Design patterns

Vietnam-Korea University of Information and Communication Technology,
The University of Danang

Introduction

- In many fields, design of systems is based on pre-built patterns
- Examples
  - Electronic circuits are usually designed by assembling other components (such as power supplies, filters, buses, etc.)
  - Designing buildings can be assembled from existing components…

Introduction

- A software **design pattern** is an organization of software components, specifically **classes** or **objects**, that provides a **common solution** (**template**) to a **problem**
- **Benefits** of design patterns
  - The designer's **experiences are reused**
    - Code reuse, high maintainability
  - **Common problems** will be **solved quickly** thanks to the **available solutions**
    - Reducing cost

Introduction

- Design patterns are proposed by
  - **Gamma, Helm, Johnson, and Vlissides**
  - The book "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley
  - Published in 1994
- **23 design patterns** for object-oriented design
- 23 design patterns proposed by 4 people, called "**Gang of Four**" or GoF
- "Description of **communicating objects and classes** that are **customized to solve** a **general design problem** in a **particular context**" - Gamma, Helm, Johnson, and Vlissides

What is a design pattern?

- A design pattern consists of
  - **Pattern name**
  - **Intent**
    - Objective
  - **Problem**
    - When to apply design pattern?
    - Problem, context, conditions for application
  - **Solution**
    - Not a specific solution but a template that can be customized
  - **Consequences**
    - Describing the advantages and disadvantages of using design patterns

Classification

- Design patterns are divided into three categories according to the **purpose of use**
  - **Creational patterns** deal with object creation problems
  - **Structural patterns** relate to the organization of classes/objects
  - **Behavioral patterns** describe interactions between objects/classes
- Design patterns are divided into two categories according to the **scope of application**
  - **Classes**: describe the relationship between classes
  - **Objects**: describe interactions between objects

Classification

		Purpose		
		Creational (5)	**Structural (7)**	**Behavioral (11)**
Scope	**Class (4)**	Factory Method	Adapter (class)	Interpreter Template Method
	Object (19)	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational patterns

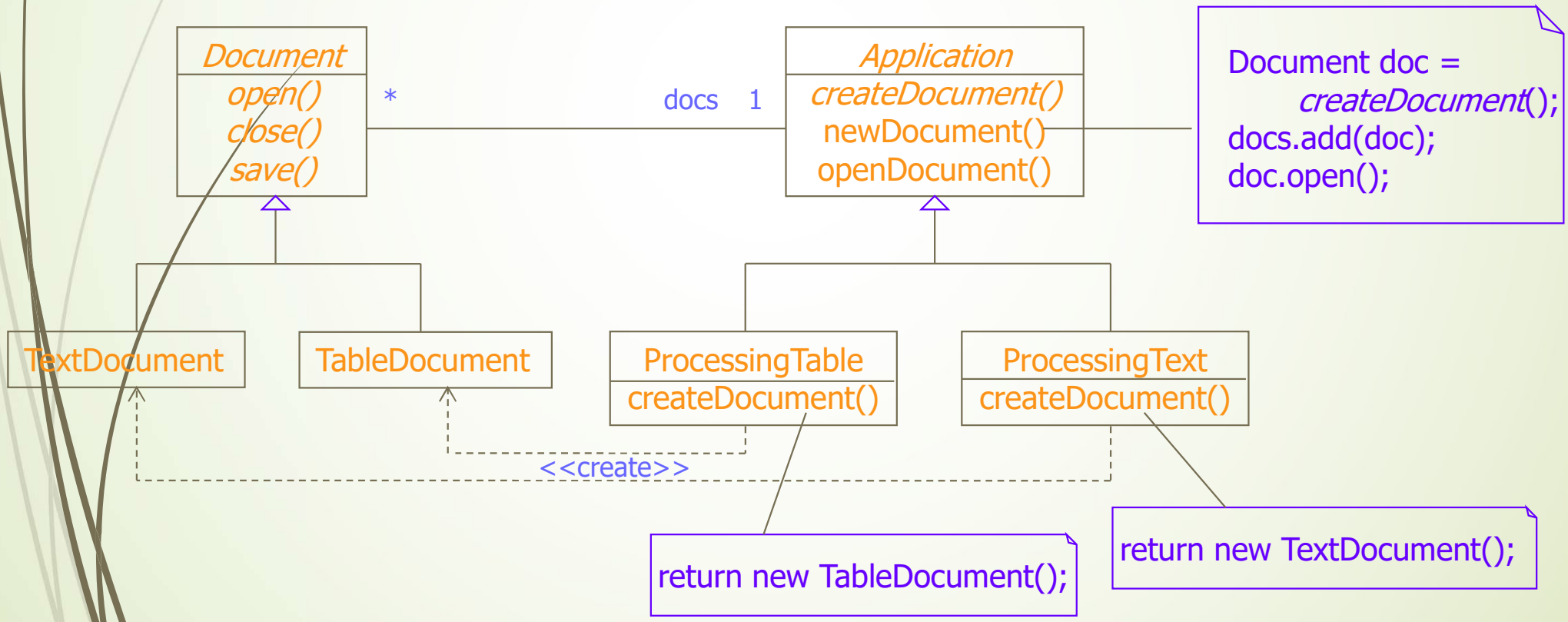
- 5 patterns
  - **Factory Method**
  - Abstract Factory
  - **Builder**
  - Prototype
  - **Singleton**

Factory Method

- Motivation
  - We want to develop a set of office programs, such as word processing (text), spreadsheets (tables)... They share an interface. We have defined:
    - An abstract class ***Application*** implements the common features of the interface
    - An abstract class **Document** groups the properties of documents that can be processed by programs
- Problem
  - Which class can create new objects of the ***Document*** class in the code of the ***Application*** class?

Factory Method

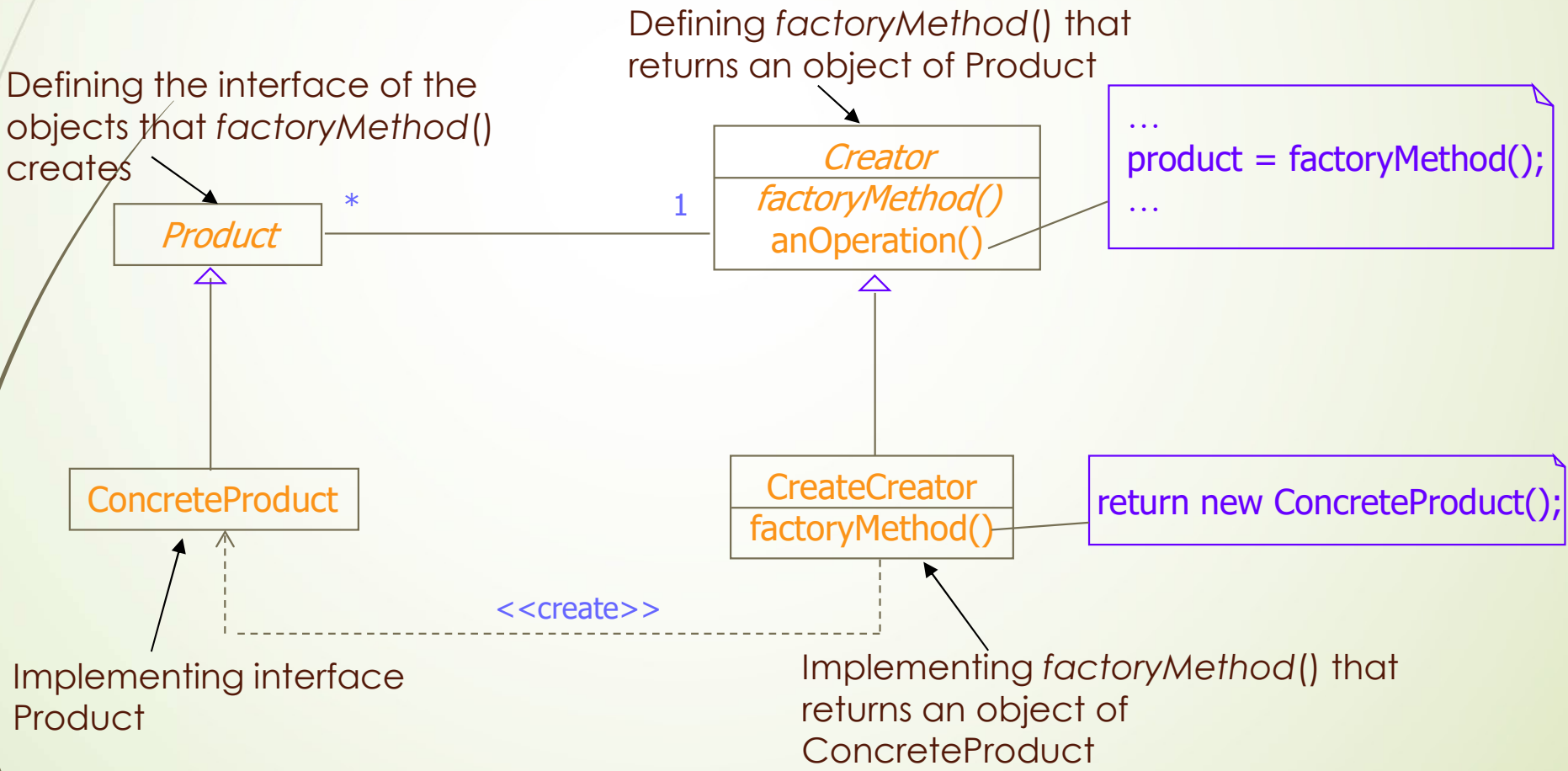
- Solution
  - The subclasses of the ***Application*** class are responsible for creating the ***Document*** objects



Factory Method

- Structure

Intent: Provides an interface for creating objects in a superclass, but let subclasses decide which class to instantiate/create



