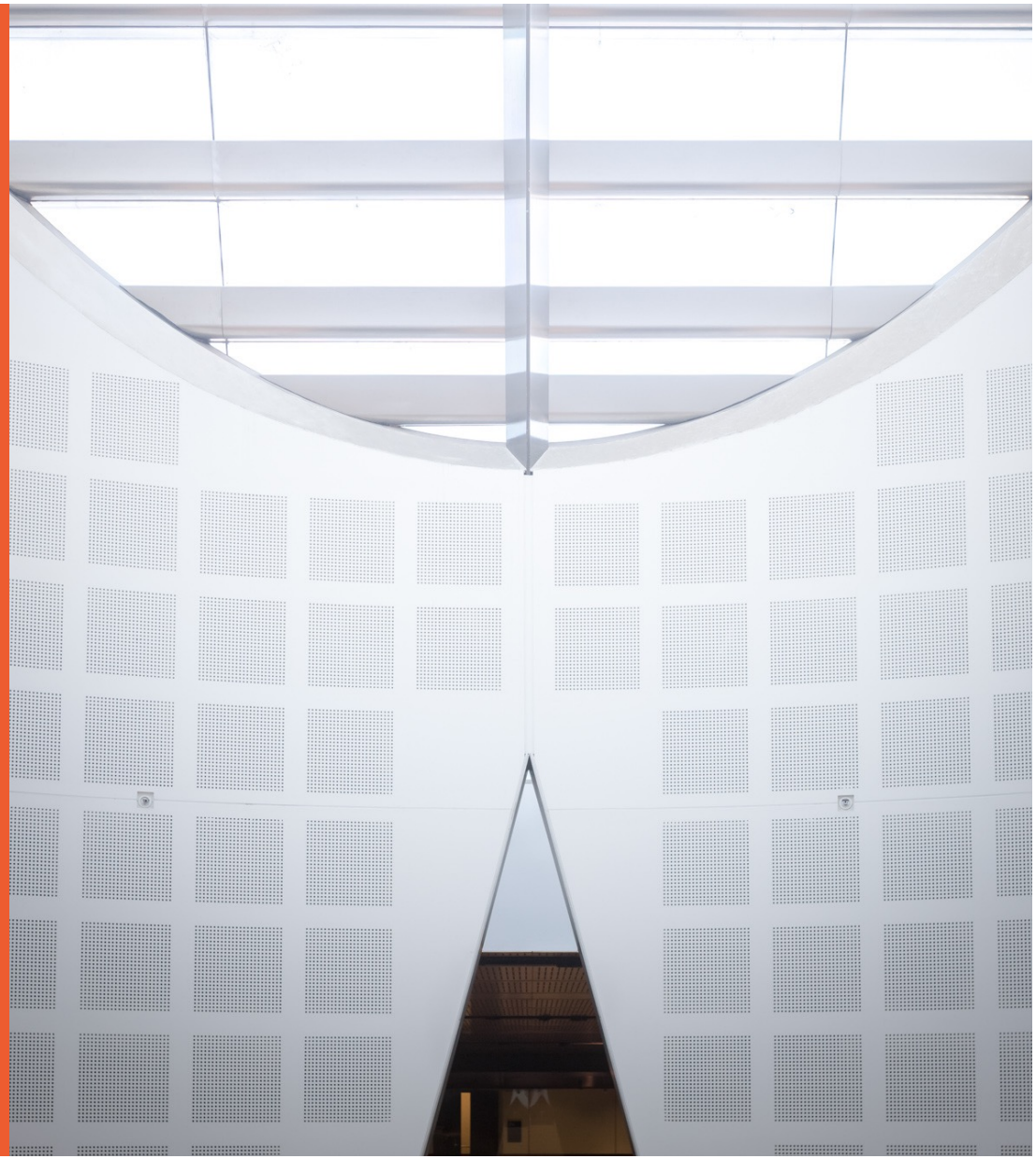
SOFT2201/COMP9201: Software Design and Construction 1

