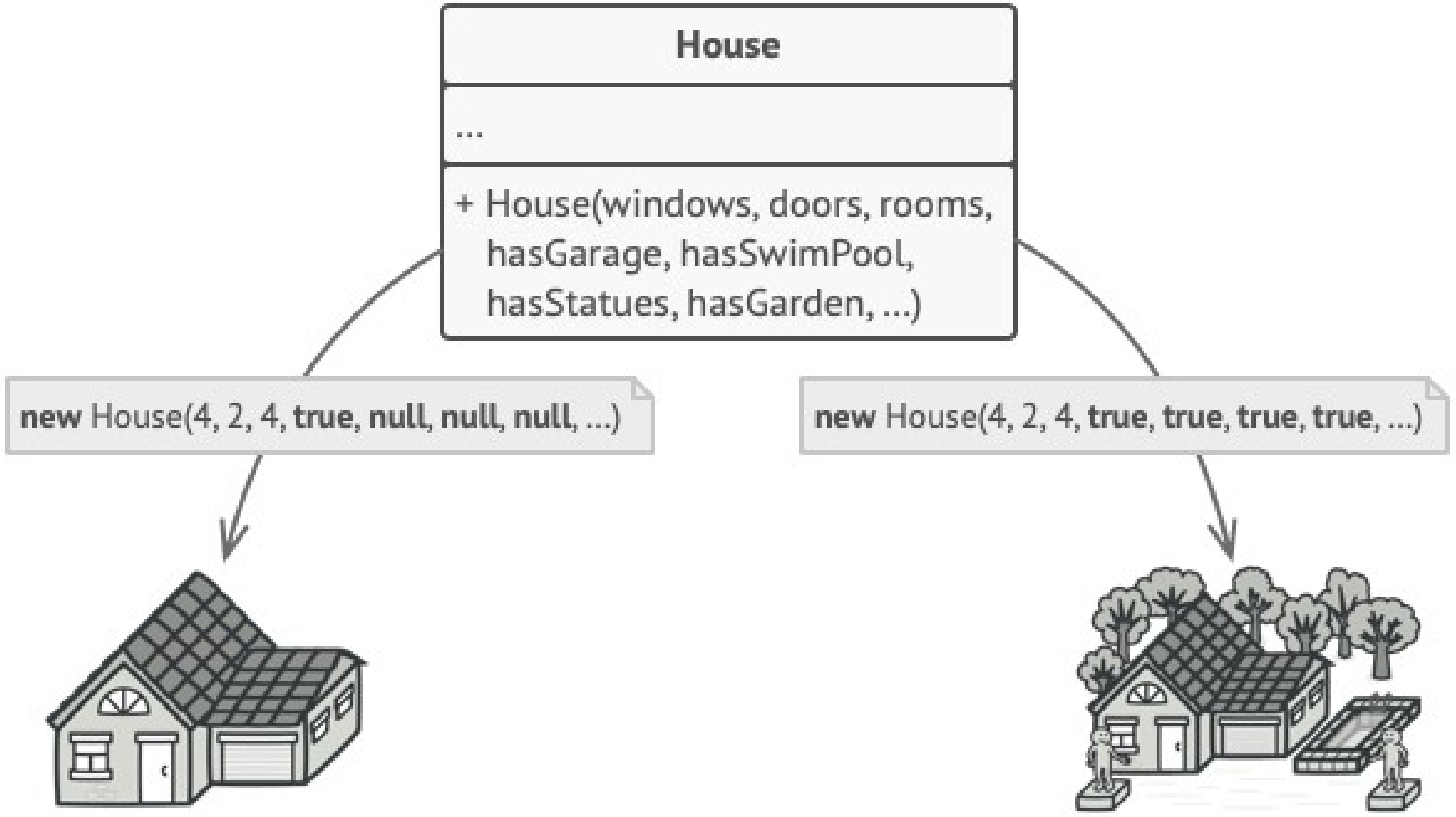
Builder is a creational design pattern that lets you construct complex objects step by step.

The pattern allows you to produce different types and representations of an object using the same construction code.

Builder Design Pattern



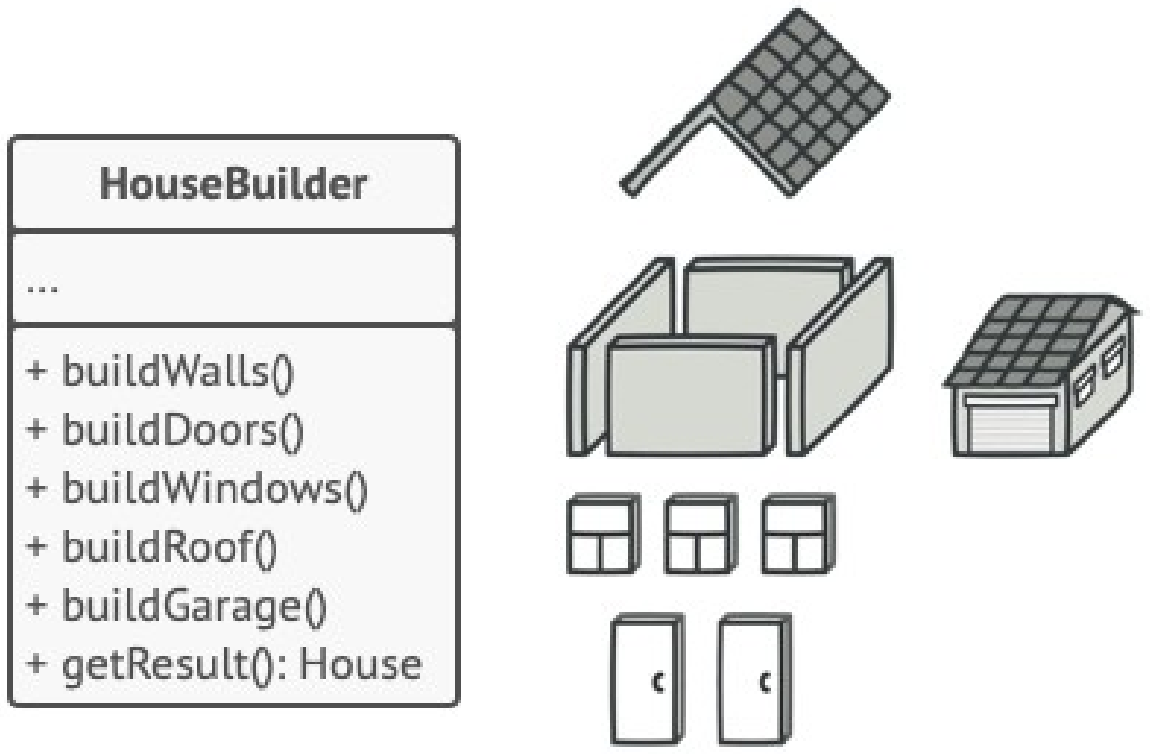
Builder Design Pattern


House
...
+ House(windows, doors, rooms, hasGarage, hasSwimPool, hasStatues, hasGarden, ...)
new House(4, 2, 4, true, null, null, null, ...)
new House(4, 2, 4, true, true, true, true, ...)

Builder Design Pattern

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called *builders*.

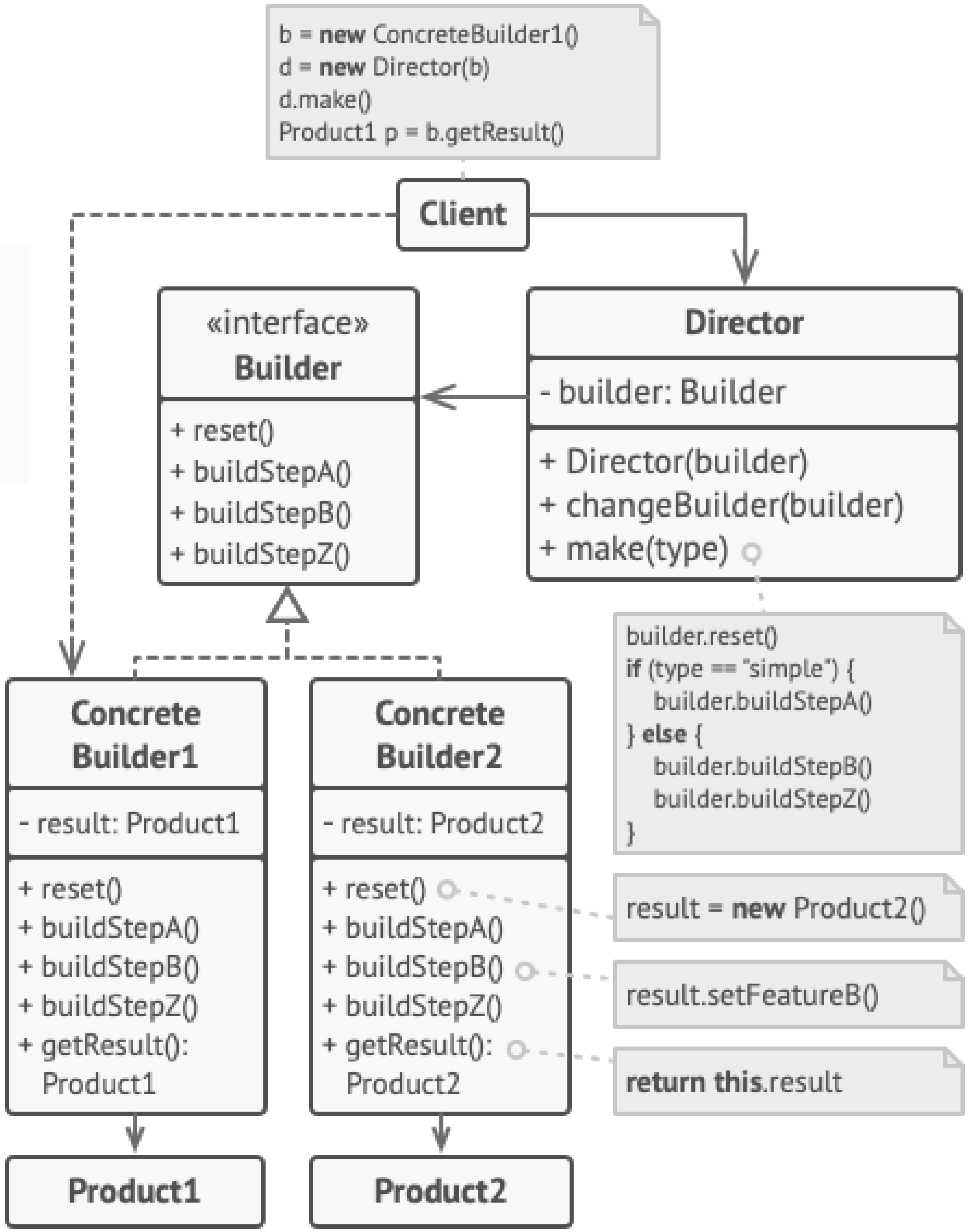
- The pattern organizes object construction into a set of steps (buildWalls, buildDoor, etc.).
- To create an object, you execute a series of these steps on a builder object.
- The important part is that you don’t need to call all of the steps.
- You can call only those steps that are necessary for producing a particular configuration of an object.