Builder

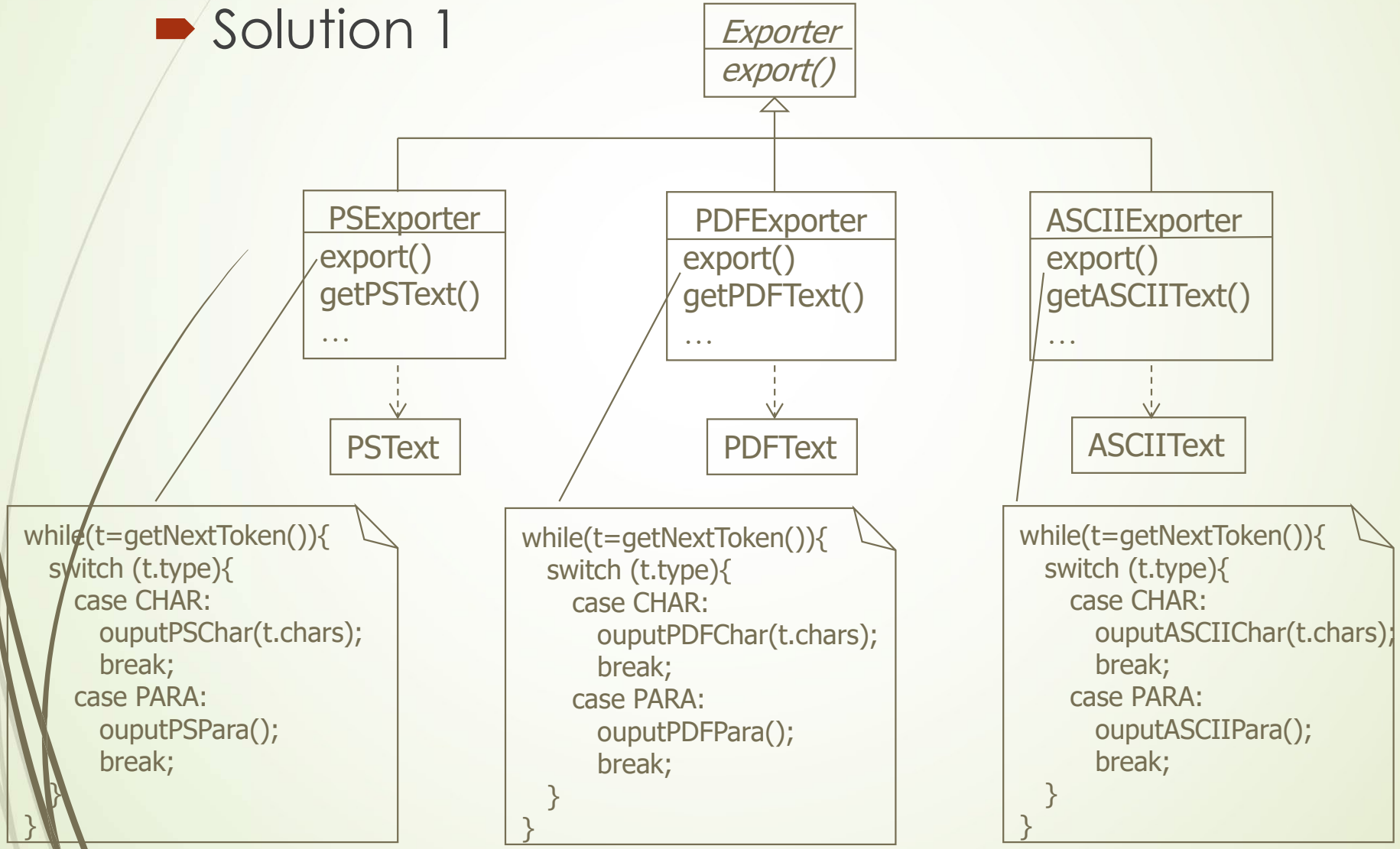
- Motivation
  - We want to develop a text editor where documents can be stored in various formats: HTML, PostScript, PDF, ASCII…
- Problem
  - How to organize the program so that a **new format can be added easily**?

Builder

Limitations:

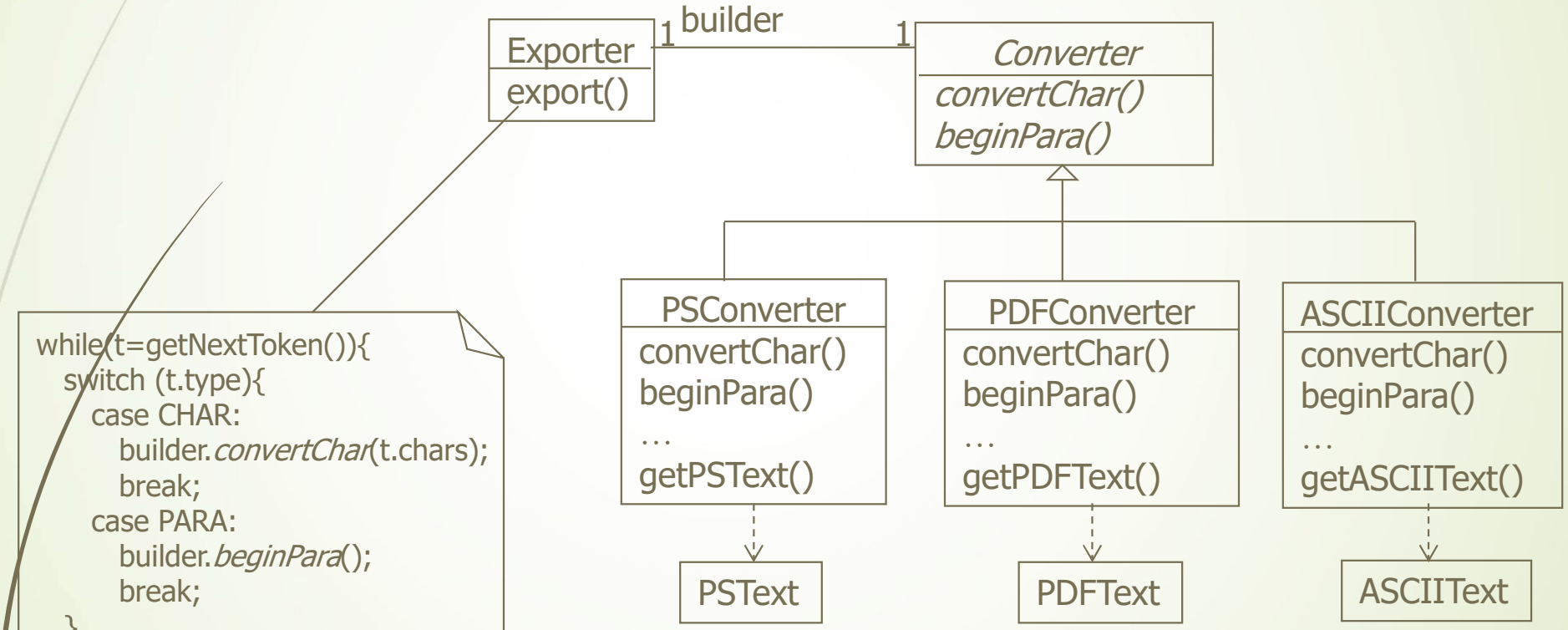
- Code redundancy
- When adding a new document format, the entire program code that outputs the document must be rewritten

- Solution 1



Builder

- Solution 2



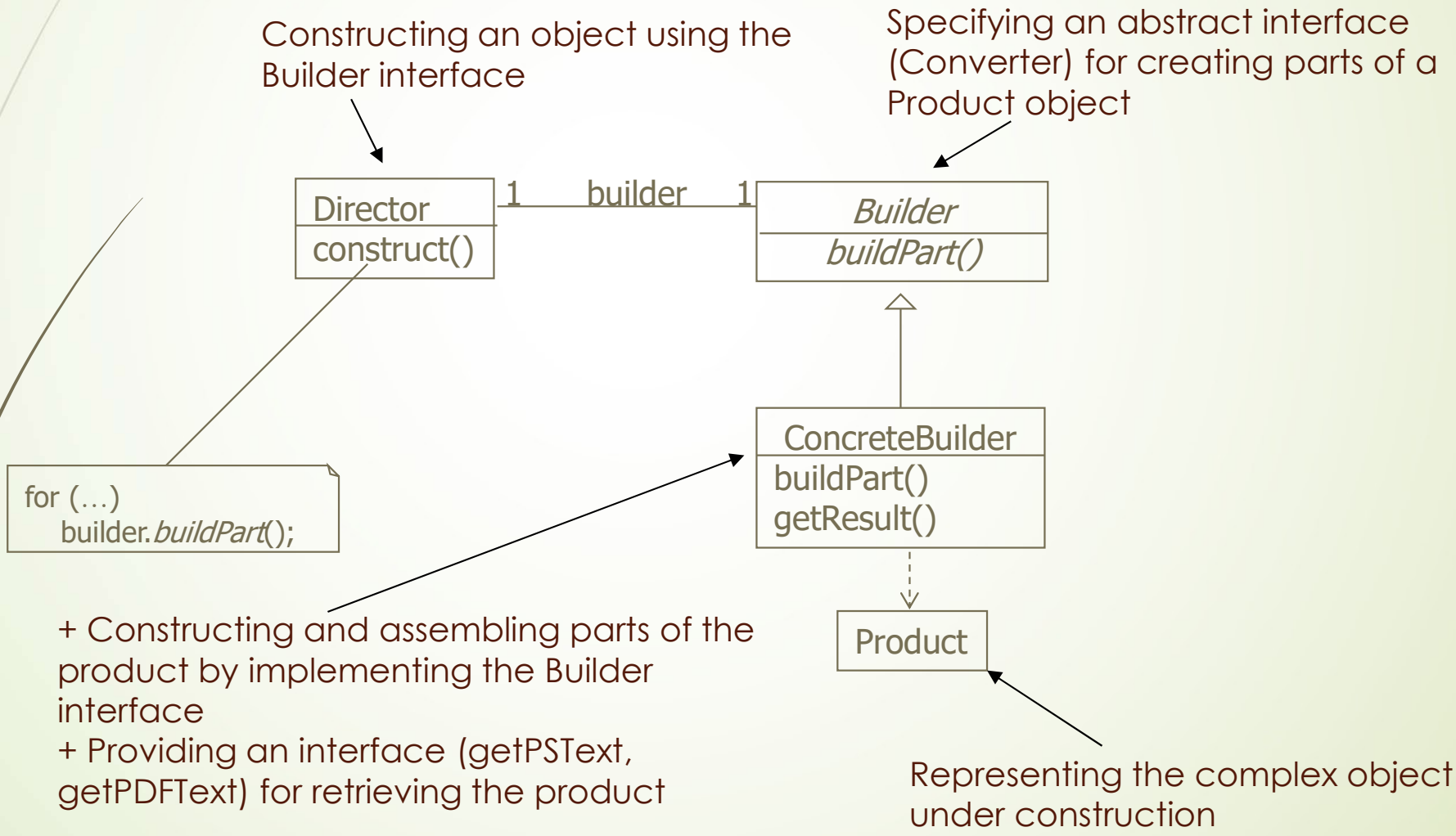
Benefits:

+ Saving code
+ Adding easily new format types

Builder

- Structure

Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations

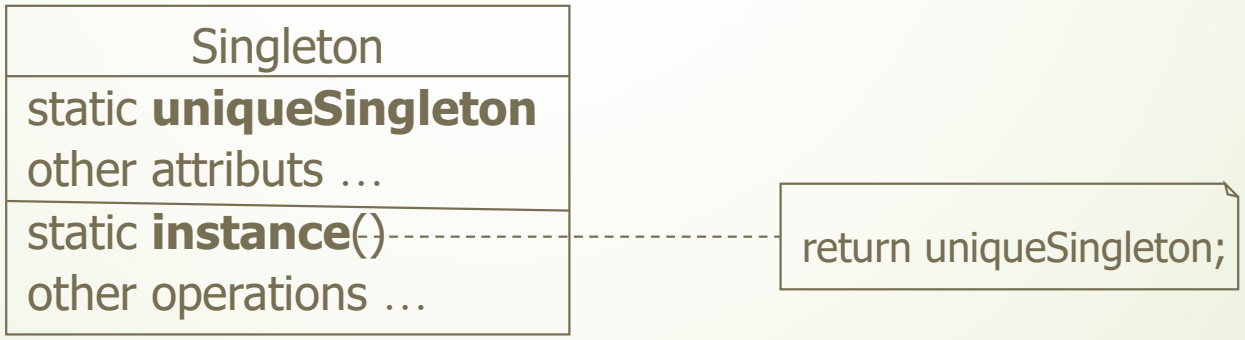


Singleton

- Intent
  - Ensuring a **class only has one object**, and provide a global point of access to it
- Motivation
  - We want to develop an application and resource management system on a computer. Some of the objects on the system must be unique such as printer queue, application manager, etc., and these objects are used by a collection of applications.
- Problem
  - How to organize program code so that **an object is unique**?