Singleton, Decorator, and Façade

Dr Xi Wu
School of Computer Science

THE UNIVERSITY OF SYDNEY



Copyright warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

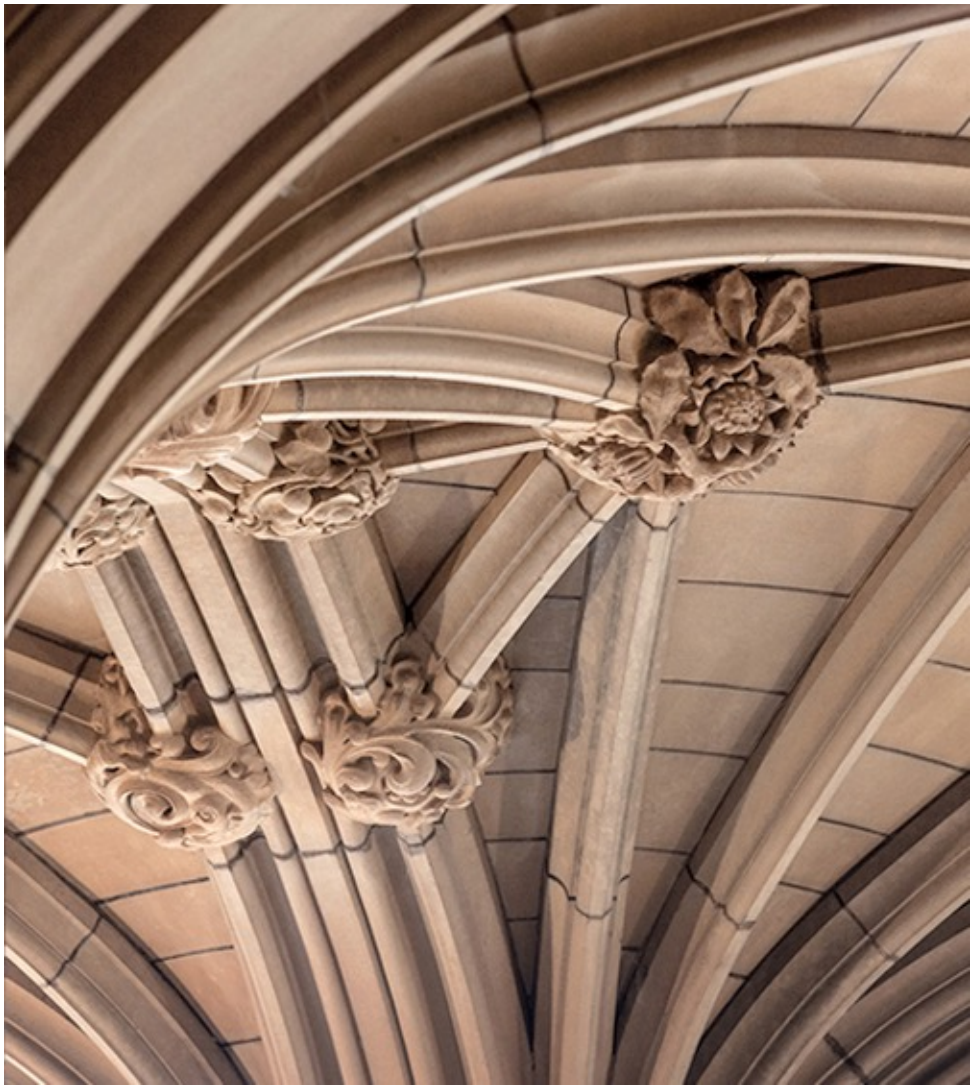
Agenda

- Creational Design Pattern
  - Singleton

- Structural Design Pattern
  - Decorator and Façade

Singleton Pattern

Object Creational



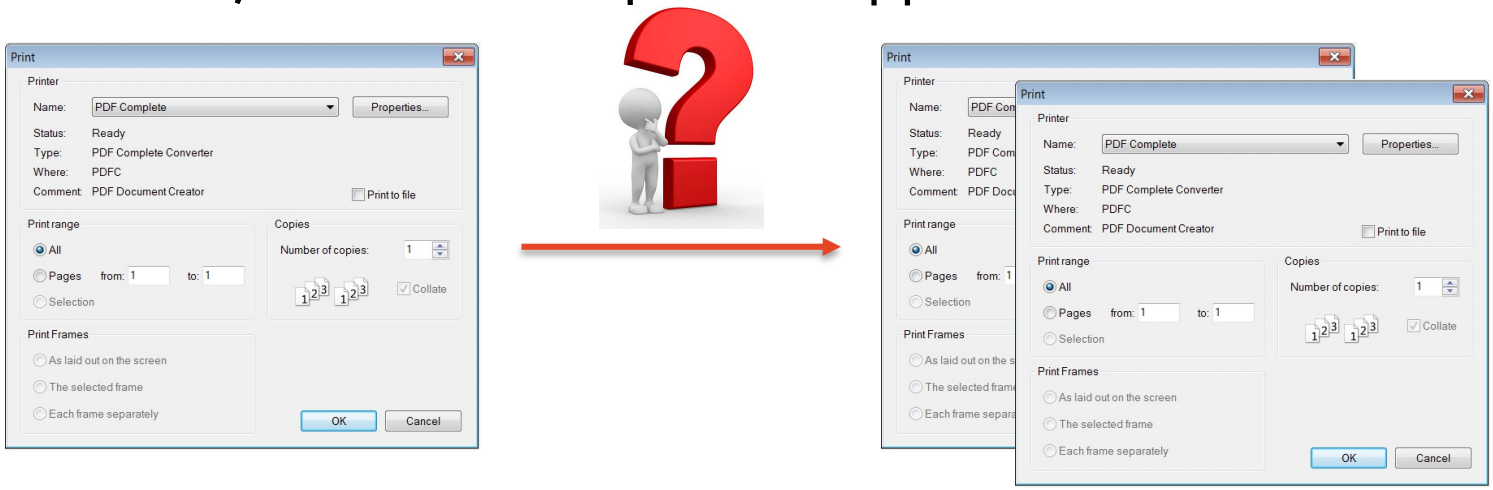
THE UNIVERSITY OF SYDNEY

Creational Patterns (GoF)

Pattern Name	Description
Factory Method	Define an interface for creating an object, but let sub-class decide which class to instantiate (class instantiation deferred to subclasses)
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations
Prototype	Specify the kinds of objects to create using a prototype instance, and create new objects by copying this prototype
Singleton	**Ensure a class only has one instance, and provide global point of access to it**

Motivated Scenario

- Suppose you have finished your assignment report in your computer and would like to print it out.
  - The first time you click on “print”, it pops up a printing set up window
  - The second time you click on “print” without closing the previous printing window, what do we expect to happen?