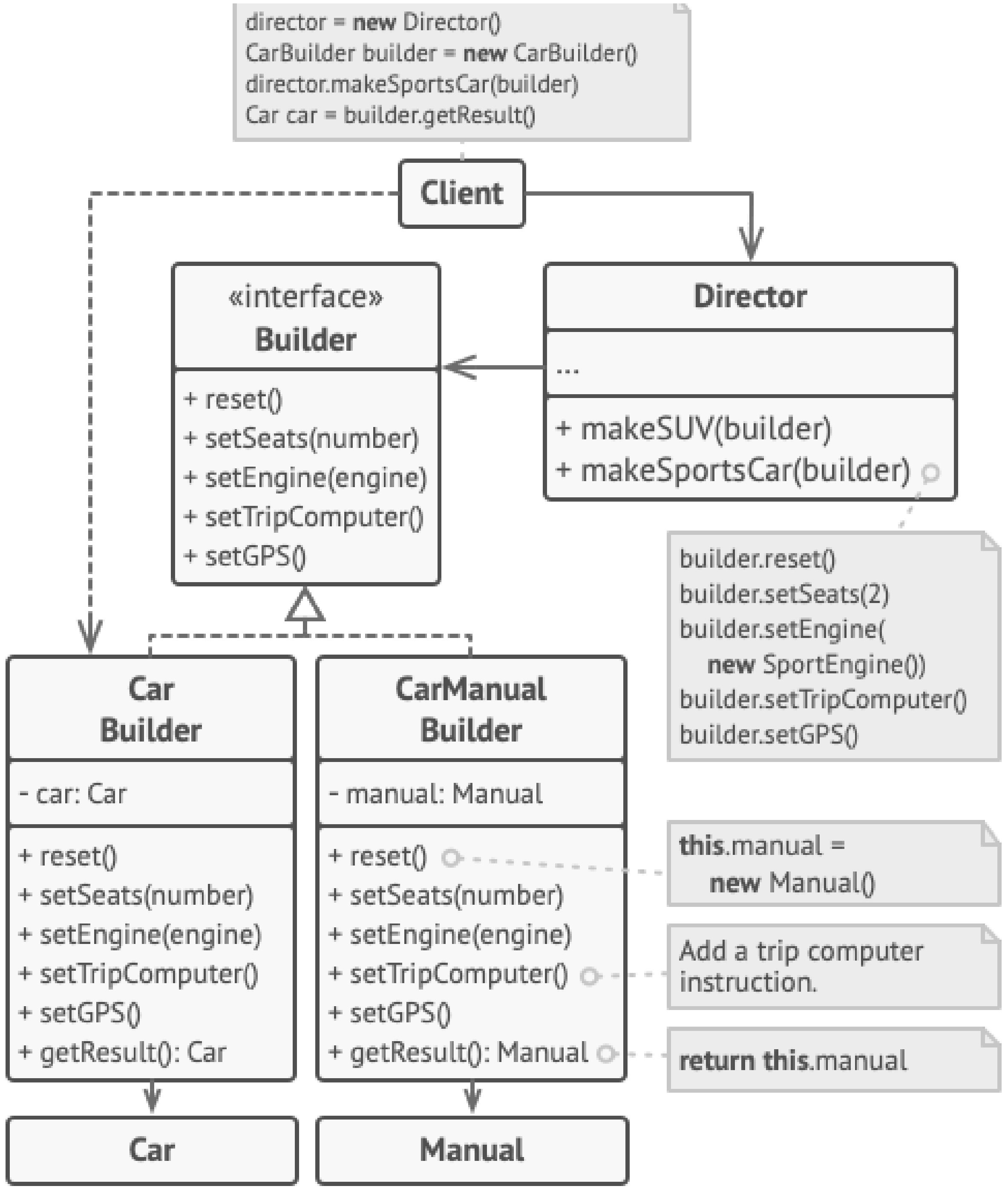
Builder Design Pattern

1.The **Builder** interface declares product construction steps that are common to all types of builders.

2. Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.



Builder Design Pattern



- *Structural Patterns*
  - **Adapter**
  - **Bridge**
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - **Proxy**

Adapter Design Pattern

(structural pattern)

Adapter Design Pattern

Adapter is a structural design pattern that allows objects with incompatible interfaces to collaborate.

Adapter Design Pattern

Problem Statement:

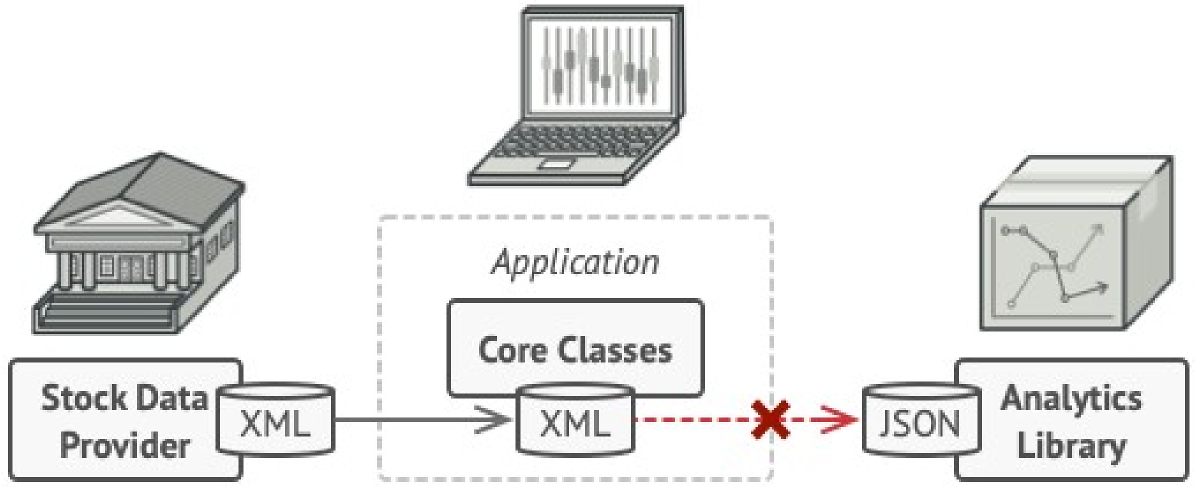
Imagine that you’re creating a stock market monitoring app.

The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.

At some point, you decide to improve the app by integrating a smart 3rd-party analytics library.

But the analytics library only works with data in JSON format.

You can’t use the analytics library “as is” because it expects the data in a format that’s incompatible with your app.



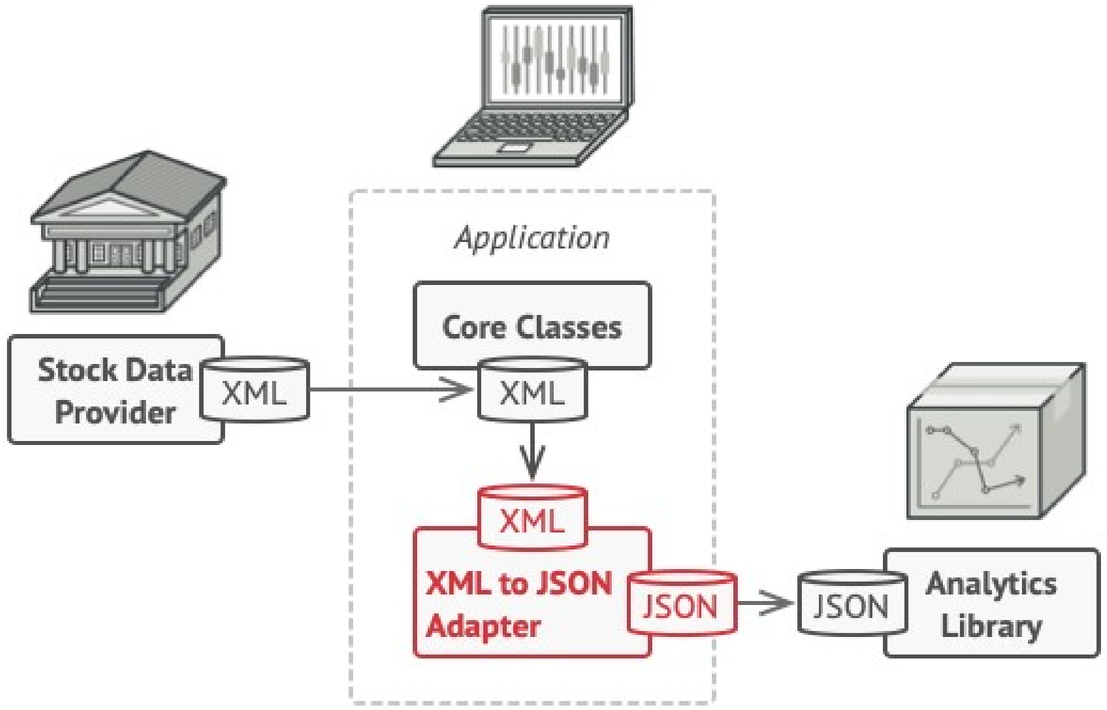
Adapter Design Pattern

Solution:

You can create an *adapter*. This is a special object that converts the interface of one object so that another object can understand it.

An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes.

The wrapped object isn't even aware of the adapter. For example, you can wrap an object that operates in meters and kilometers with an adapter that converts all of the data to imperial units such as feet and miles.