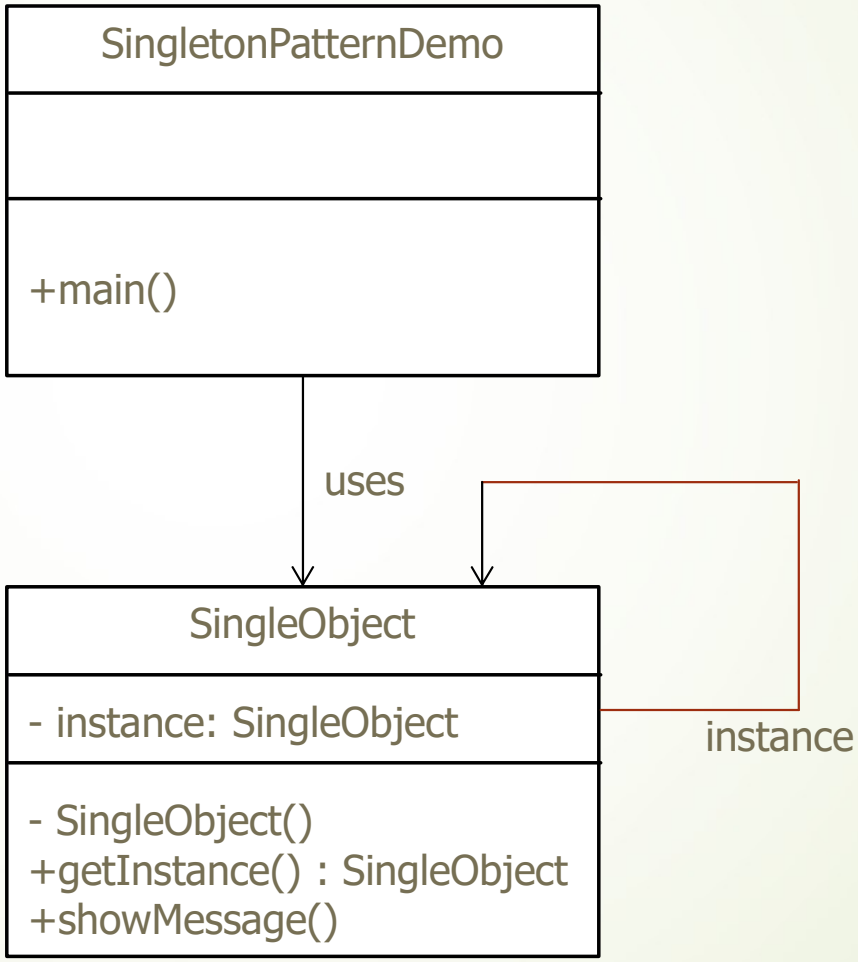
Singleton

- Bad solution
  - Using global variables to store objects
    - Limitation: different objects may be assigned to the global variable
- Good solution
  - There is only one class that can create an object and access that unique object (singleton)



Singleton

- Example with code



Singleton

- Create a Singleton Class

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Singleton

- Get the only object from the singleton class

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Structural Patterns

- 7 patterns
  - **Adapter**
  - Bridge
  - **Composite**
  - **Decorator**
  - Facade
  - Flyweight
  - Proxy

Adapter

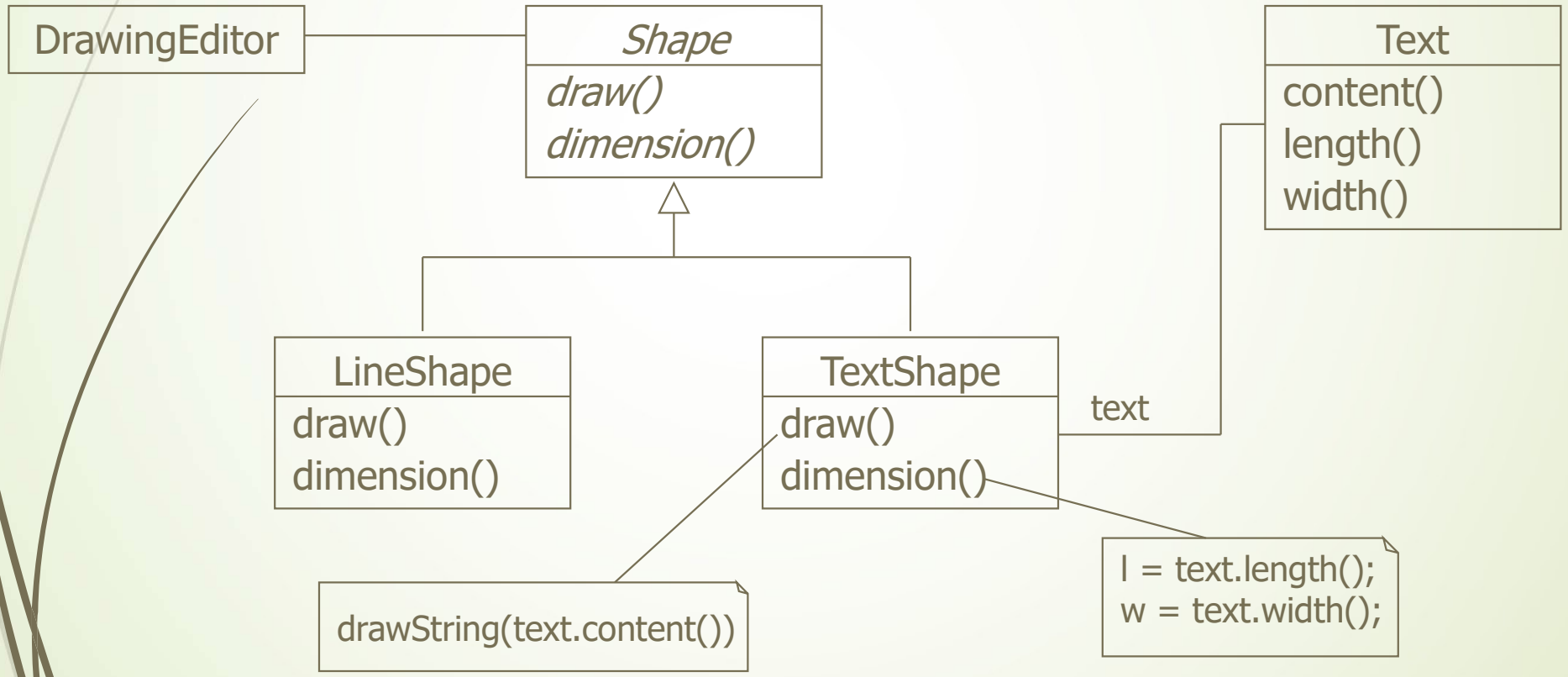
- Motivation
  - We want to develop a graphic editing tool (draw lines, polygons, text strings, ...). Interfaces for graphic objects are defined by the abstract class *Shape*. Each specific type of graphic object is defined as a subclass of *Shape*, such as *LineShape*, *PolygonShape*, *TextShape*, etc.
- Problem
  - For the *TextShape* class, we want to use operations on text that are already implemented for the *Text* class in another application.

Adapter

- Solutions
  - Defining the *TextShape* class so that it adapts the interfaces of the *Text* class to the *Shape class*. This can be done in two ways:
    - Solution 1: *TextShape* contains an object of *Text* and inherits *Shape* – Adapter (object)
    - Solution 2: *TextShape* inherits *Shape* and the *Text* – Adapter (class)

Adapter

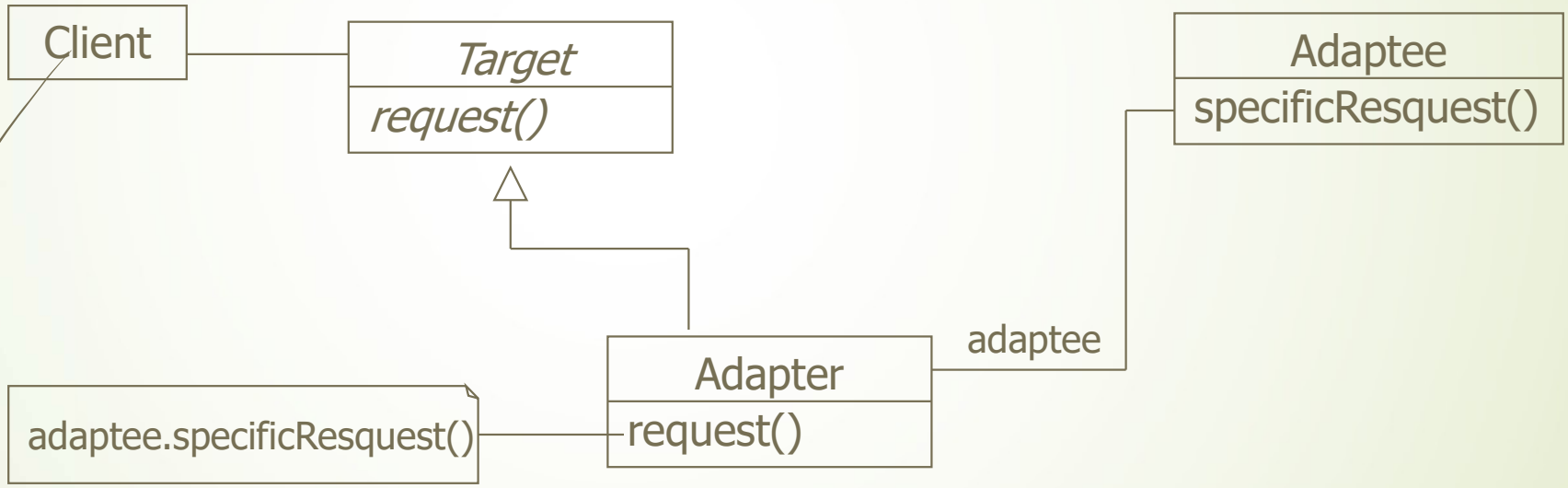
- Adapter (object)