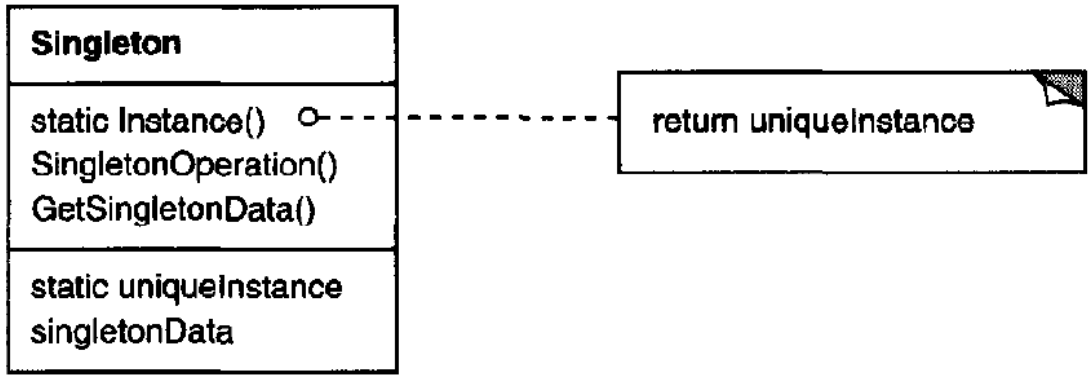


Singleton

- Intent
  - Ensure a class only has one instance, and provide a global point of access to it
- Motivation
  - Make the class itself responsible for keeping track of its sole instance (intercept requests to create new objects and provide a way to access its instance)

Singleton

- Structure



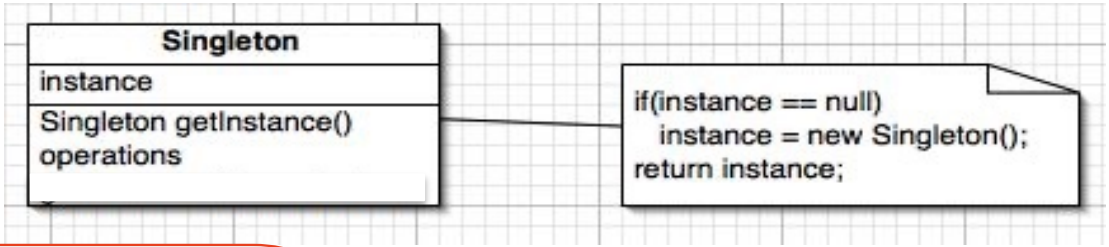
- Participants
  - Defines an instance() operation that lets clients access its unique instance. instance() is a class operation
  - May be responsible for creating its own unique instance

Singleton

- Collaboration
  - Clients access a Singleton instance solely through Singleton's instance() operation.

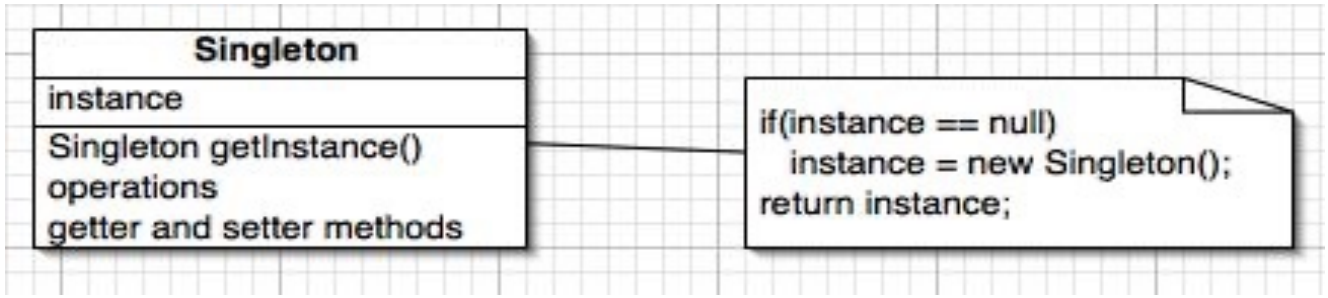
- Applicability
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
  - The sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton Implementation




```
public class Singleton {
      private static Singleton instance = null;
        // Private constructor to prevent direct
   initialisation.
      private Singleton() {
      }
      public static Singleton getInstance() {
         if (instance == null) {
            instance = new Singleton();
         }
         return instance;
      }
}
public class Client {
       …
       Singleton single = Singleton.getInstance();
}
```

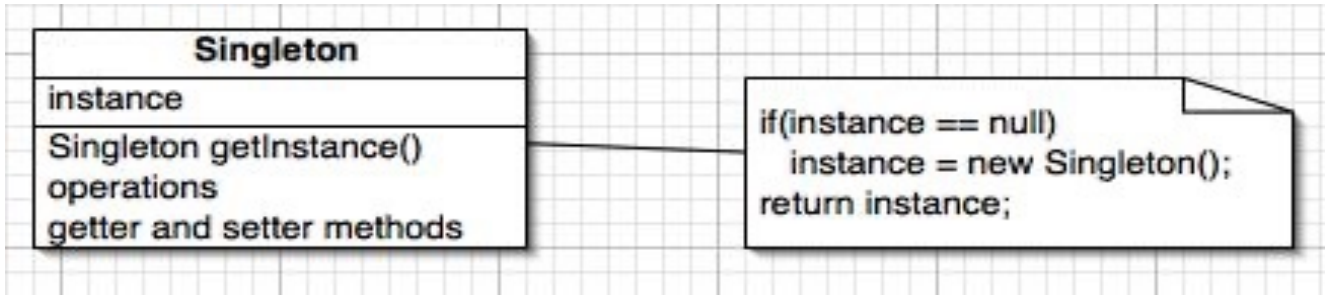
Singleton Example




```
public class ConfigurationReader {
   private static ConfigurationReader instance = null;

   private ConfigurationReader() { }
   public static ConfigurationReader getInstance() {
      if (instance == null) {
         instance = new ConfigurationReader();
      }
      return instance;
   }
}
public class Client {
    public boolean doStuff() {
        …
        ConfigurationReader theOneAndOnlyConfigurationFileReader = ConfigurationReader.getInstance();
    }
```