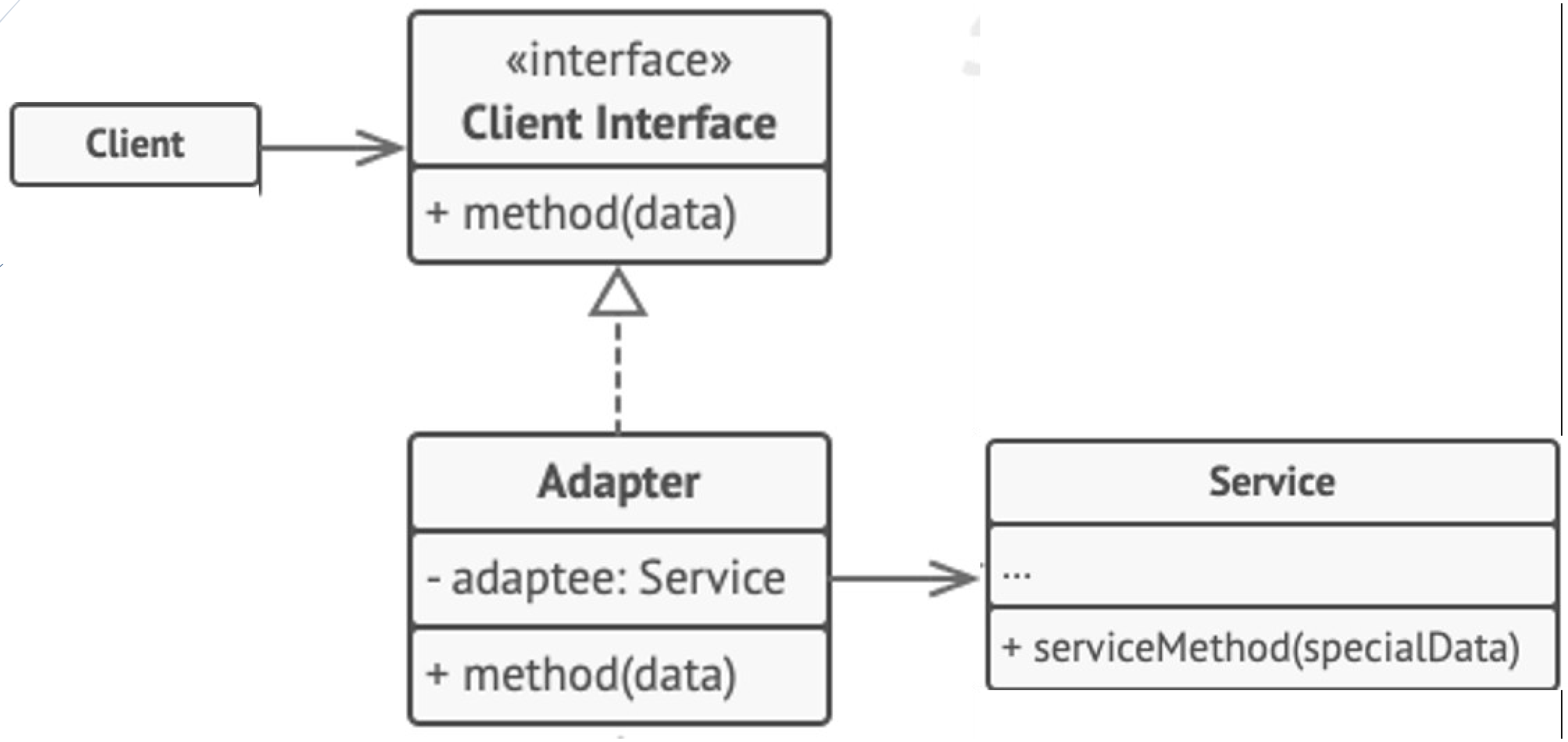
Adapter Design Pattern

Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate.

Here’s how it works:

1. The adapter gets an interface, compatible with one of the existing objects.
2. Using this interface, the existing object can safely call the adapter’s methods.
3. Upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects.

Adapter Design Pattern



Bridge Design Pattern

(structural pattern)

Bridge Design Pattern

Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies

abstraction and implementation

which can be developed independently of each other.

Bridge Design Pattern

Problem Statement:

Say you have a geometric Shape class with a pair of subclasses: Circle and Square.

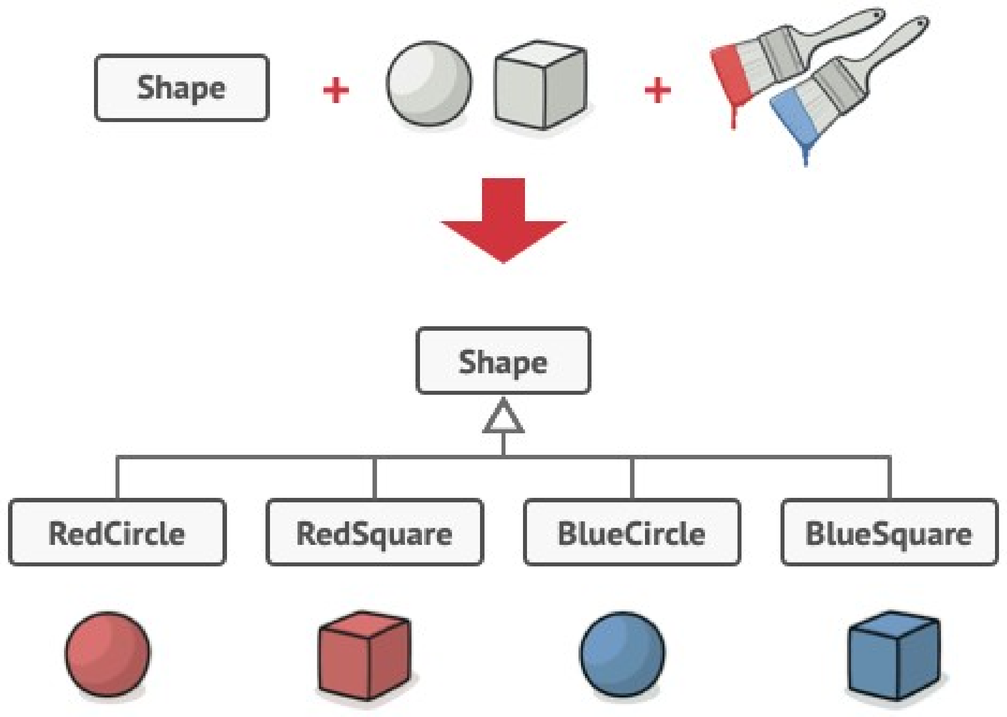
You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses.

However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.

Adding new shape types and colors to the hierarchy will grow it exponentially.

For example, to add a triangle shape you'd need to introduce two subclasses, one for each color.

And after that, adding a new color would require creating three subclasses, one for each shape type. The further we go, the worse it becomes.



Bridge Design Pattern

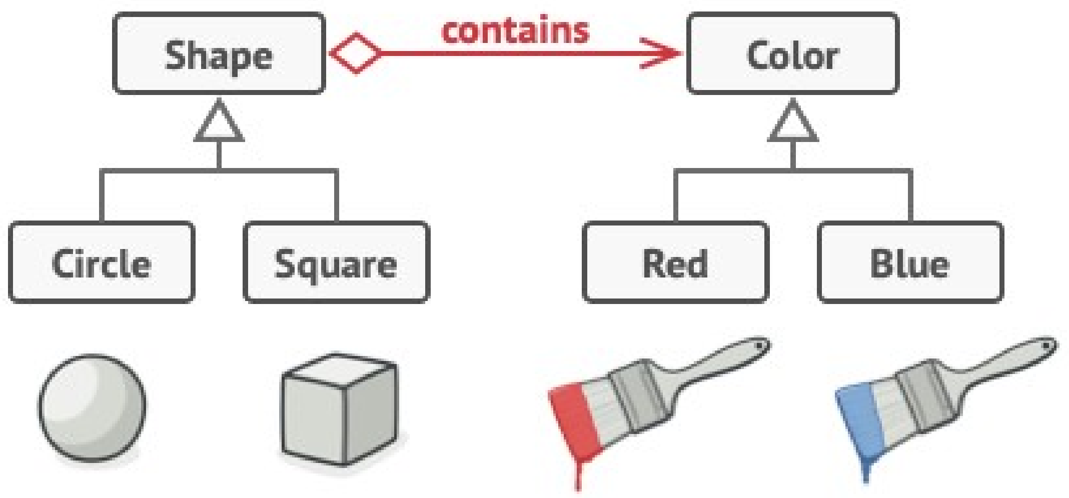
Solution:

This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color.

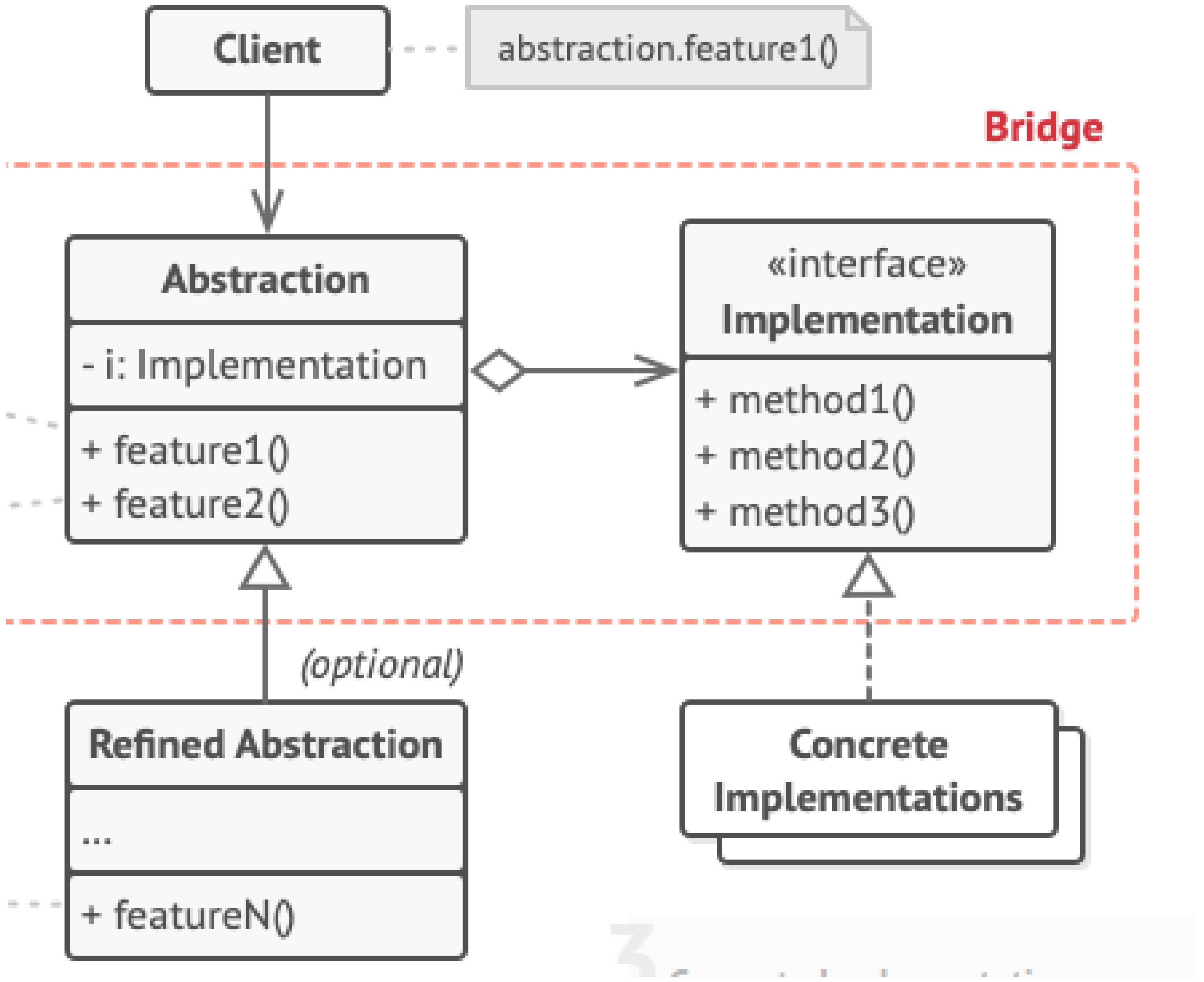
That's a very common issue with class inheritance.