Adapter

Intent: Converting the interface of a class into another expected interface

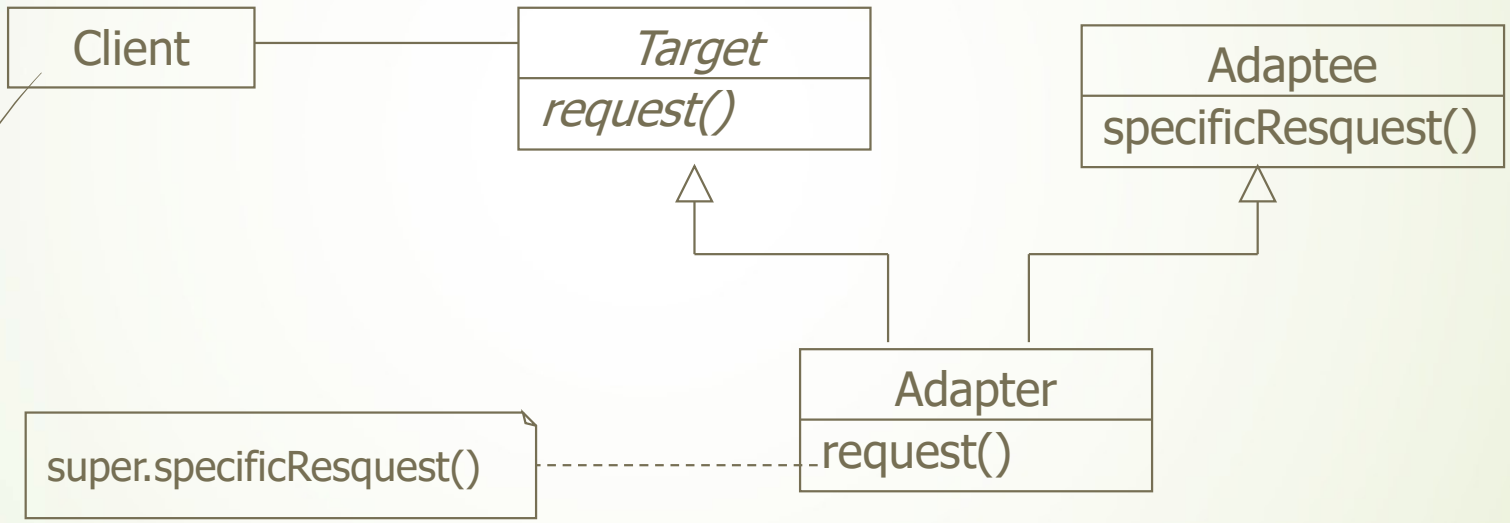
- Adapter (object)
  - contains an object and inherits a class



Adapter

Intent: Converting the interface of a class into another expected interface

- Adapter (class)
  - Multiple inheritance



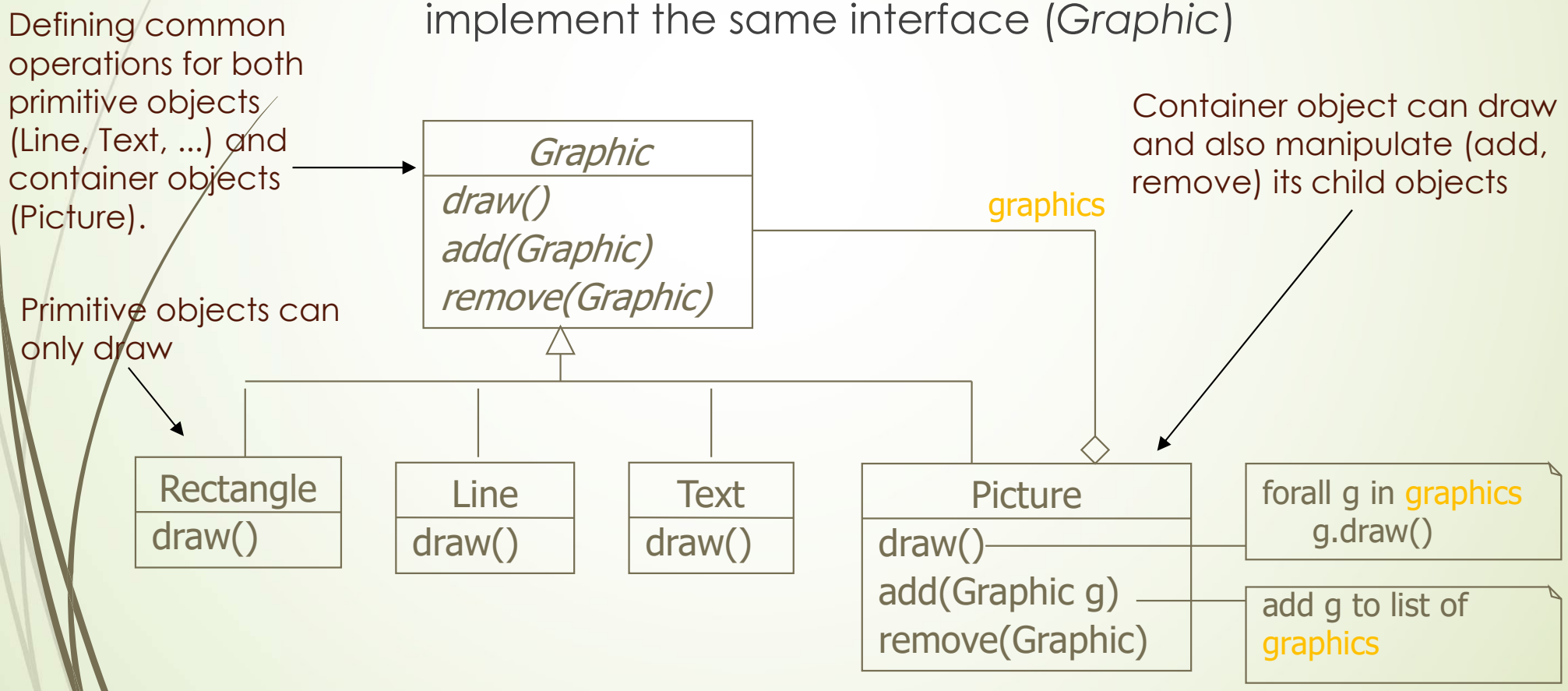
Question: How to apply Adapter (class) to the problem?

Composite

- Motivation
  - We want to develop a graphical editor that allows complex pictures to be built from simple components: simple components are grouped to build larger components, and these components are further grouped to create even larger components…
- Problem
  - In the application, there are two types of objects: the primitive graphic objects (lines, texts, rectangles...) and the container objects that contain them. How to handle these two types of objects in the same way, that is, without having to distinguish them?

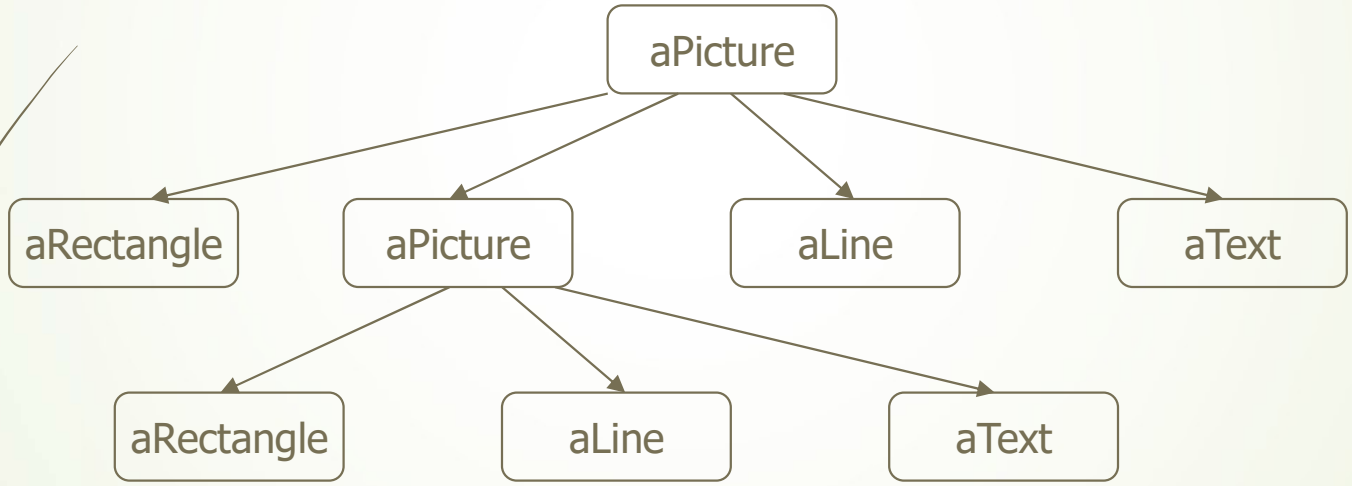
Composite

- Solution
  - Defining classes for the primitive objects (*Line, Text, Rectangle*) and the container object so that they implement the same interface (*Graphic*)



Composite

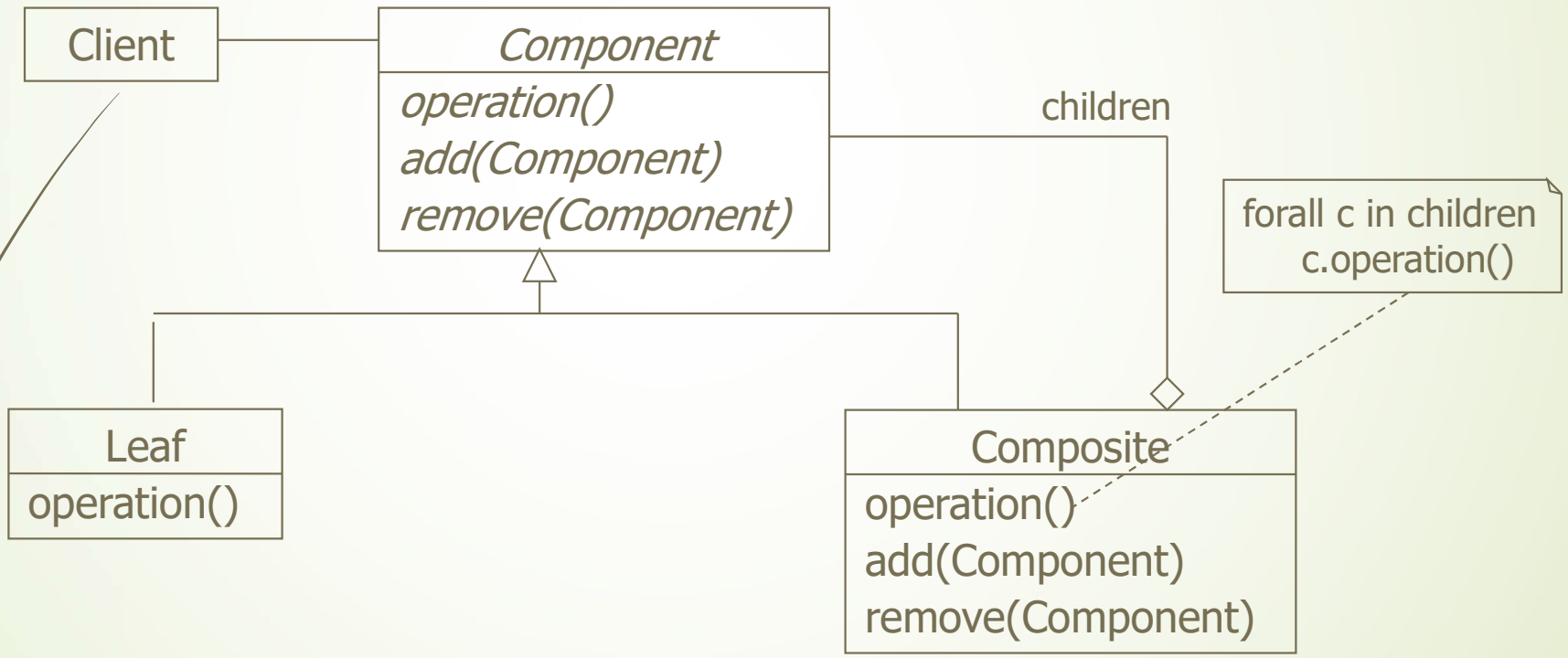
- Example: a typical composite object structure of recursively composed Graphic objects