Singleton Example

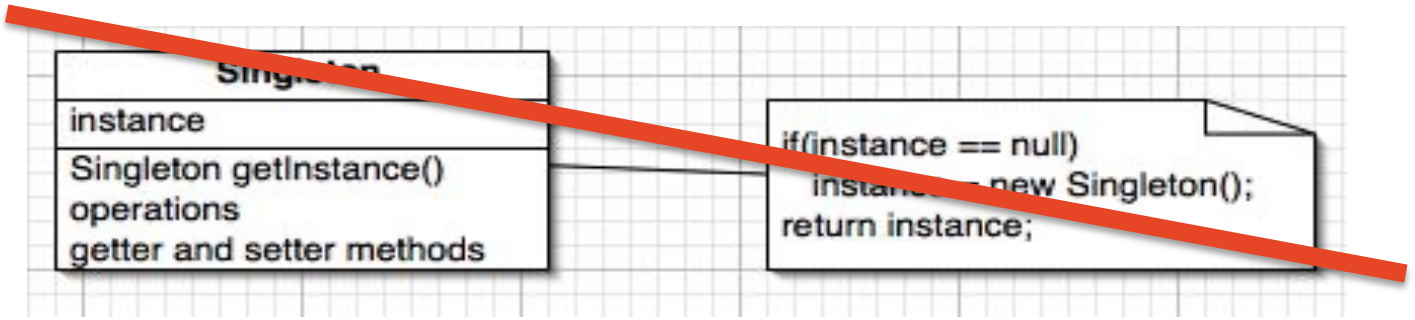



```
public class ConfigurationReader2 {
   private static ConfigurationReader2 instance = null;

   private ConfigurationReader2() { }
   public static ConfigurationReader2 getInstance() {
      if (instance == null) {
         instance = new ConfigurationReader2();
      }
      return instance;
   }
}
public class Client {
    public boolean doStuff() {
        …
        ConfigurationReader theOneAndOnlyConfigurationFileReader = ConfigurationReader.getInstance();
        ConfigurationReader2 theOtherConfigurationFileReader = ConfigurationReader2.getInstance();
    }
```

Do you really always want one specific instance of one specific class?

Singleton Alternative Example

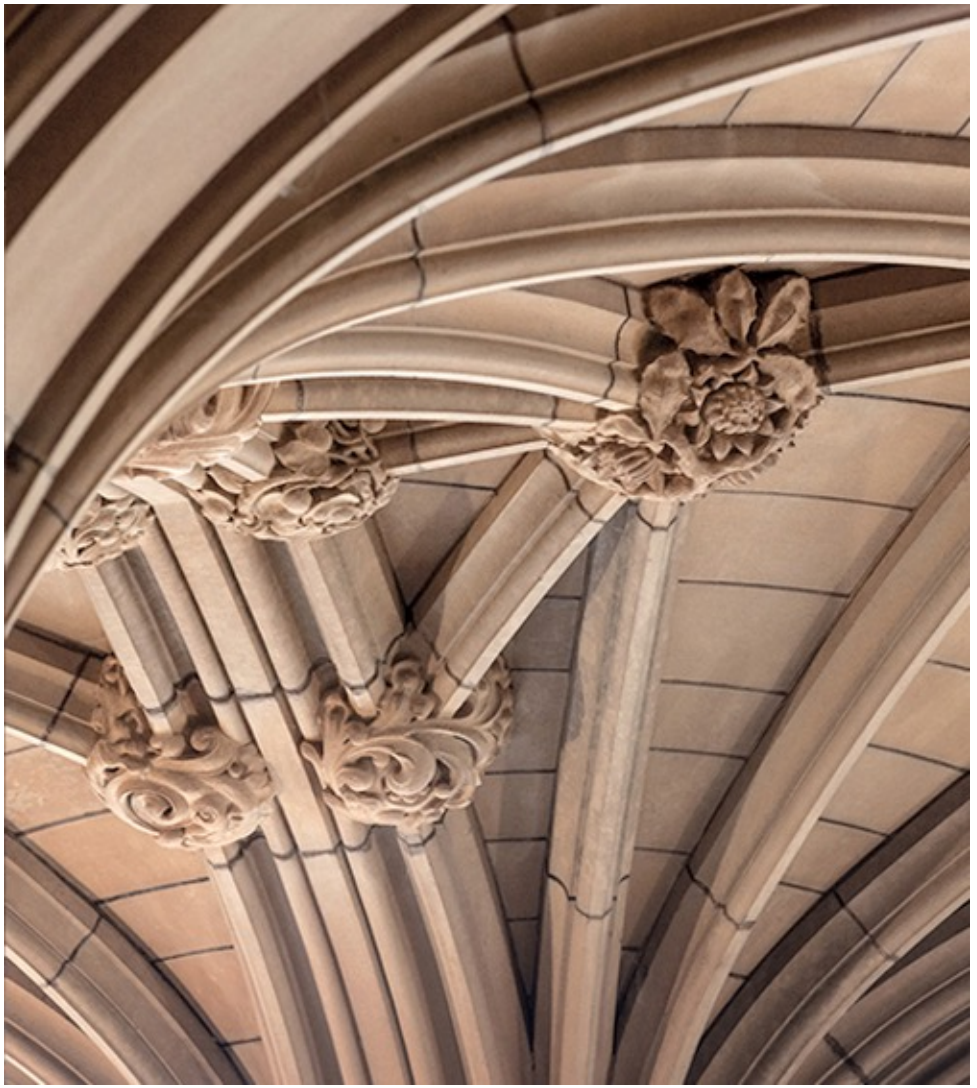



```
public class ConfigurationReader {
    public ConfigurationReader() { }
    // . . .
}

public class Client {
    // Explicit dependency in interface.
    // The 'single' instance is passed around by reference
    public boolean doStuff(ConfigurationReader configurationReader) {
        // . . .
}
```

Decorator Pattern

Object Structural



THE UNIVERSITY OF SYDNEY

Structural Patterns (GoF)

Pattern Name	Description
Adapter	Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.
Decorator	**Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)**
Façade	Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that simplifies subsystem use.