The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.

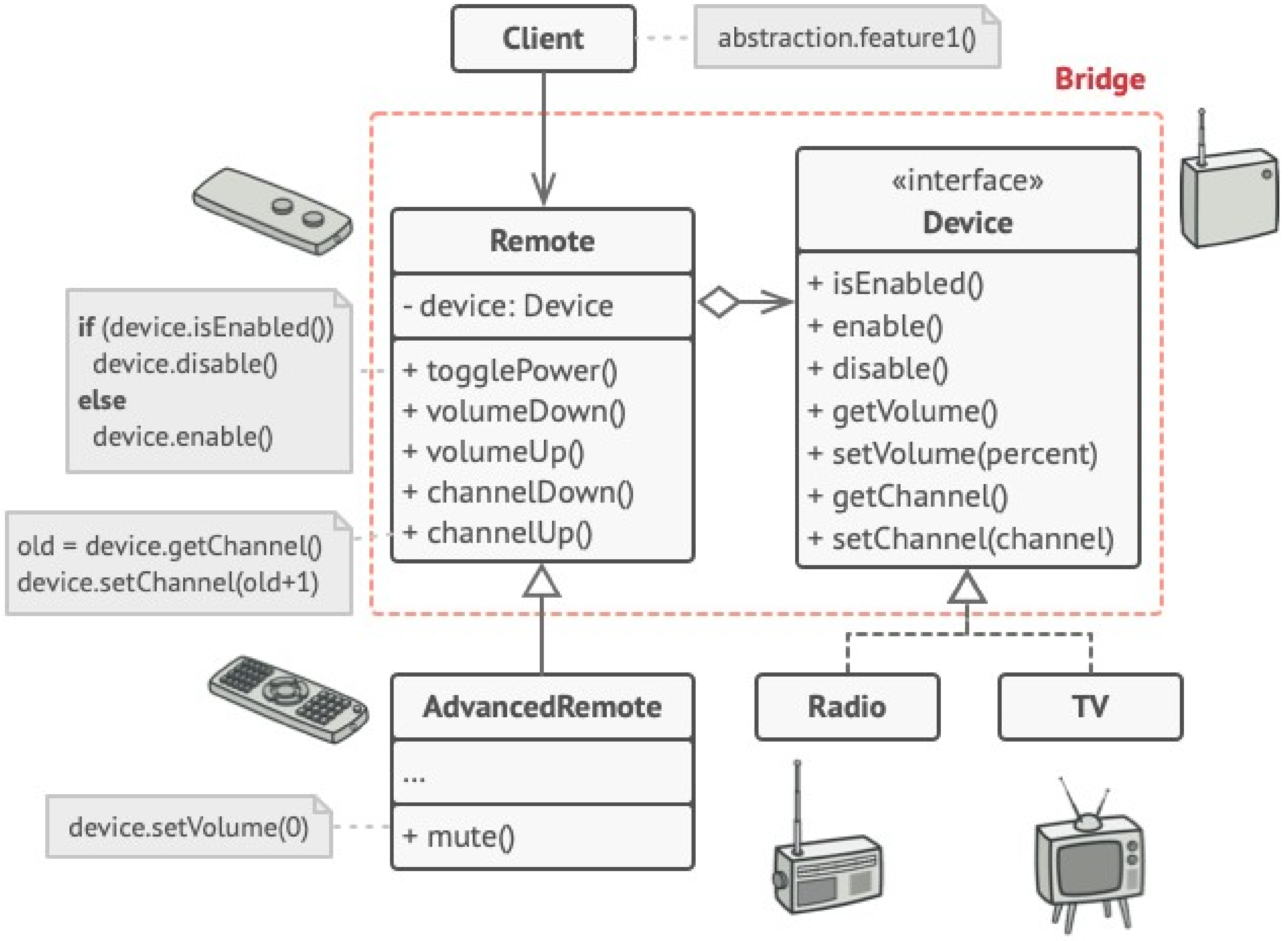
What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all of its state and behaviors within one class.



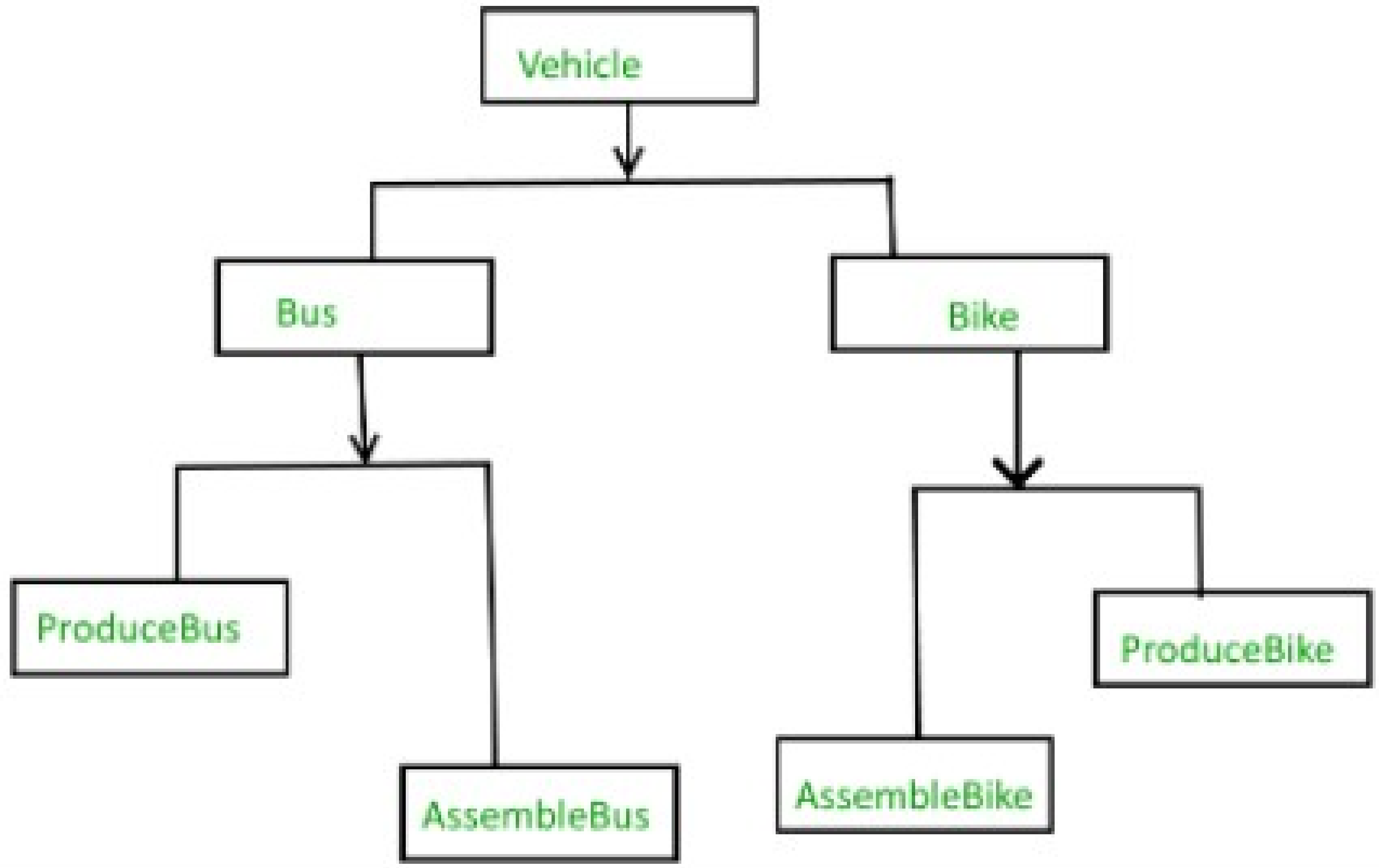
Bridge Design Pattern



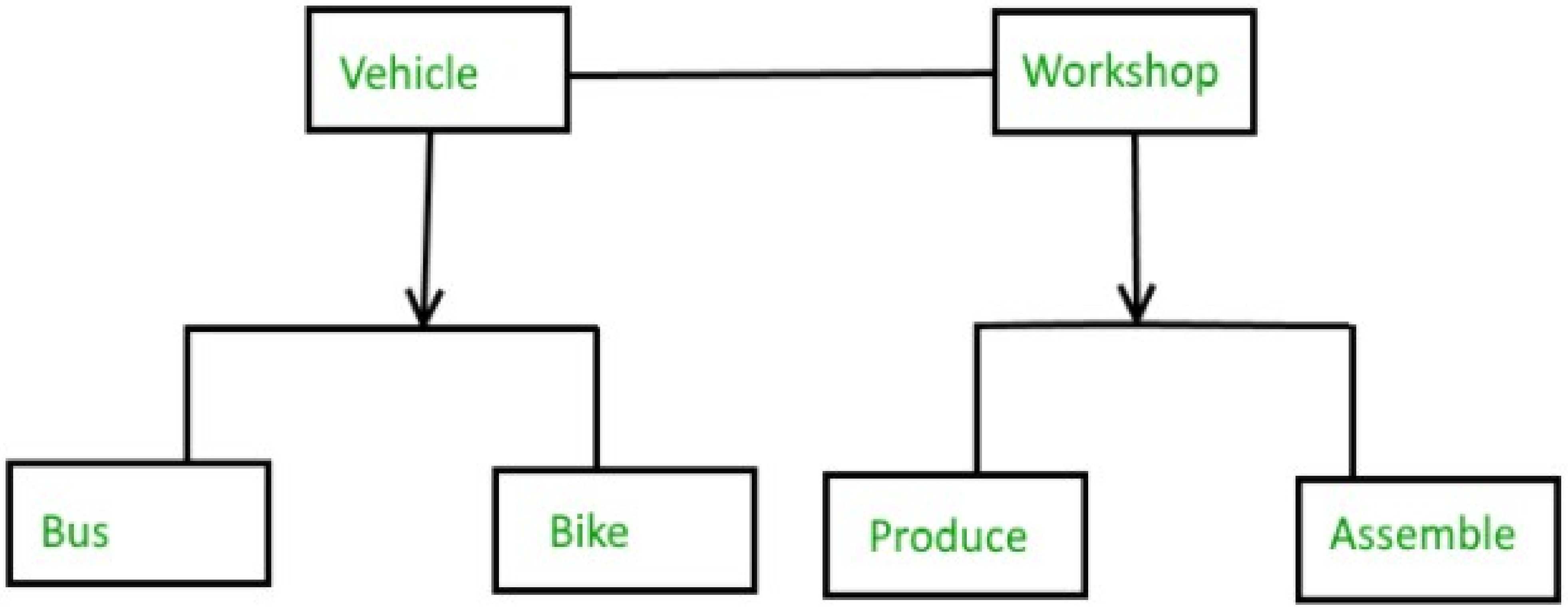
Bridge Design Pattern



Without Bridge Design Pattern



Bridge Design Pattern



Bridge Design Pattern

Example:

We have custom business logic to process employee data, and this processed employee data will be saved as an XML on a Windows machine and as a JSON file on a LINUX machine.