Composite

- Structure

Intent: Composing objects into tree structures to represent part-whole hierarchies



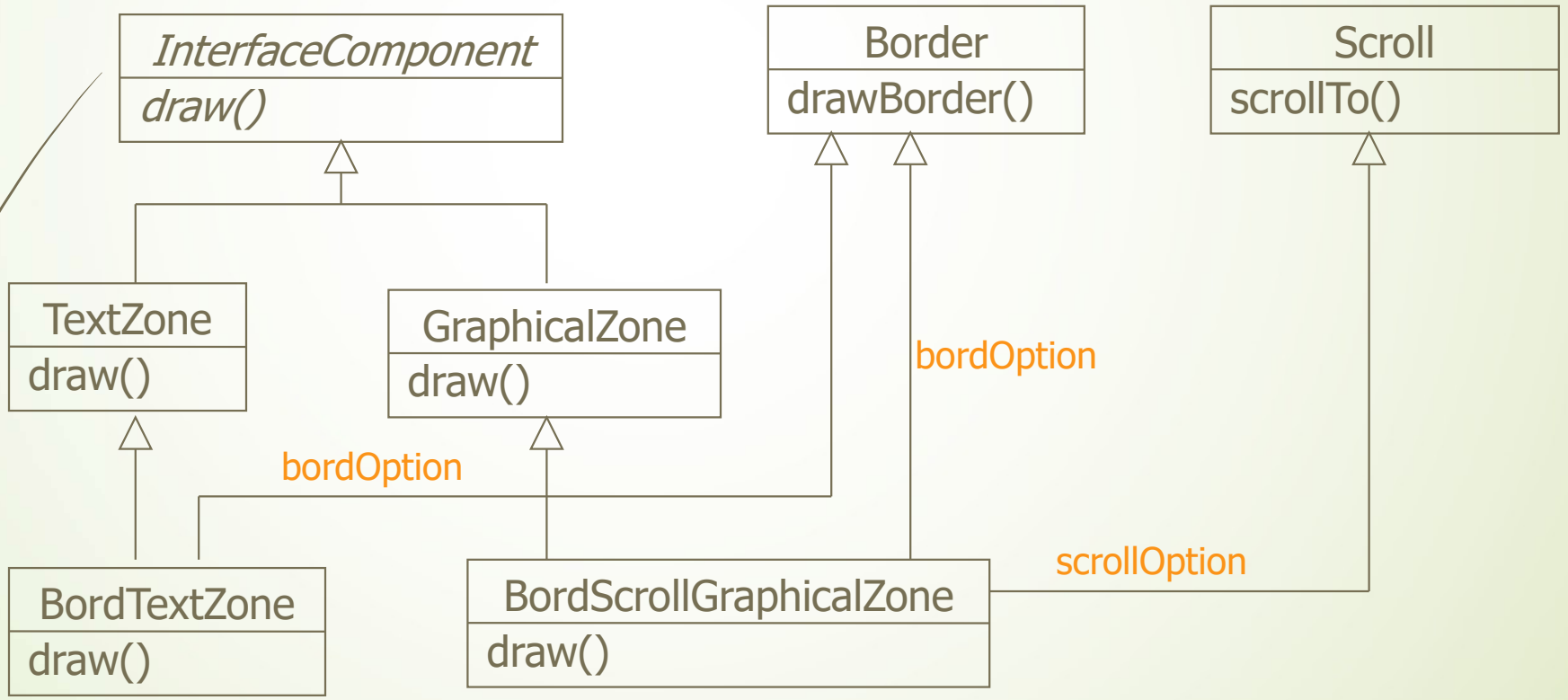
Decorator

- Motivation
  - We want to build a graphical user interface tool that allows the design of window graphical interface elements. Each of these interface elements can have common properties such as scroll bar, border, etc.
- Problem
  - How to effectively implement these common properties?

Decorator

Limitation: Combining a large number of properties complicates the class hierarchy

- Solution 1

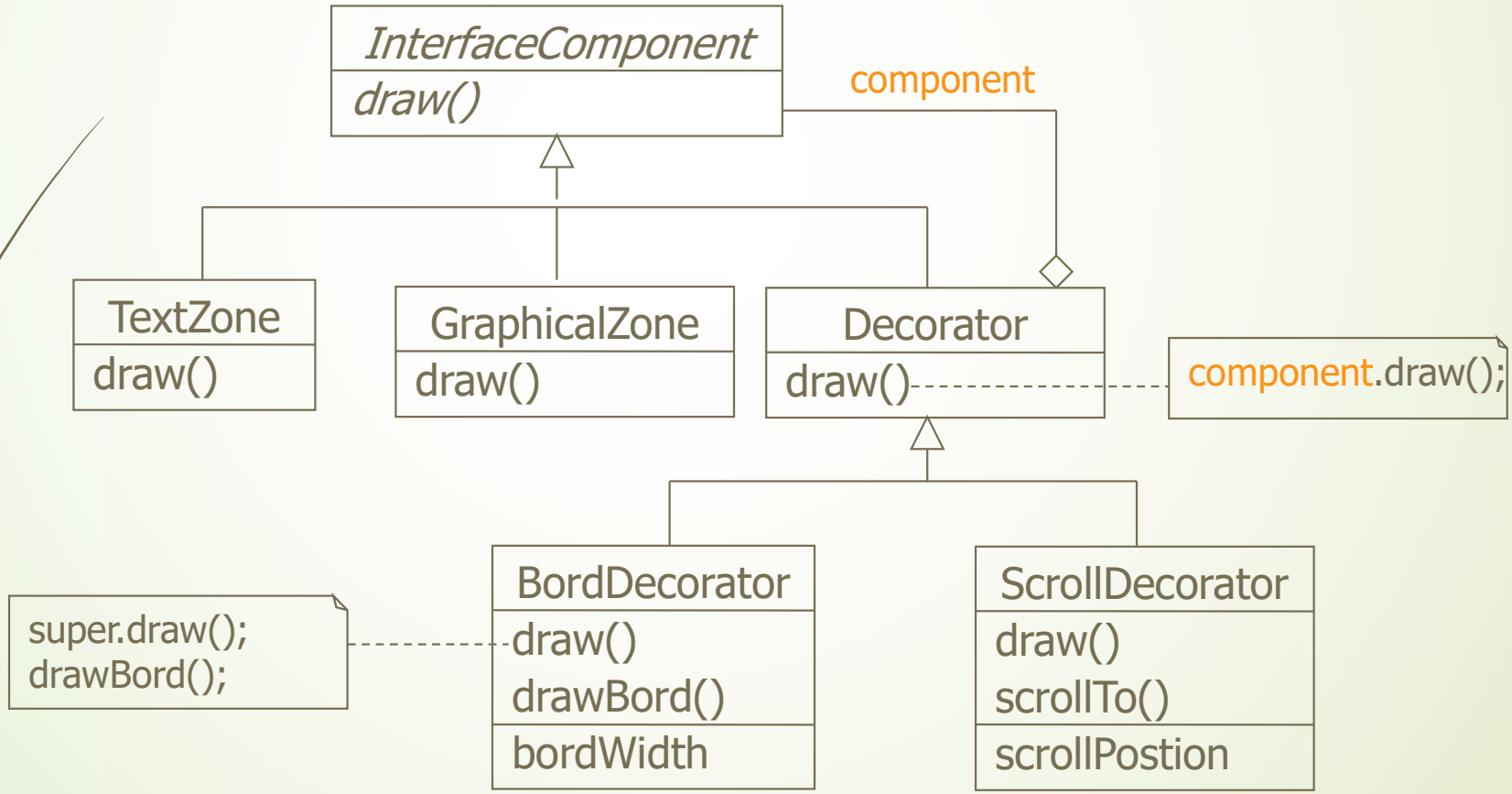


Decorator

Advantages:

+ Common properties can be added more easily
+ The class hierarchy is always simple