Motivated Scenario

- Suppose you are shopping in a famous store, which has a virtual clothing try on application.
  - T-Shirts, Trousers and Sneakers
  - Suit and Leather Shoes

Motivated Scenario

```
public class People {
    2 usages
    private String name;
    1 usage
    public People (String name){this.name = name;}
    1 usage
    public void wearTShirt() {System.out.print("T-Shirt ");}
    1 usage
    public void wearTrousers() {System.out.print("Trousers ");}
    1 usage
    public void wearSneakers() {System.out.print("Sneakers ");}
    public void wearSuit() {System.out.print("Suit ");}
    public void wearLeatherShoes() {System.out.print("Leather Shoes ");}
    1 usage
    public void show() {System.out.println(name + " is trying on:");}
}
```

Client Perspective:

```
People people = new People( name: "Happy");
people.show();
people.wearTShirt();
people.wearTrousers();
people.wearSneakers();
```

Output:

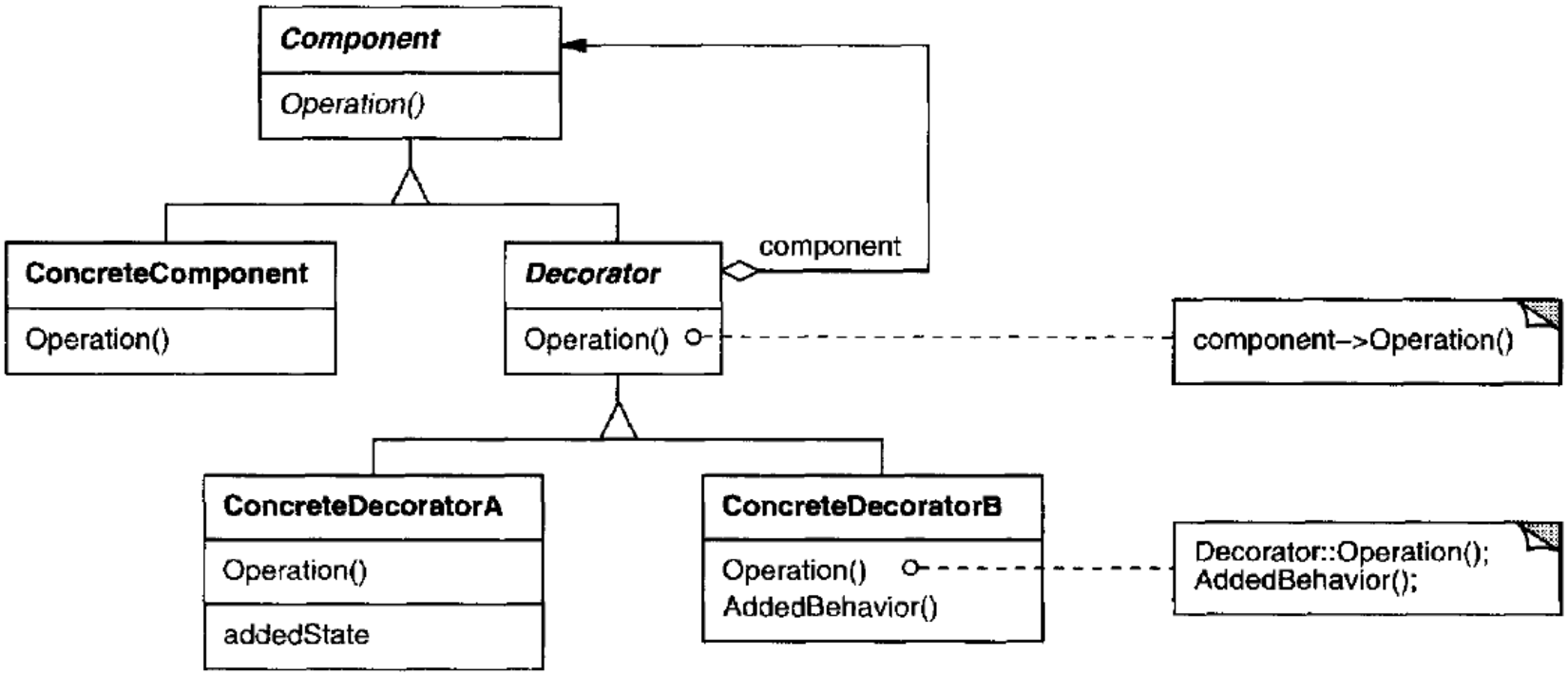
```
Happy is trying on:
T-Shirt Trousers Sneakers
```

Question: I also want to try on Dress and Boots. What will happen here?

Decorator Pattern

- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality

Decorator – Structure



Decorator – Participants

- **Component**
  - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent**
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator**
  - Adds responsibilities to the component

Revisit the Motivated Example

```
public interface People {
    7 usages  7 implementations
    public void show();
}
```

```
public class Women implements People{
    2 usages
    private String name;
    1 usage
    public Women(String name){this.name = name;}
    7 usages
    public void show() {System.out.println(name + " is trying on:");}
}
```

```
public abstract class Cloth implements People {
    2 usages
    protected People people;
    5 usages
    public Cloth (People people){this.people = people;}
    7 usages  5 overrides
    public void show(){people.show();}
}
```