The saving part differs based on the operating system.

As per the Bridge Design Pattern, we may abstract (decouple) the business processing logic from the saving logic and it will have no knowledge of how the data will be saved.

The abstraction contains a reference (via composition) to the implementer.

The implementer class (saving of data) details will be provided during the runtime based on the operating system and both abstraction and implementer can be developed independently.

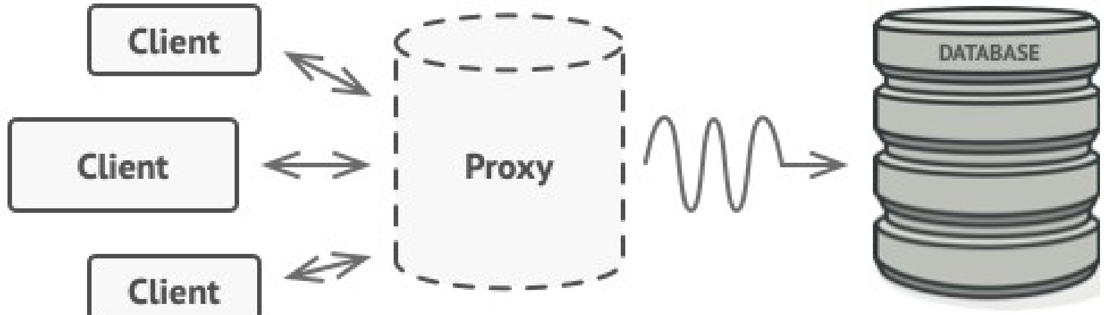
Proxy Design Pattern

(structural pattern)

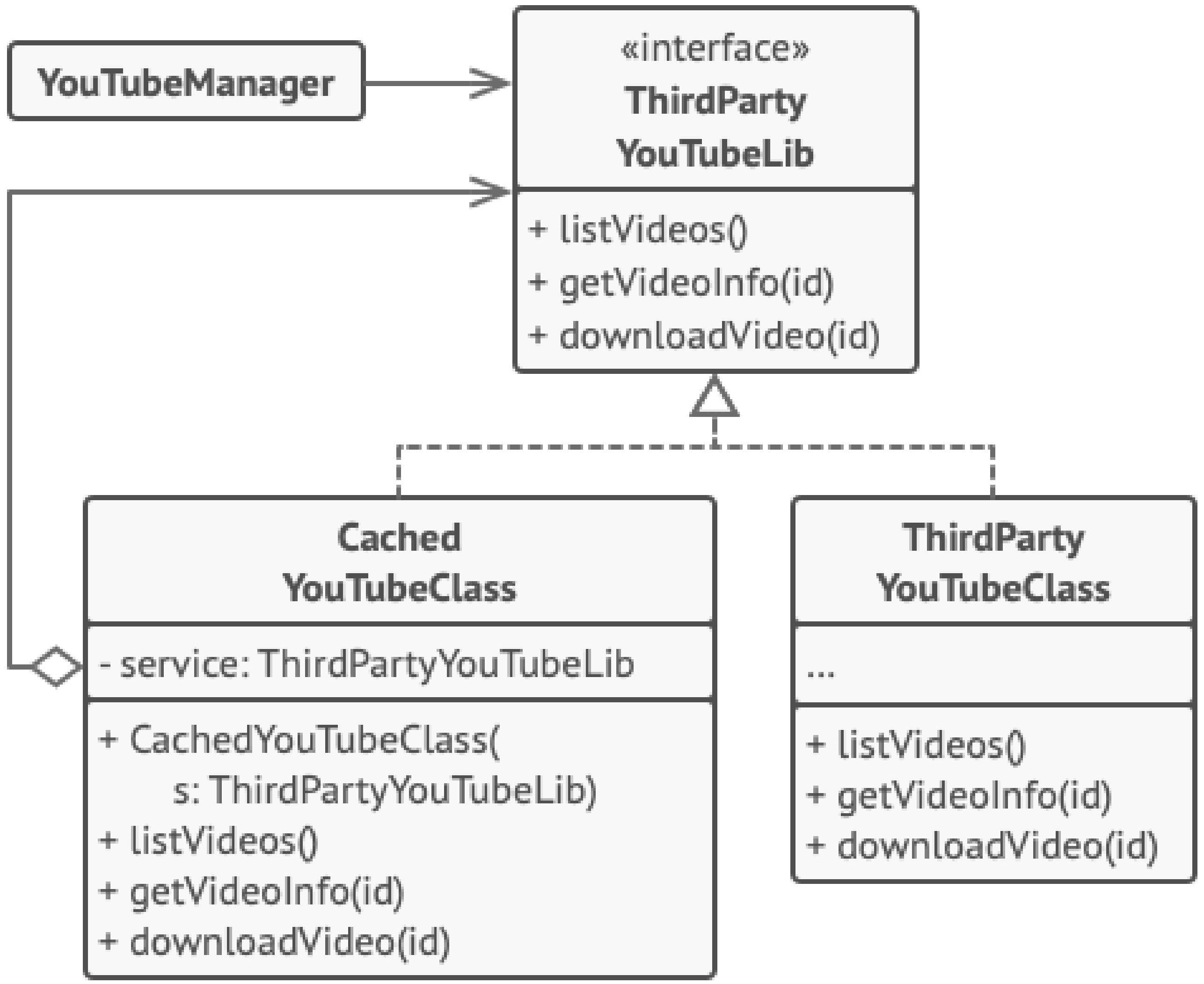
Proxy Design Pattern

Proxy is a structural design pattern that lets you provide a substitute or placeholder for another object.

A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.



Proxy Design Pattern



- *Behavioral Patterns*
  - **Chain of Responsibility**
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - **State**
  - **Strategy**
  - Template Method
  - Visitor

Strategy Design Pattern

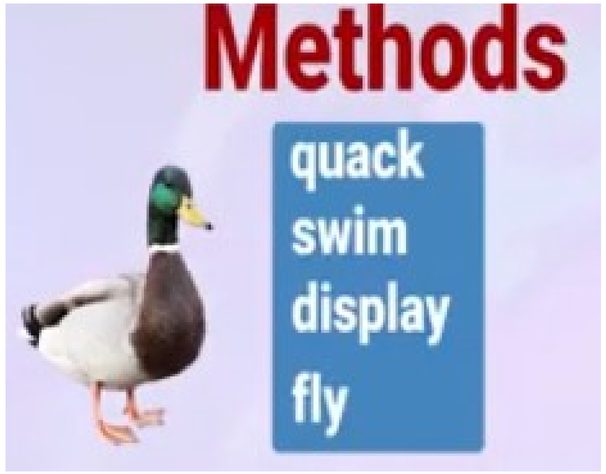
(Behavioral pattern)

Strategy Design Pattern

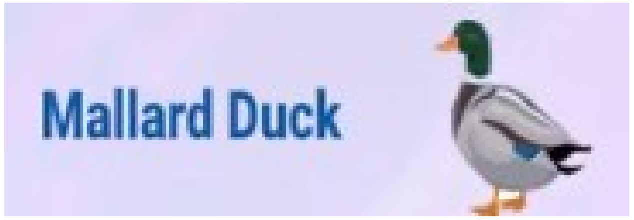
Defines a family of algorithms, encapsulates each one, and make them interchangeable,

This is really helpful if you have many passible variations of an algorithms.

Strategy Design Pattern (Example)



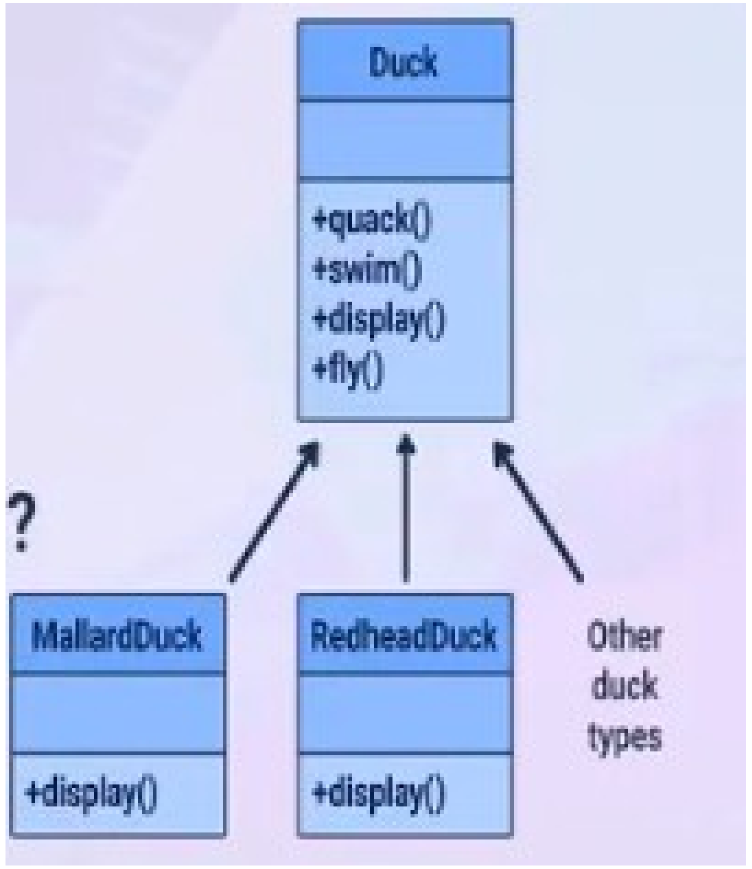
Strategy Design Pattern (Example)