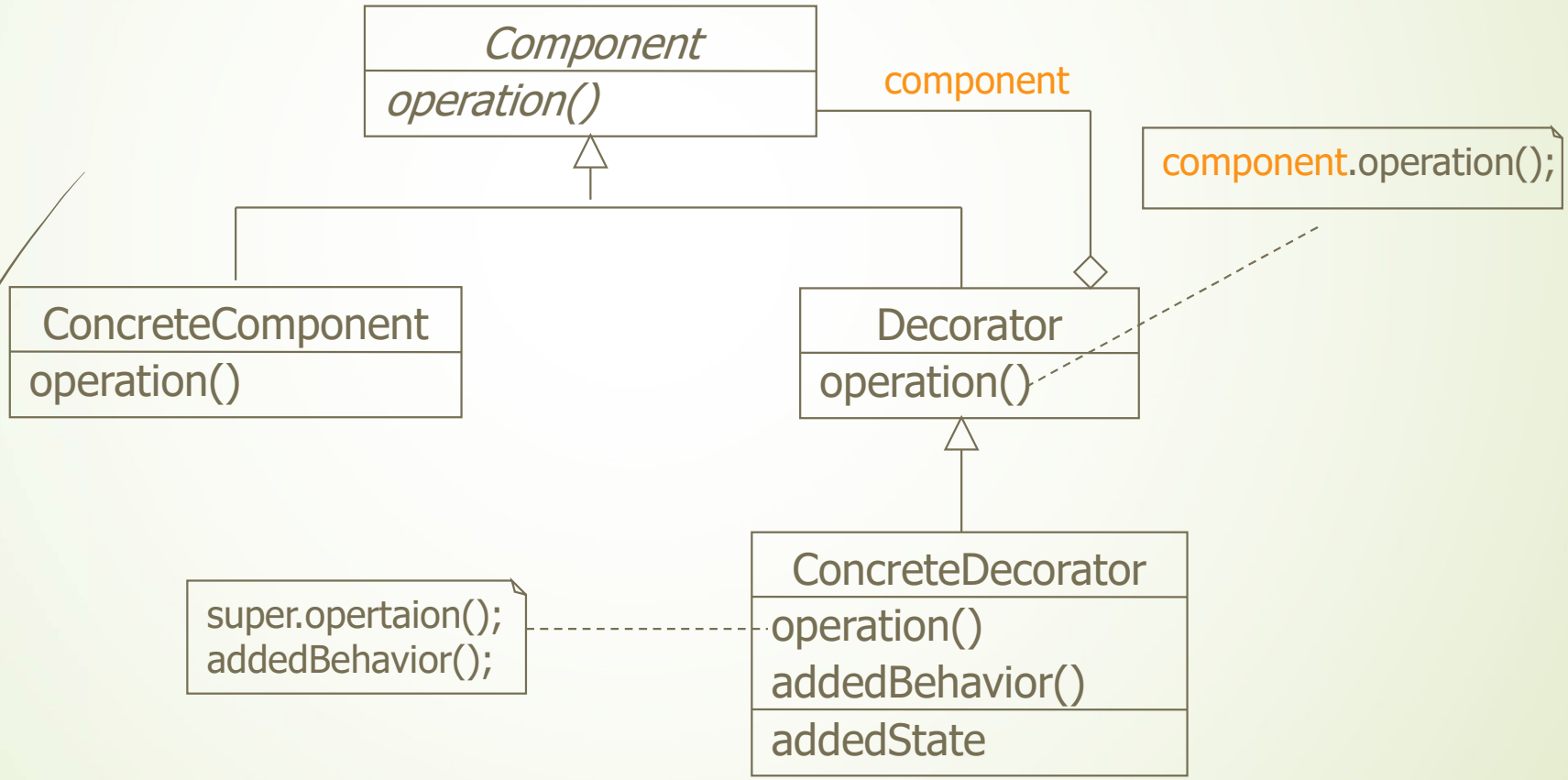
- Solution 2



Decorator

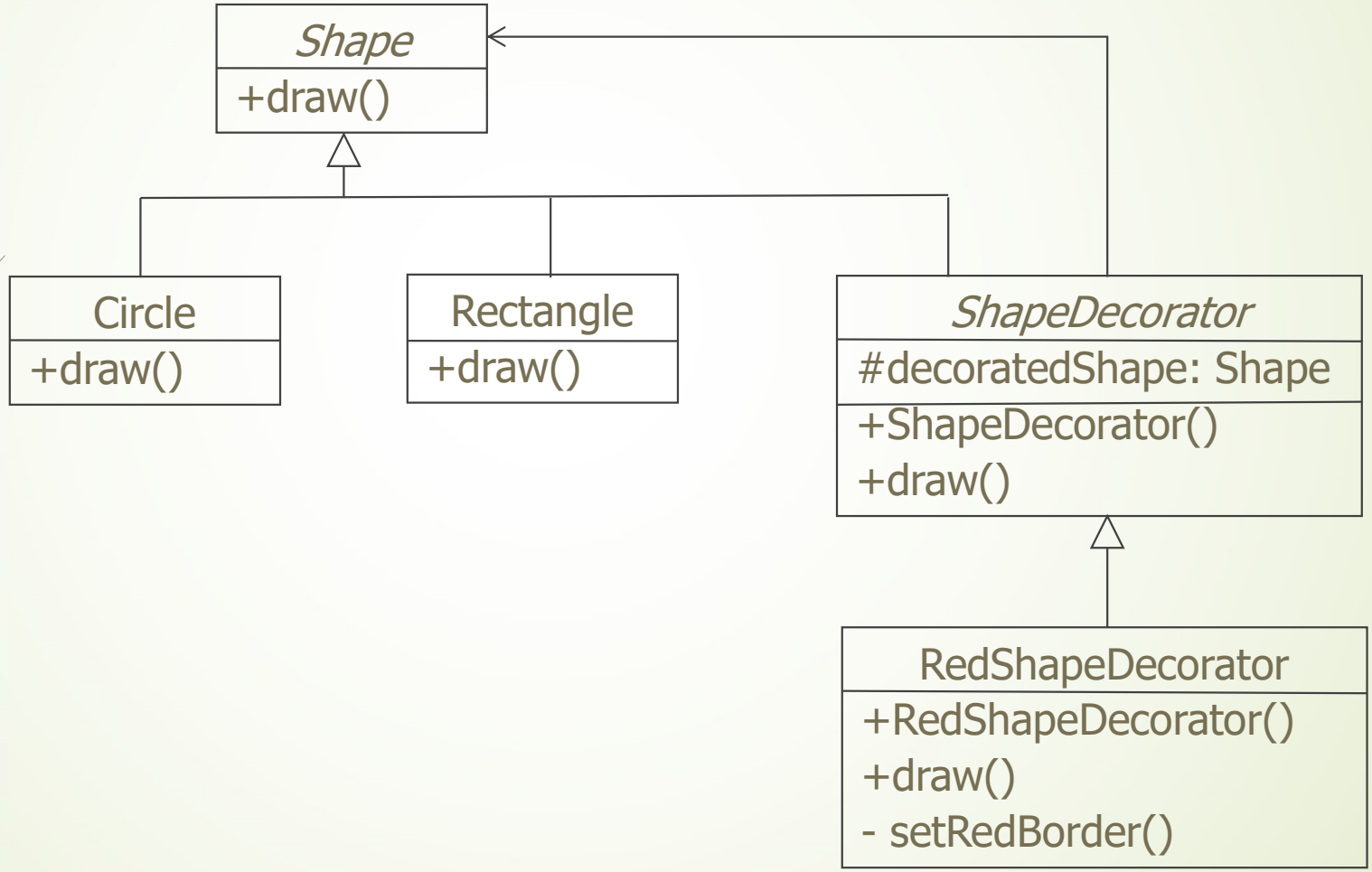
- Structure

Intent: Attaching additional responsibilities to an object dynamically



Decorator

- Example with code



Decorator

- Create a Shape interface

```
public interface Shape {
   void draw();
}
```

- Create classes implementing the Shape interface

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

```
public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Circle");
   }
}
```

Decorator

- Create abstract decorator class implementing the *Shape* interface

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

Decorator

- Create concrete decorator class extending the *ShapeDecorator* class

```
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```

Decorator

- Use the *RedShapeDecorator* to decorate *Shape* objects

```
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

Behavioral Patterns

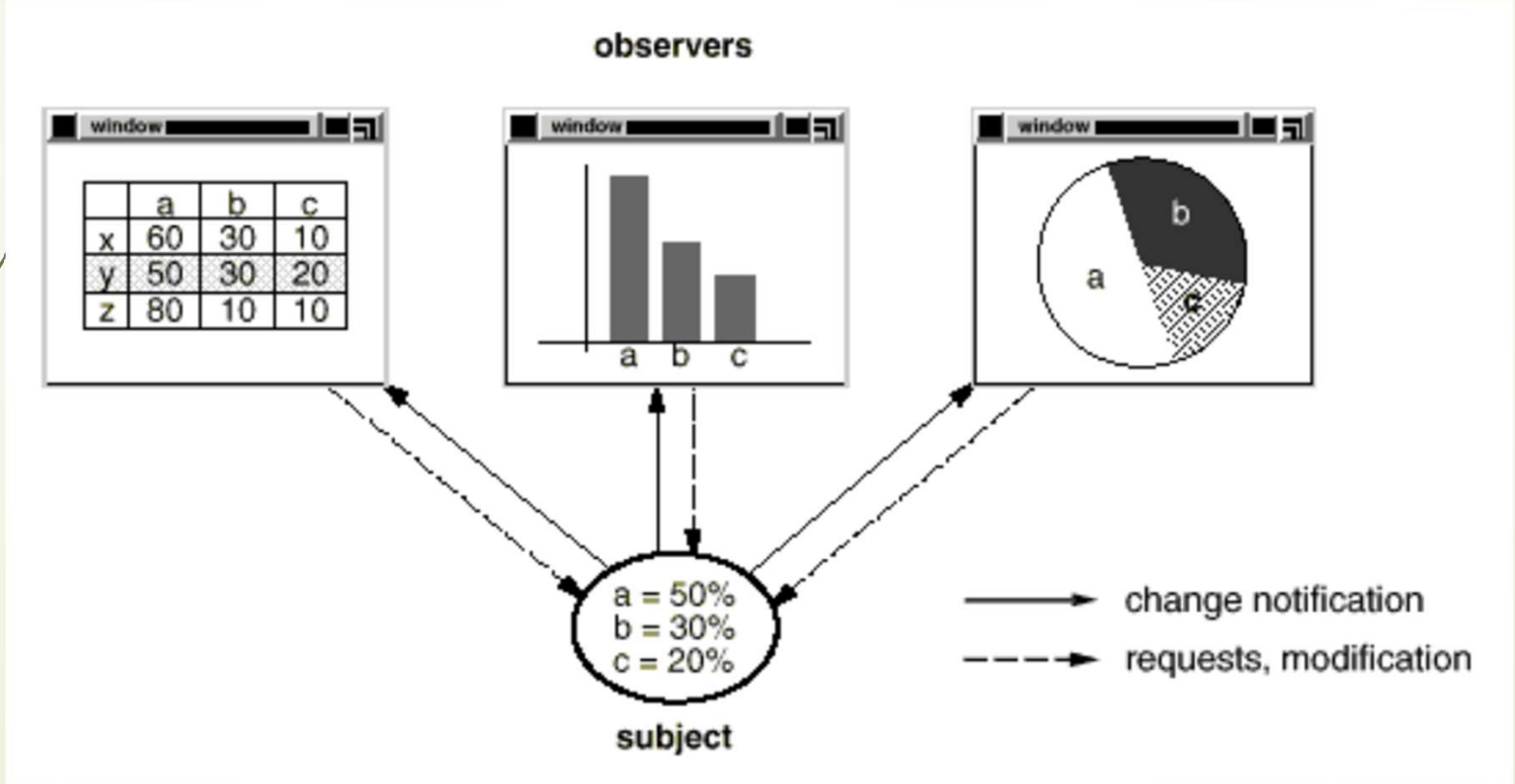
- 11 patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - **Observer**
  - State
  - Strategy
  - **Template Method**
  - Visitor

Observer

- Motivation
  - We want to develop a tool to visually represent data using different types of graphs. The same data can be represented by different types of graphs in different windows.
- Problem
  - When there is a data change in each window, the remaining windows must be changed accordingly