Revisit the Motivated Example

```java
public abstract class Cloth implements People {
    2 usages
    protected People people;
    5 usages
    public Cloth (People people){this.people = people;}
    7 usages  5 overrides
    public void show(){people.show();}
}
```

```java
public class TShirt extends Cloth{
    1 usage
    public TShirt(People people) {super(people);}
    7 usages
    public void show() {
        super.show();
        System.out.print("T-Shirt ");
    }
}
```

```java
public class Trousers extends Cloth{
    1 usage
    public Trousers (People people) {super(people);}
    7 usages
    public void show() {
        super.show();
        System.out.print("Trousers ");}
}
```

```java
public class Sneakers extends Cloth{
    1 usage
    public Sneakers(People people){super(people);}
    7 usages
    public void show(){
        super.show();
        System.out.print("Sneakers ");}
}
```

Revisit the Motivated Example

Client Perspective

```
People people = new Women( name: "Happy");

TShirt tShirt = new TShirt(people);
Trousers trousers = new Trousers(tShirt);
Sneakers sneakers = new Sneakers(trousers);

sneakers.show();
```

Output

```
Happy is trying on:
T-Shirt Trousers Sneakers
```

Question 1:
I also want to try on Dress and Boots.
What will happen here?

Question 2:
This system opens to Men and Baby.
What will happen here?

Decorator Pattern – Why Not Inheritance?

- We want to add responsibilities to individual objects, not an entire class
  - E.g., A GUI toolkit should let you add properties like borders or behaviors like scrolling to any user interface component

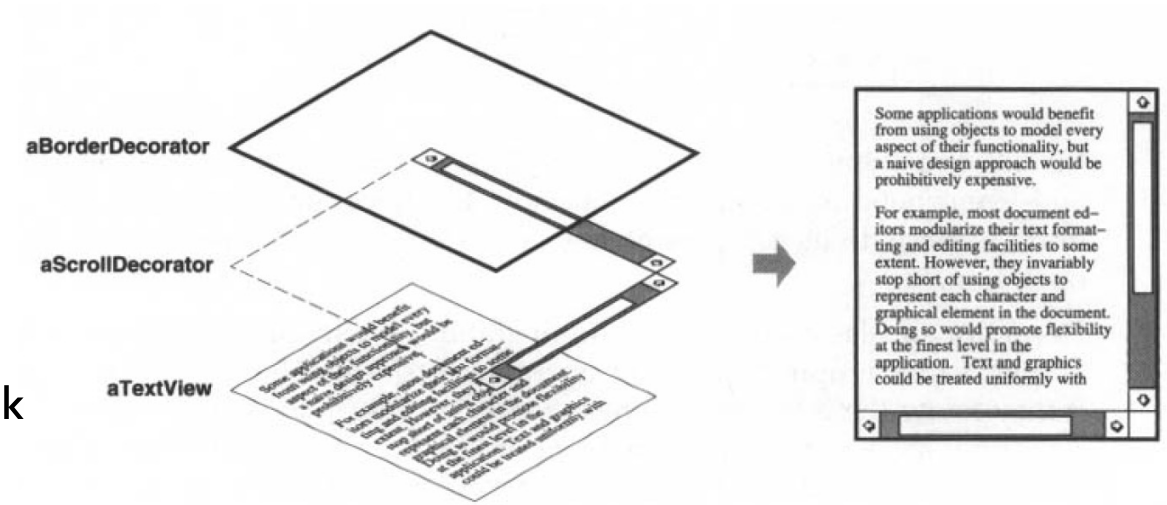
- Is adding responsibilities using inheritance a good design? For example, inheriting a border class puts a border every subclass instance
  - Why, why not?

Decorator Pattern – Why Not Inheritance?

- Adding responsibilities using inheritance restricts runtime change, and requires an implementation for every decoration.
  - This design is inflexible
    - The choice of border is made statically; a client cannot control how and when to decorate the component with a border
    - More flexible design is to enclose the component in another object that adds the border

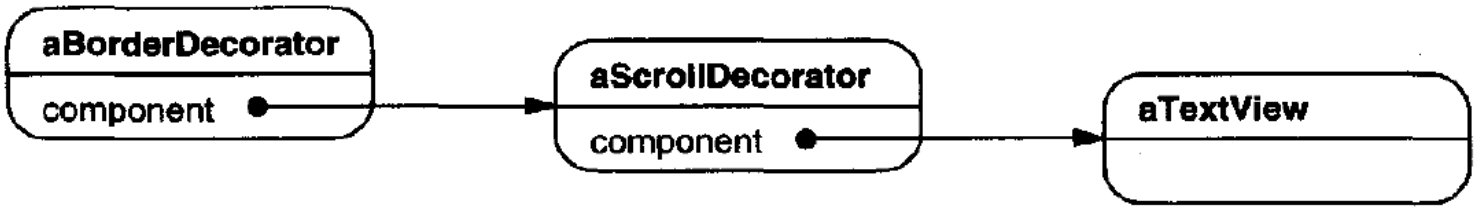
Decorator Pattern – Text Viewer Example

- TextView object has no scroll bars and border by default (not always needed)
- ScrollDecorator to add them
- BorderDecorator to add a thick black border around the TextView