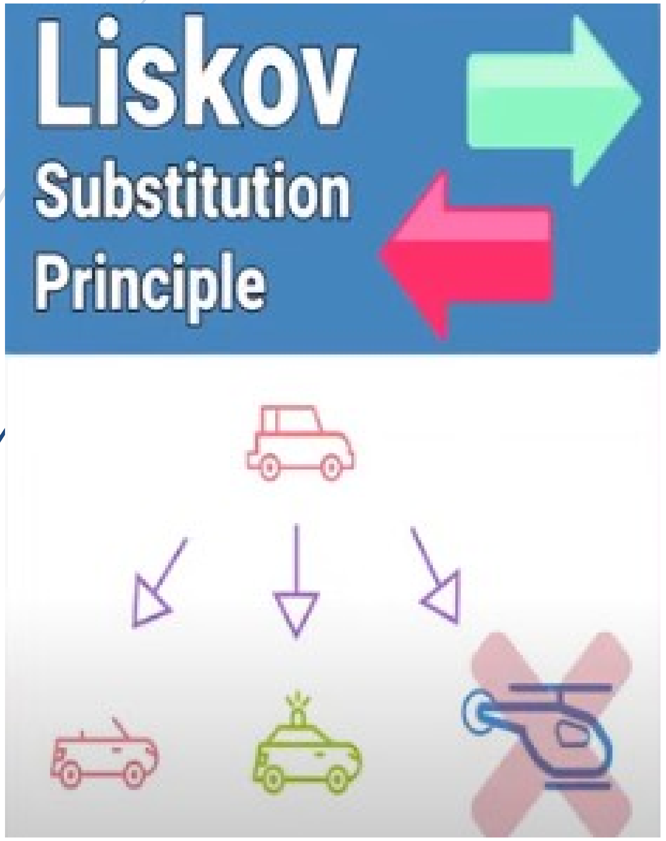
Strategy Design Pattern (Example)

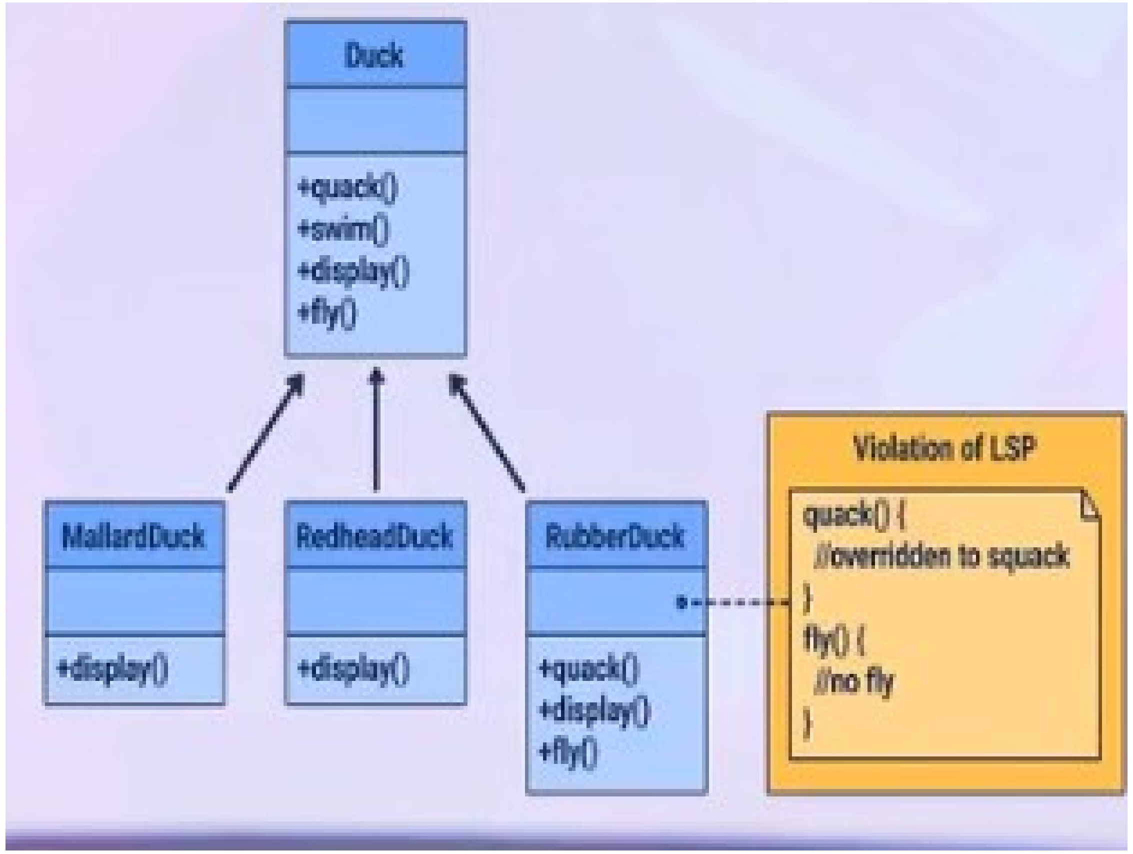
Add fly()

- What about rubber duck or wooden duck?



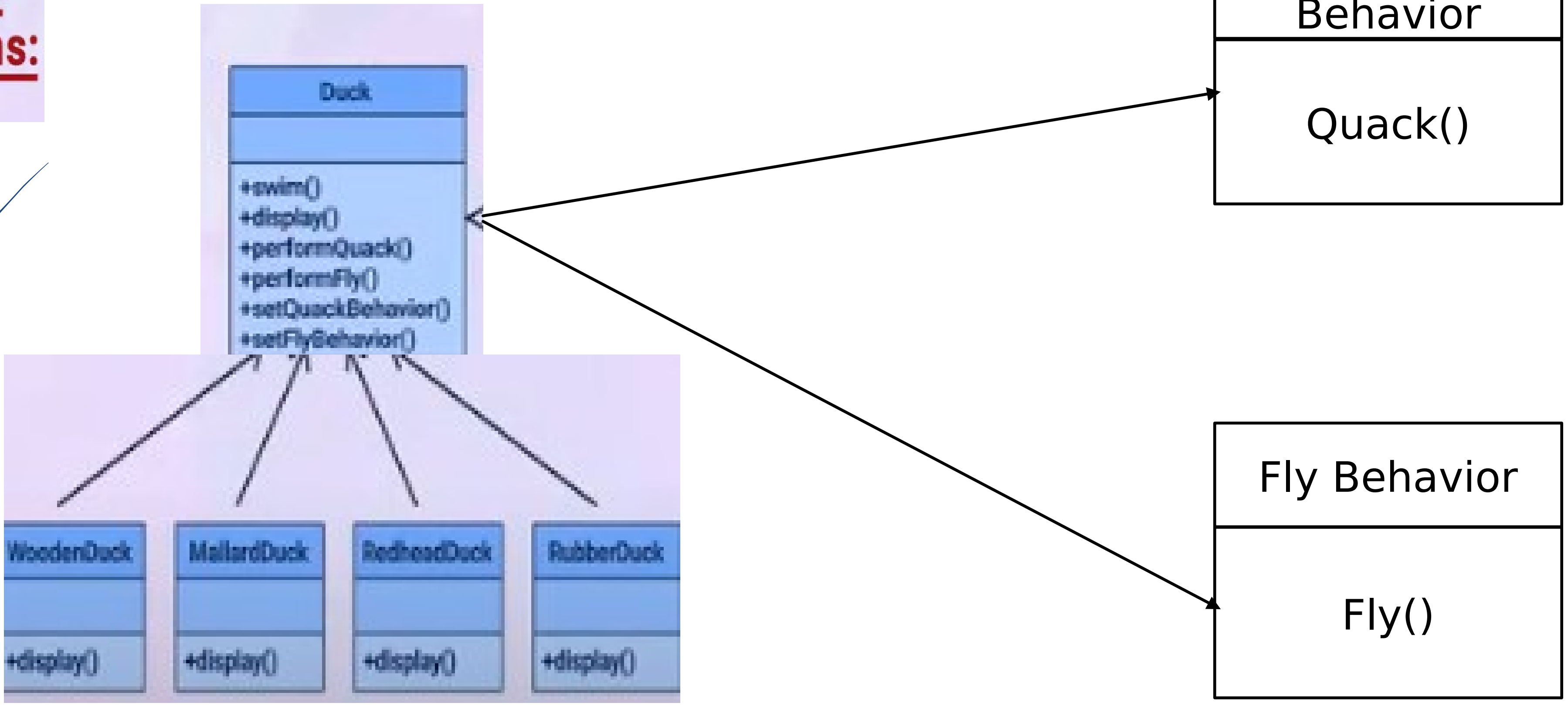
Strategy Design Pattern (Example)