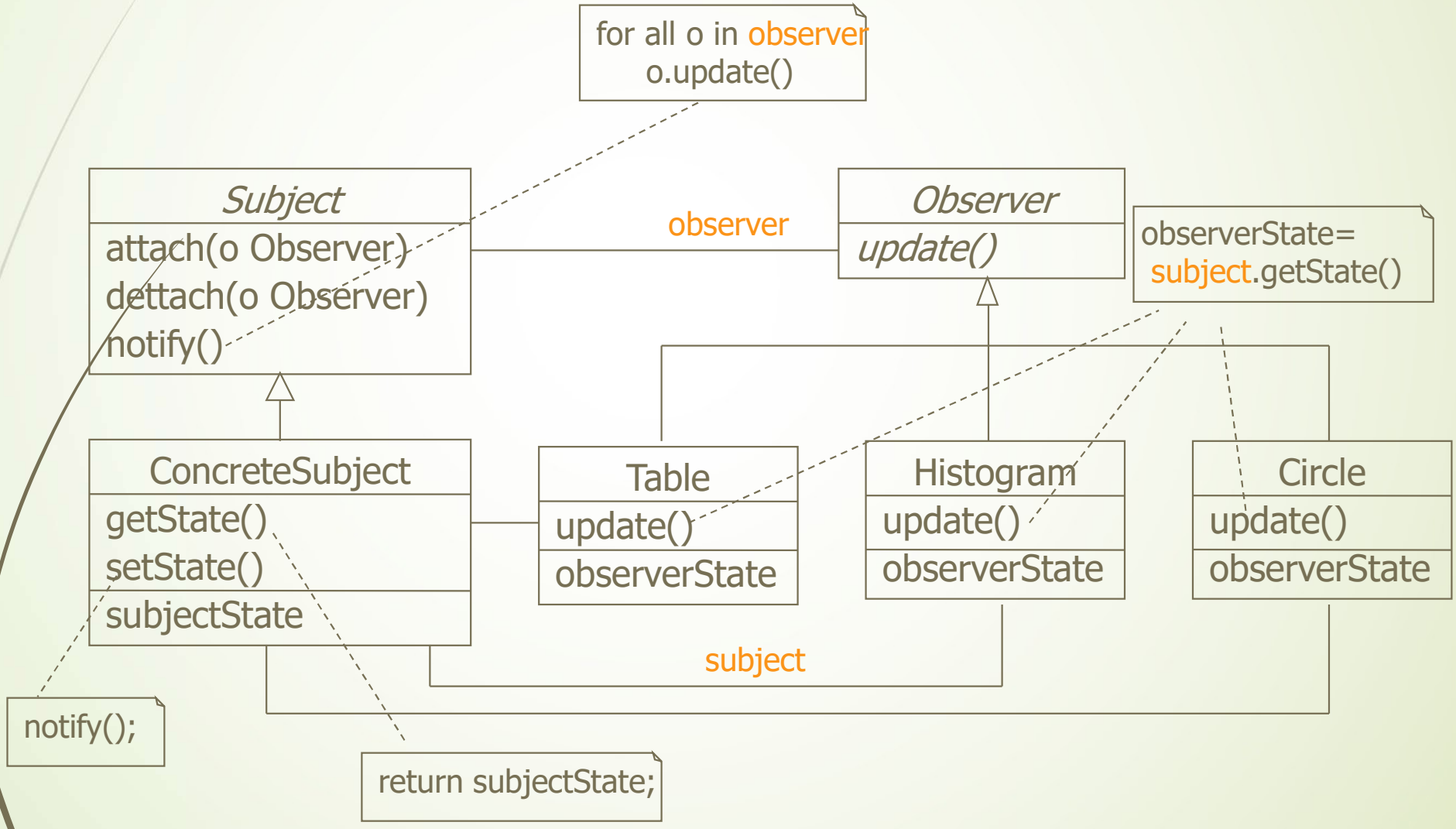
Observer

- The tool will be developed



Observer

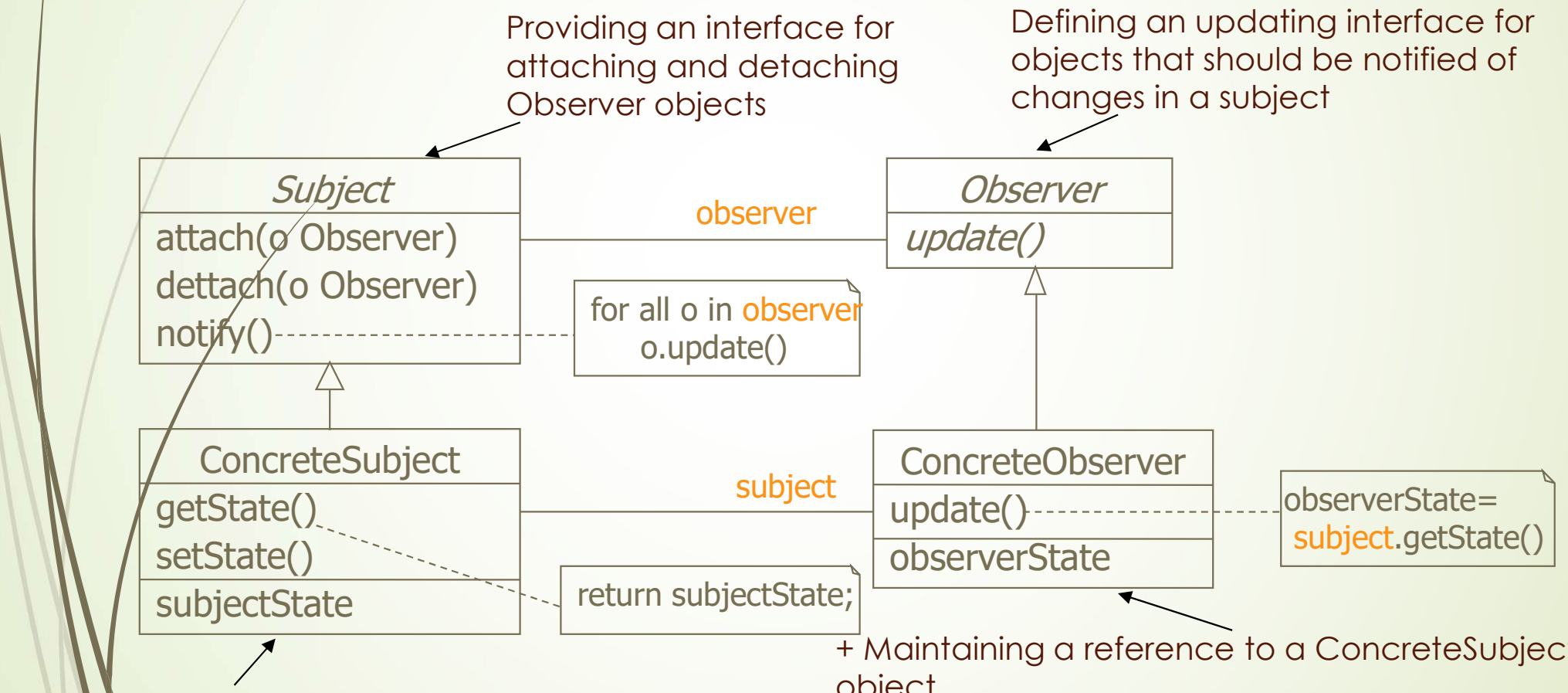
- Solution



Observer

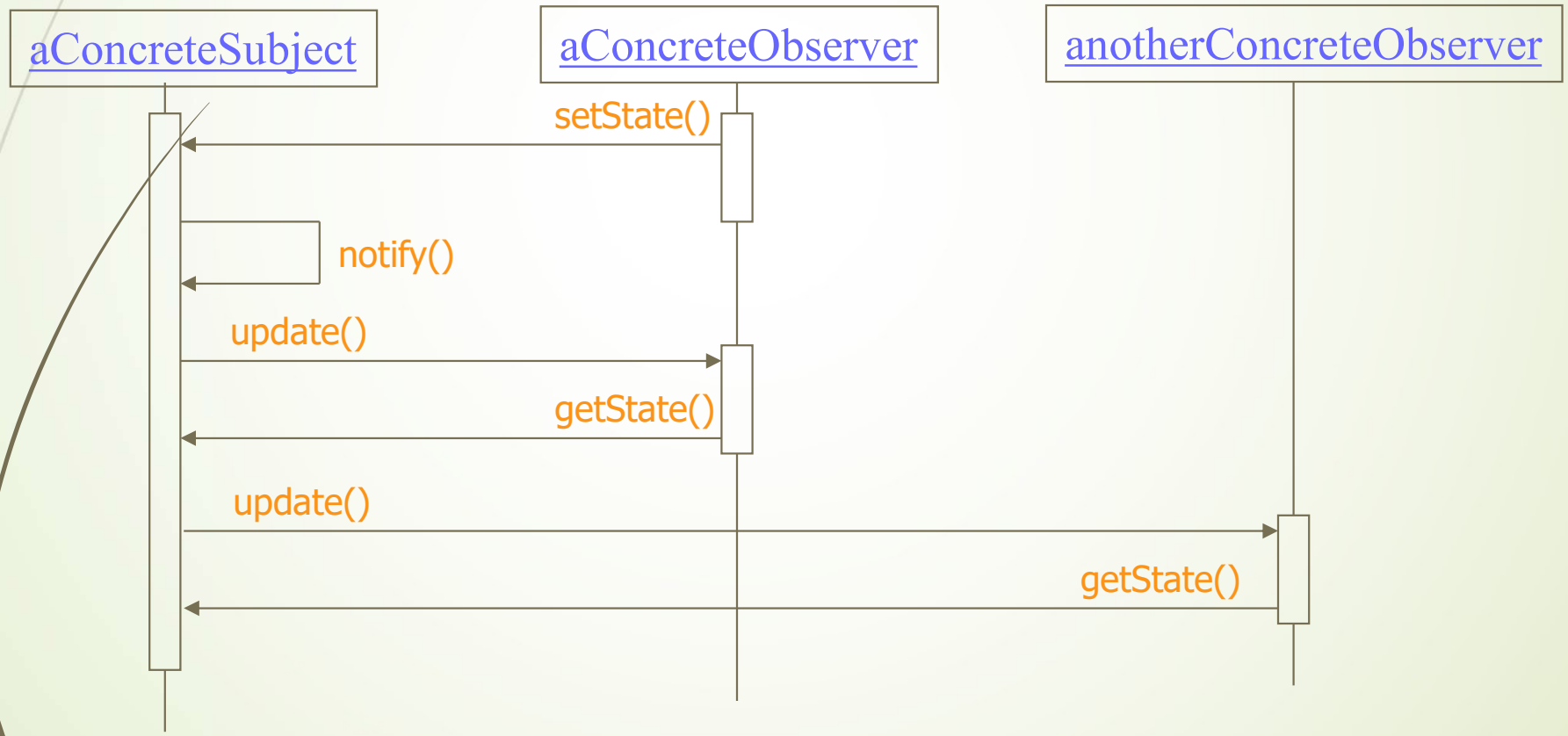
- Structure

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



Observer

- Typical interactions in Observer

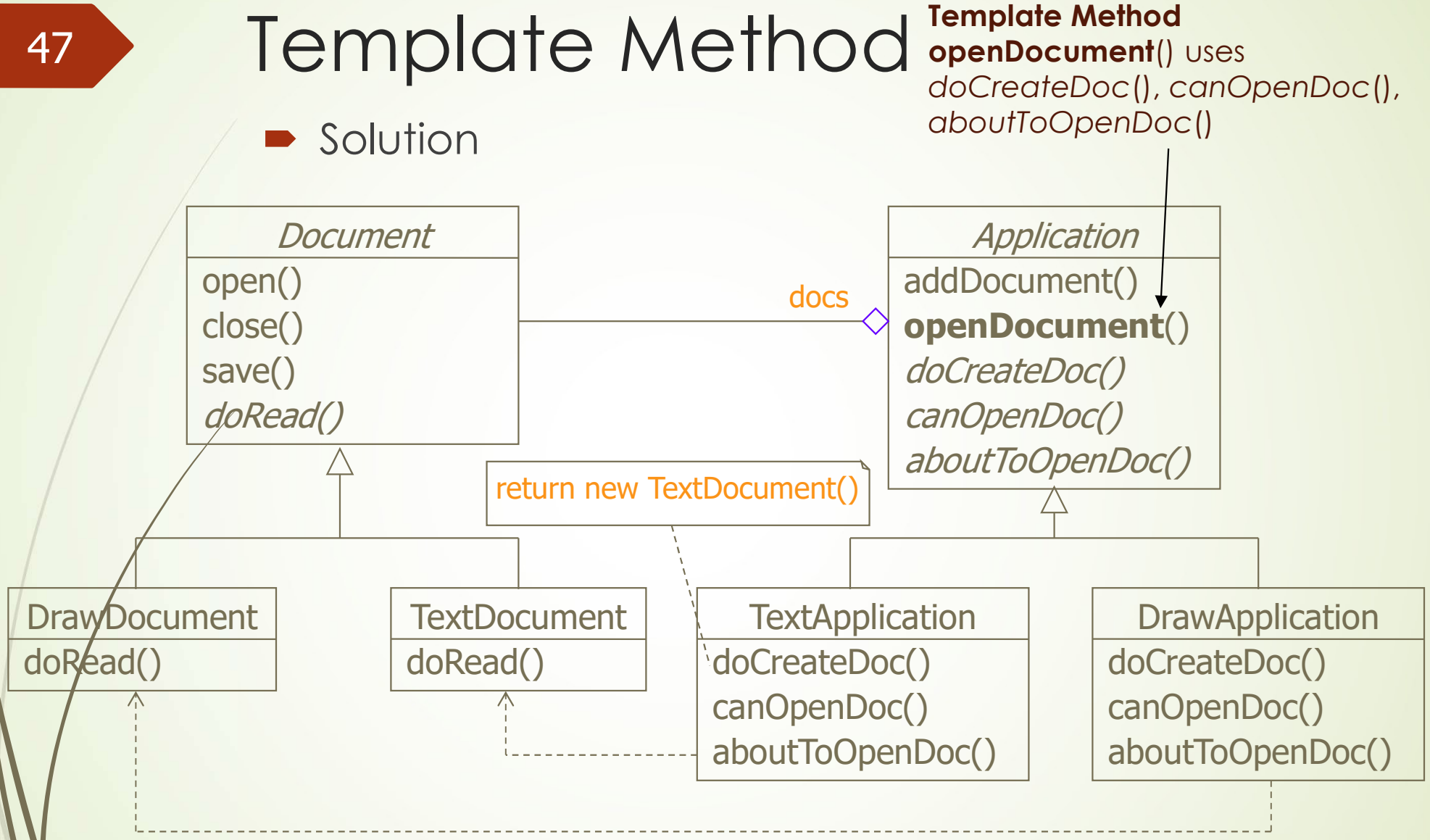


Template Method

- Motivation
  - We want to develop software, including the *Application* and *Document* classes, *Application* is responsible for opening an existing document from file. *Document* represents the information of a document. Specific applications, such as *DrawApplication* and *TextApplication*, inherit from *Application* to meet some specific needs.
- Problem
  - How to organize the program code of some operations, such as opening documents (*openDocument*) can be shared uniformly for different specific applications?

Template Method

- Solution



Template Method

- Method *openDocument* is called ***Template Method***

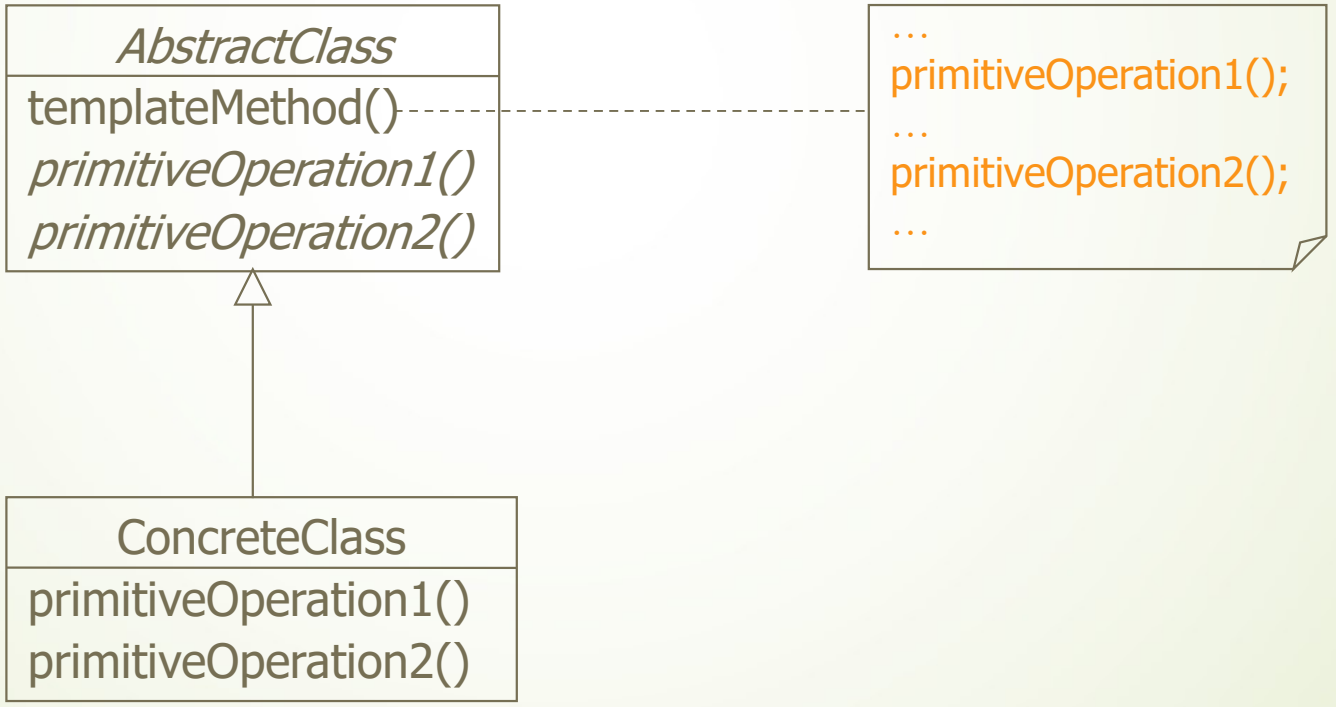
```
abstract class Application{
   abstract public Document doCreateDoc();
   abstract public Boolean canOpenDoc();
   …
   public void openDocument (String name) {
      if (!canOpenDoc(name))
      { // cannot handle this document
          return;
      }
      Document doc = doCreateDoc();
      if (doc) {
       docs.addDocument(doc);
       aboutToOpenDoc(doc);
       doc.open();
       doc.doRead();
      }
   }
   …
}
```

- ✓ *openDocument* defines the steps to open a document: checking document, creating document objects, adding documents to a set of documents, and reading documents from files.
- ✓ These steps will be implemented in subclasses (*TextApplication* and *DrawApplication*).