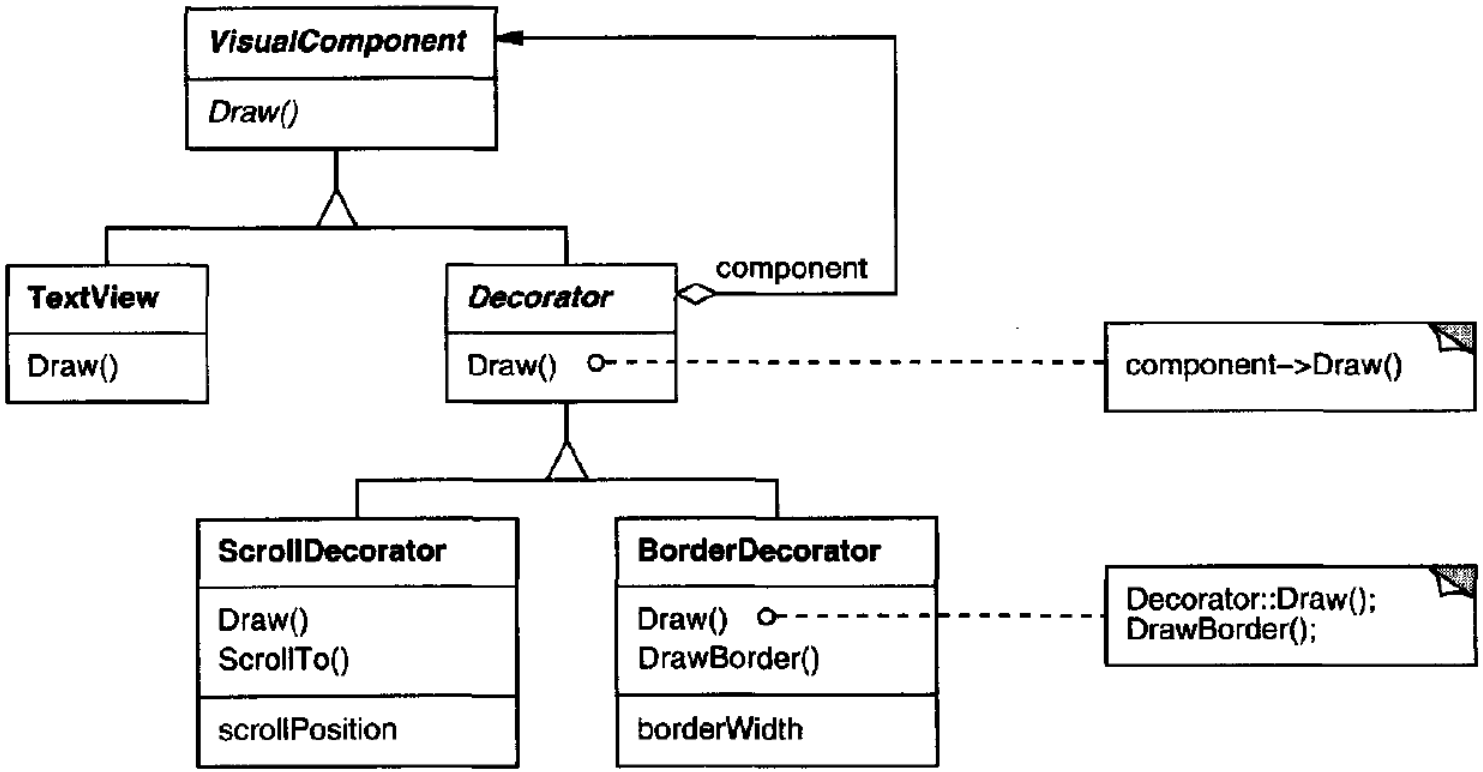


Decorator Pattern – Text Viewer Example

- Compose the decorators with the TextView to produce both the border and the scroll behaviours for the TextView



Decorator Pattern – Text Viewer Example



Decorator – Participants

- **Component** (*VisualComponent*)
  - Defines the interface for objects that can have responsibilities added to them dynamically
- **ConcreteComponent** *(TextView)*
  - Defines an object to which additional responsibilities can be attached
- **Decorator**
  - Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
- **ConcreteDecorator** *(BorderDecorator, ScrollDecorator)*
  - Adds responsibilities to the component

Decorator – Text Viewer Example

- VisualComponent is the abstract class for visual objects
  - It defines their drawing and event handling interface
- Decorator is an abstract class for visual components that decorate the other visual components
  - It simply forwards draw requests to its component; Decorator subclasses can extend this operation
- The ScrollDecorator and BorderDecorator classes are subclasses of Decorator
  - Can add operations for specific functionality (e.g., ScrollTo)

Decorator

- Collaborations
  - Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.
- Applicability
  - to add responsibilities to individual objects dynamically and transparently, without affecting other objects
  - For responsibilities that can be withdrawn
  - When extension by sub-classing is impractical
    - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

Consequences (1)

- More flexibility and less complexity than static inheritance
  - Can add and remove responsibilities to objects at run-time
  - Inheritance requires adding new class for each responsibility (increase complexity)