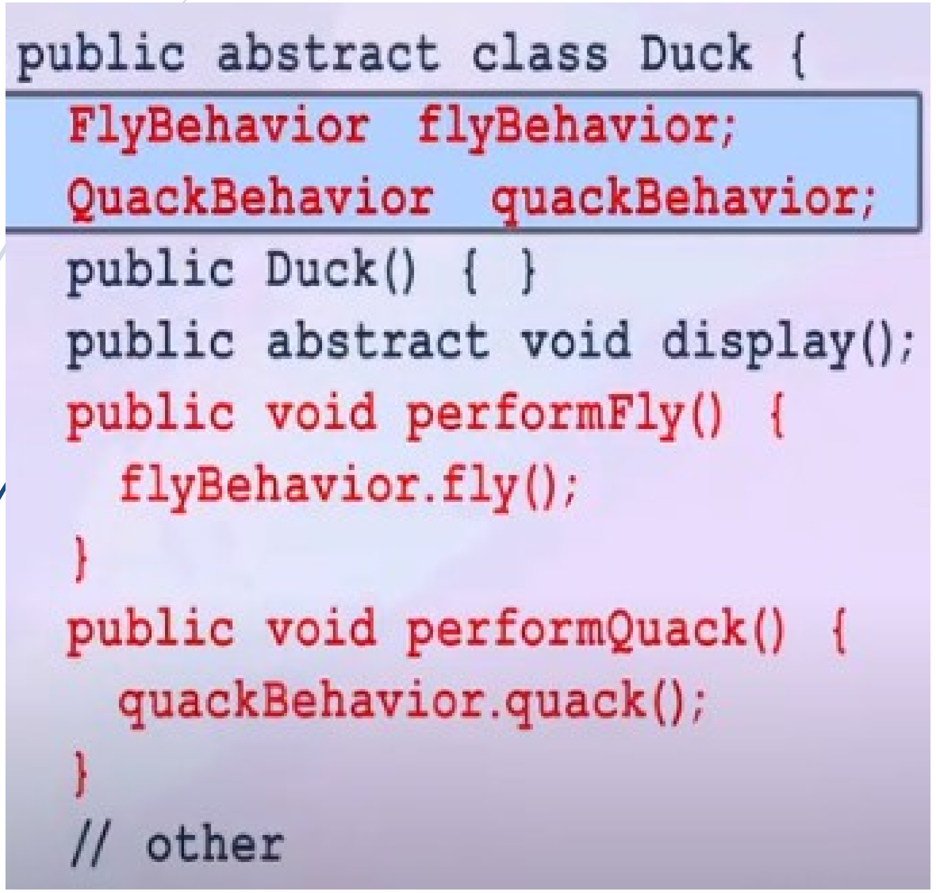


Solution

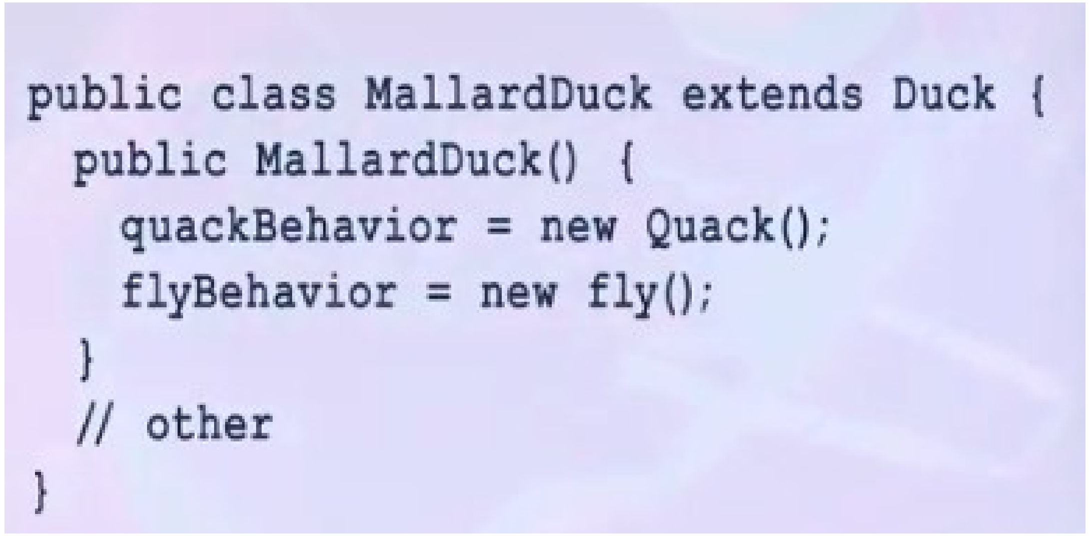
Associate Algorithms:



solution



```
public abstract class Duck {
  FlyBehavior  flyBehavior;
  QuackBehavior  quackBehavior;
  public Duck() { }
  public abstract void display();
  public void performFly() {
    flyBehavior.fly();
  }
  public void performQuack() {
    quackBehavior.quack();
  }
  // other
```



```
public class MallardDuck extends Duck {
  public MallardDuck() {
    quackBehavior = new Quack();
    flyBehavior = new fly();
  }
  // other
}
```

State Design Pattern

(Behavioral pattern)

State Design Pattern

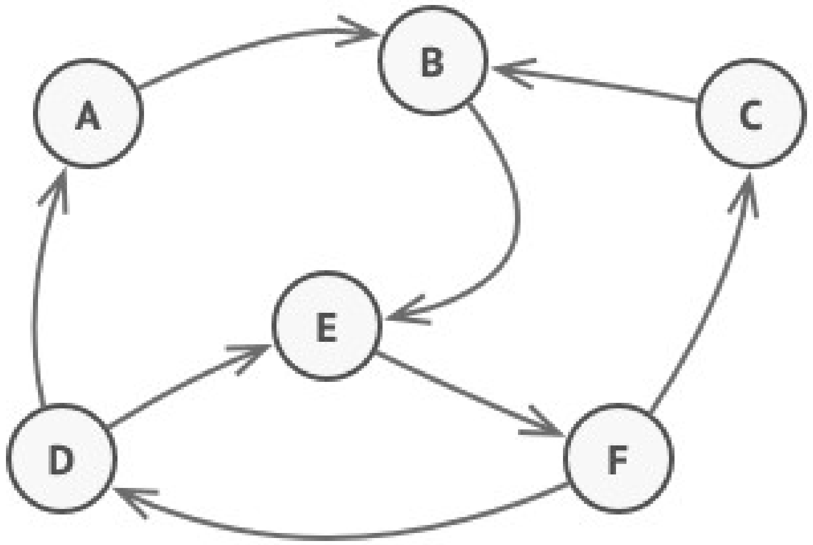
State pattern allows an object to alter its behavior when its internal state changes.

This pattern allows to define state-specific behaviors in separate classes.

A benefit of this pattern is that new states and thus new behaviors can be added without changing our main class.

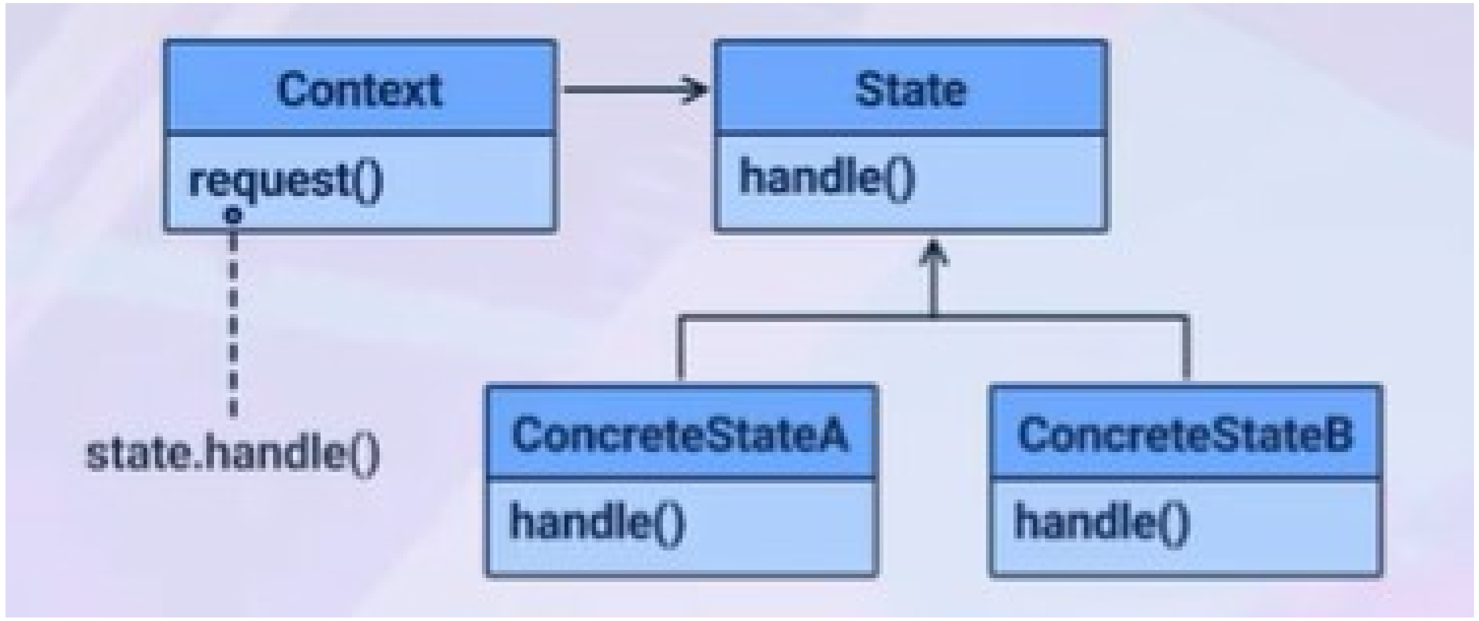
State Design Pattern

The State pattern is closely related to the concept of a **Finite-State Machine**.