Template Method

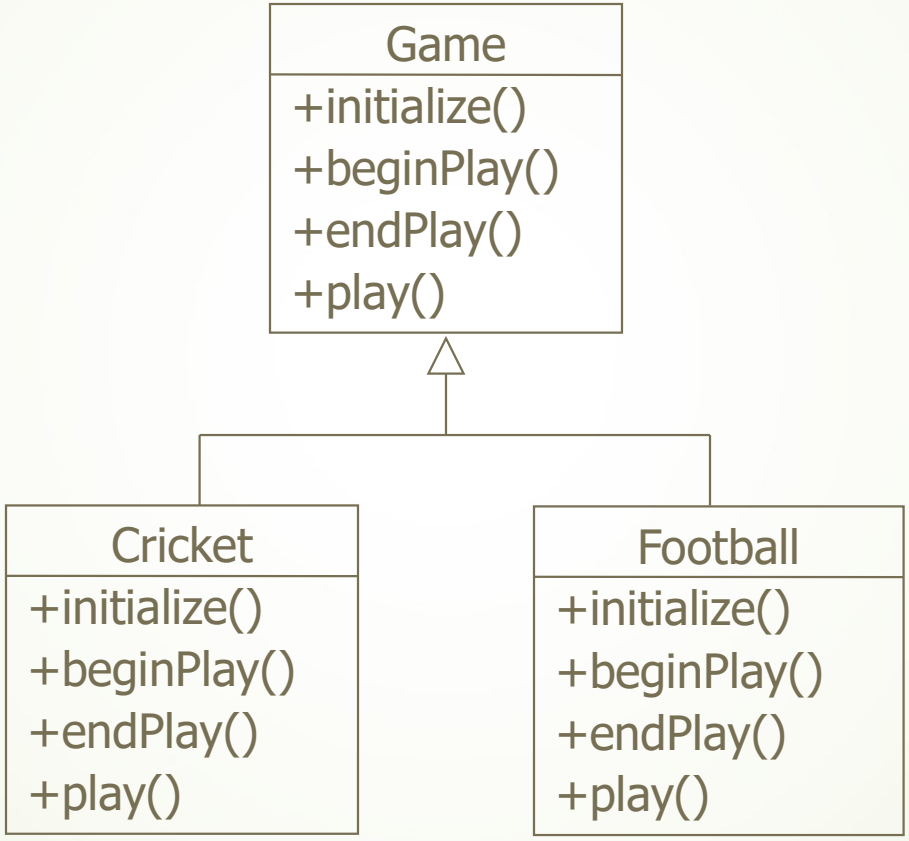
- Structure

Intent: Defining the skeleton of an algorithm in the superclass but letting subclasses override specific steps of the algorithm without changing its structure



Template Method

- Example with code



Template Method

- Create an abstract class with Template Method being final

```
public abstract class Game {
    abstract void initialize();
    abstract void startPlay();
    abstract void endPlay();

    //template method
    public final void play(){

        //initialize the game
        initialize();

        //start game
        startPlay();

        //end game
        endPlay();
    }
}
```

Template Method

- Create Cricket extending Game

```
public class Cricket extends Game {

    @Override
    void endPlay() {
        System.out.println("Cricket Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
}
```

Template Method

- Create Football extending Game

```
public class Football extends Game {

    @Override
    void endPlay() {
        System.out.println("Football Game Finished!");
    }

    @Override
    void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }

    @Override
    void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }
}
```

Template Method

- Use the *Game*'s template method play() to demonstrate a defined way of playing game

```
public class TemplatePatternDemo {
   public static void main(String[] args) {

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

More on design patterns

- References
  - **Design Patterns: Elements of Reusable Object-Oriented Software**, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1994
  - https://www.tutorialspoint.com/design_pattern/index.htm
  - https://refactoring.guru/design-patterns