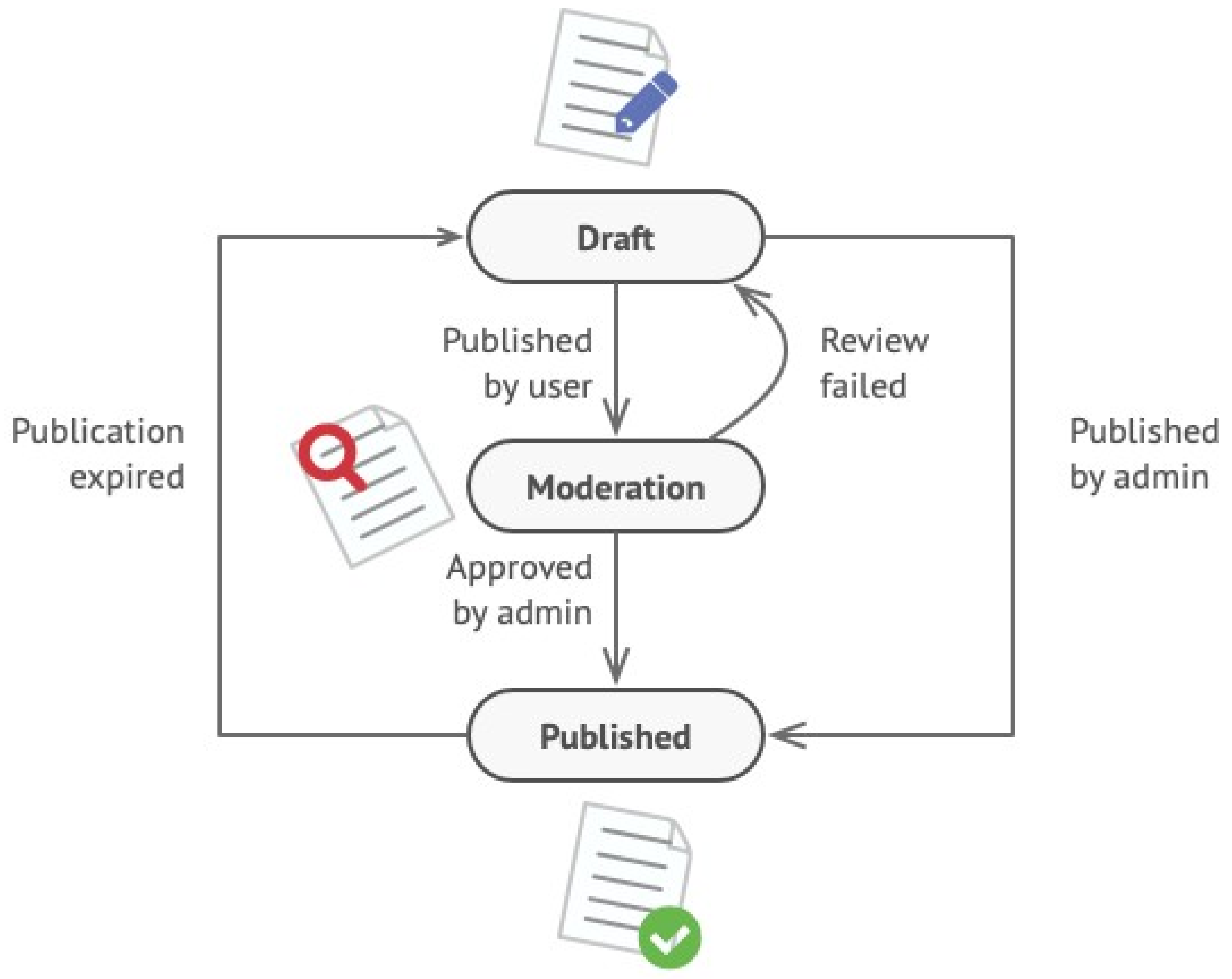


State Design Pattern

State Design Pattern Structure



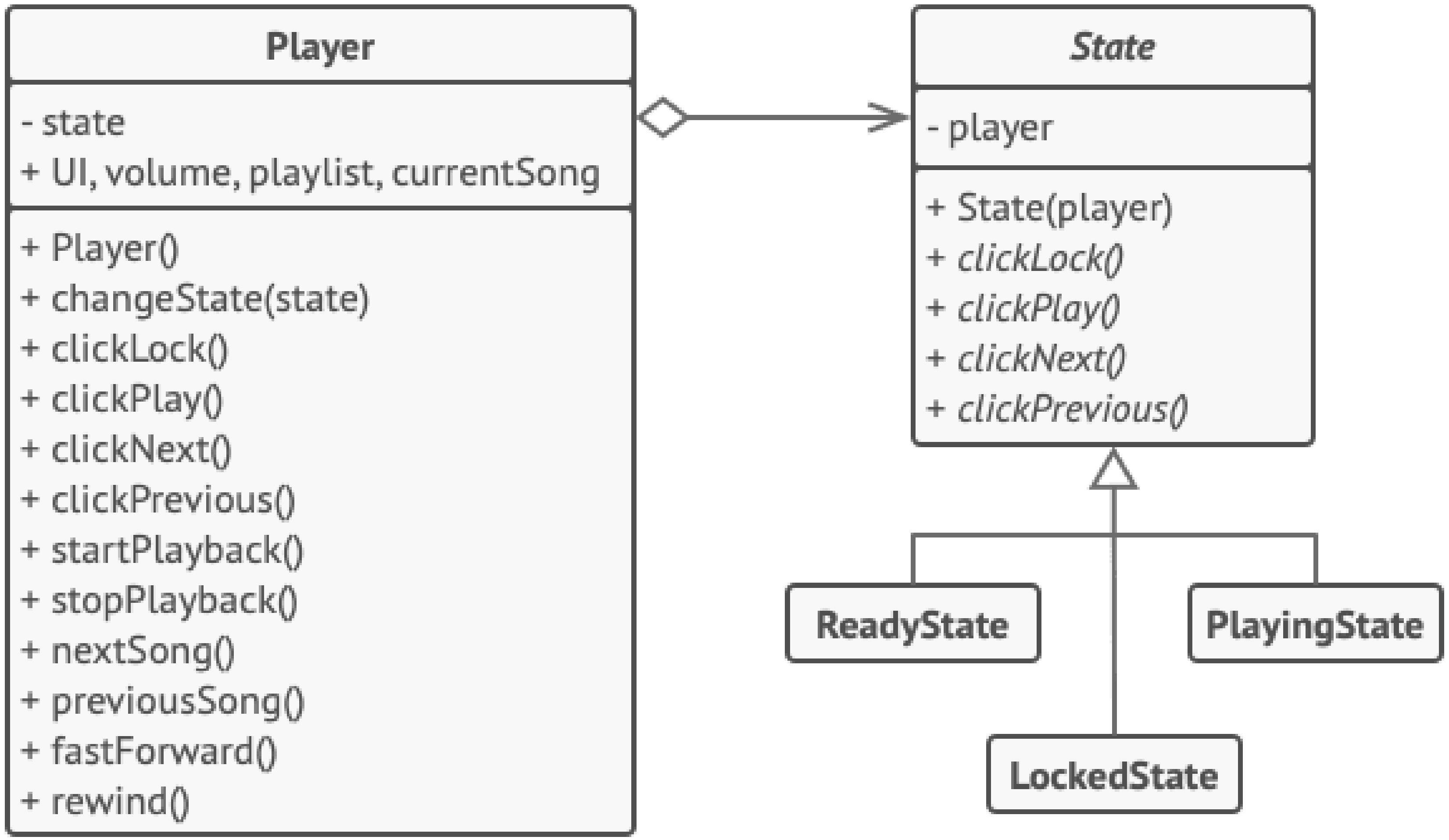
State Design Pattern



State Design Pattern

```
class Document is
switch (state)
        "draft":
            state = "moderation"
            break
        "moderation":
            if (currentUser.role == 'admin')
                state = "published"
            break
        "published":
            // Do nothing.
            break
    // ...
```

State Design Pattern



Lets Code

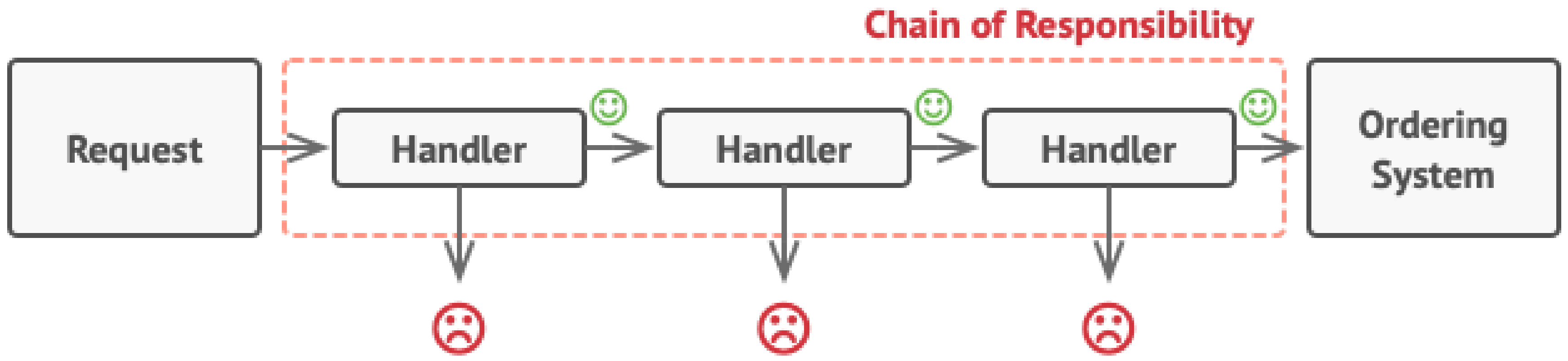
Chain of Responsibility

(Behavioral pattern)

Chain of Responsibility Design Pattern

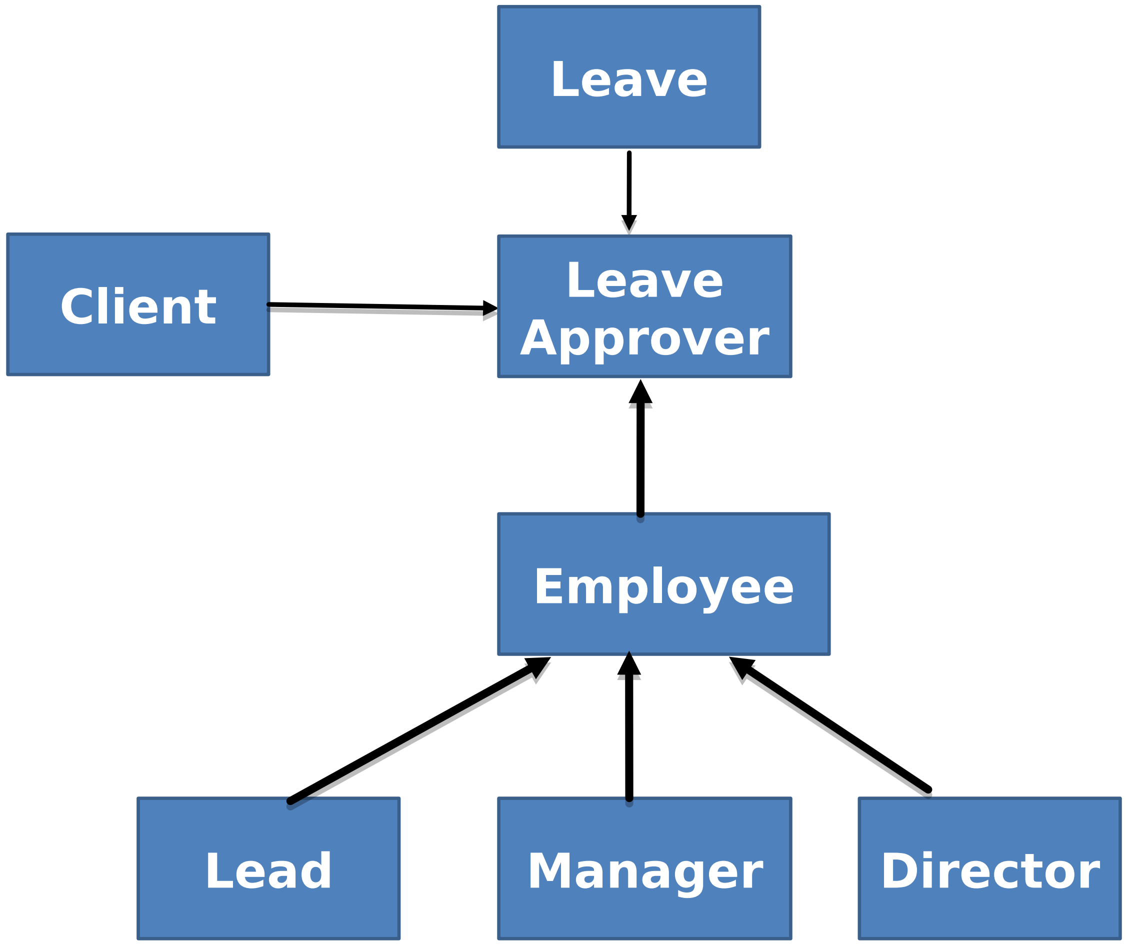
Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers.

Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.



Chain of Responsibility Design Pattern

Diagram of Leave Processing Application



Lets Code

HAVE A GOOD DAY!