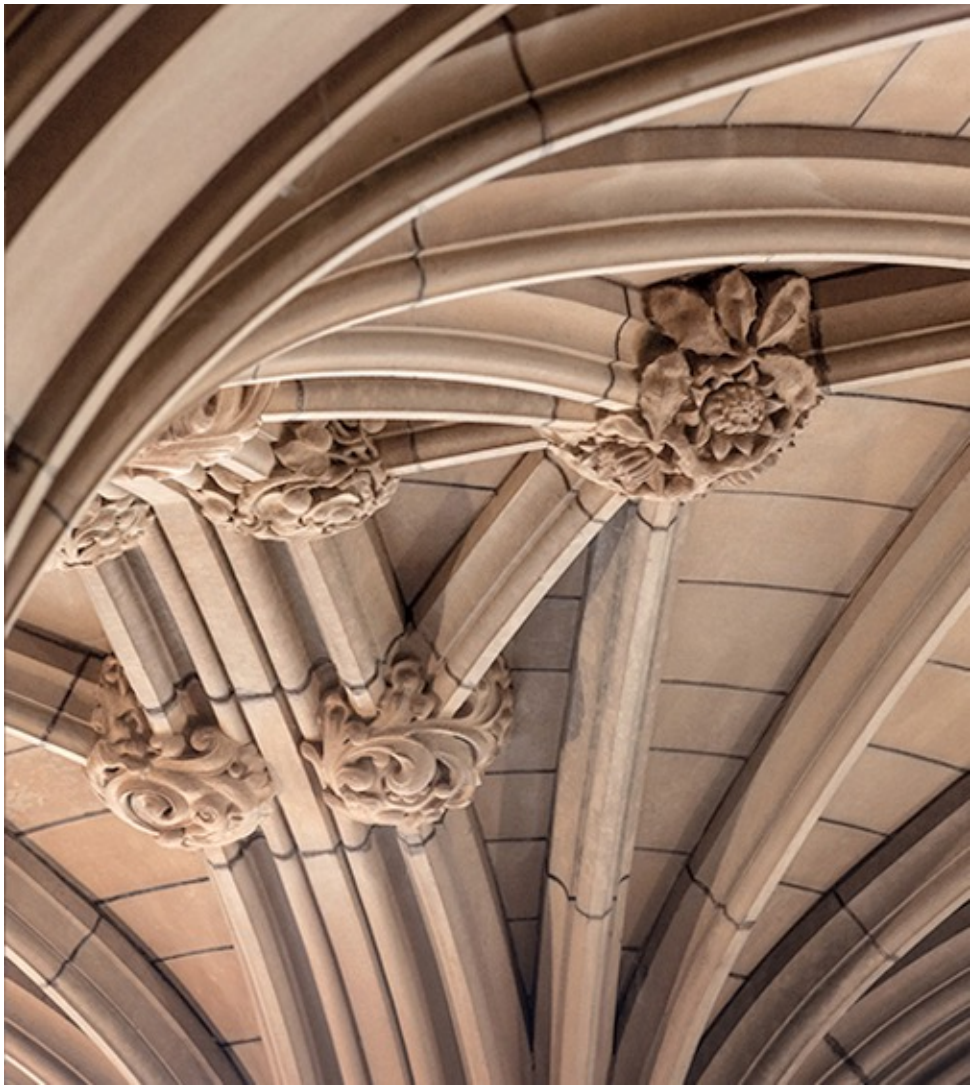
- Avoids feature-laden (heavily loaded) classes high up in the hierarchy
  - Defines a simple class and add functionality incrementally with Decorator objects – applications do not need to have un-needed features
  - You can define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions

Consequences (2)

- Decorator and its component are not identical
  - Decorated component is not identical to the component itself - you shouldn't rely on object identity when you use decorator
- Many little objects
  - Can become hard to learn and debug when lots of little objects that look alike
  - Still not difficult to customize by those who understand them

Façade Pattern

Object Structural



THE UNIVERSITY OF SYDNEY

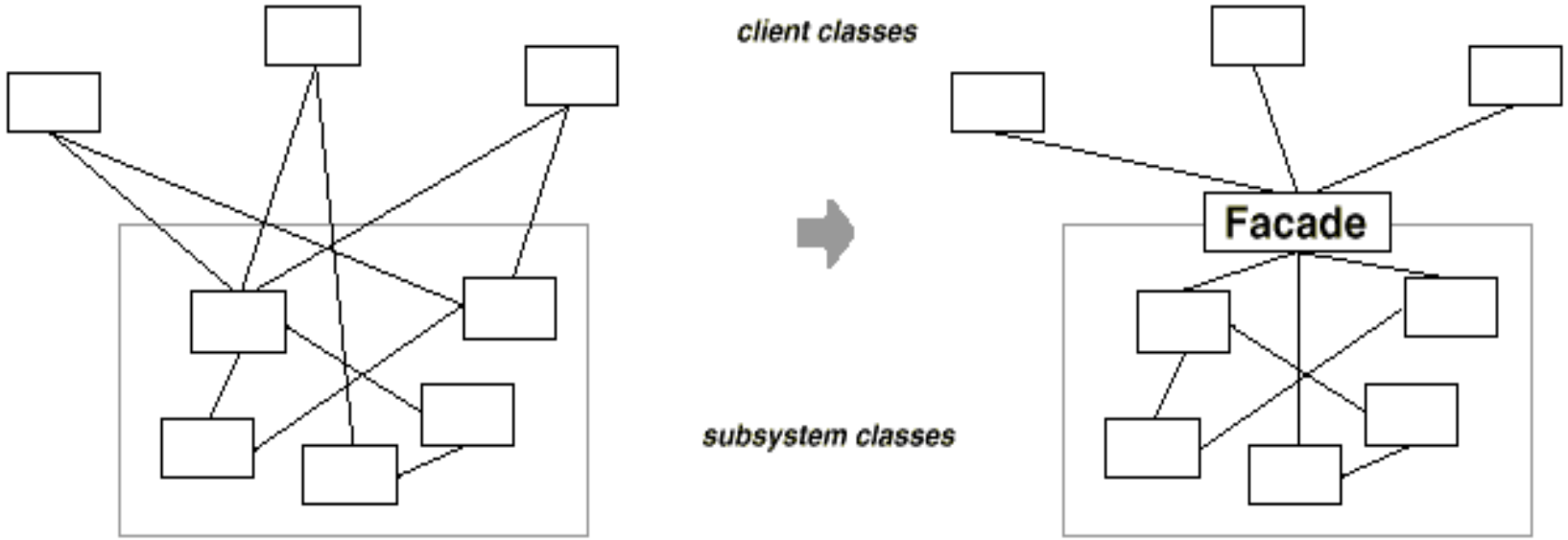
Structural Patterns (GoF)

Pattern Name	Description
Adapter	Allow classes of incompatible interfaces to work together. Convert the interface of a class into another interface clients expect.
Decorator	Attach additional responsibilities to an object dynamically (flexible alternative to subclassing for extending functionality)
Façade	**Provides a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface that simplifies subsystem use.**

Motivated Scenario

- Suppose you are going to seek some advices from your UG/PG academic advisors. However, there are many academics and you might know who takes care of your major.

Façade Motivation



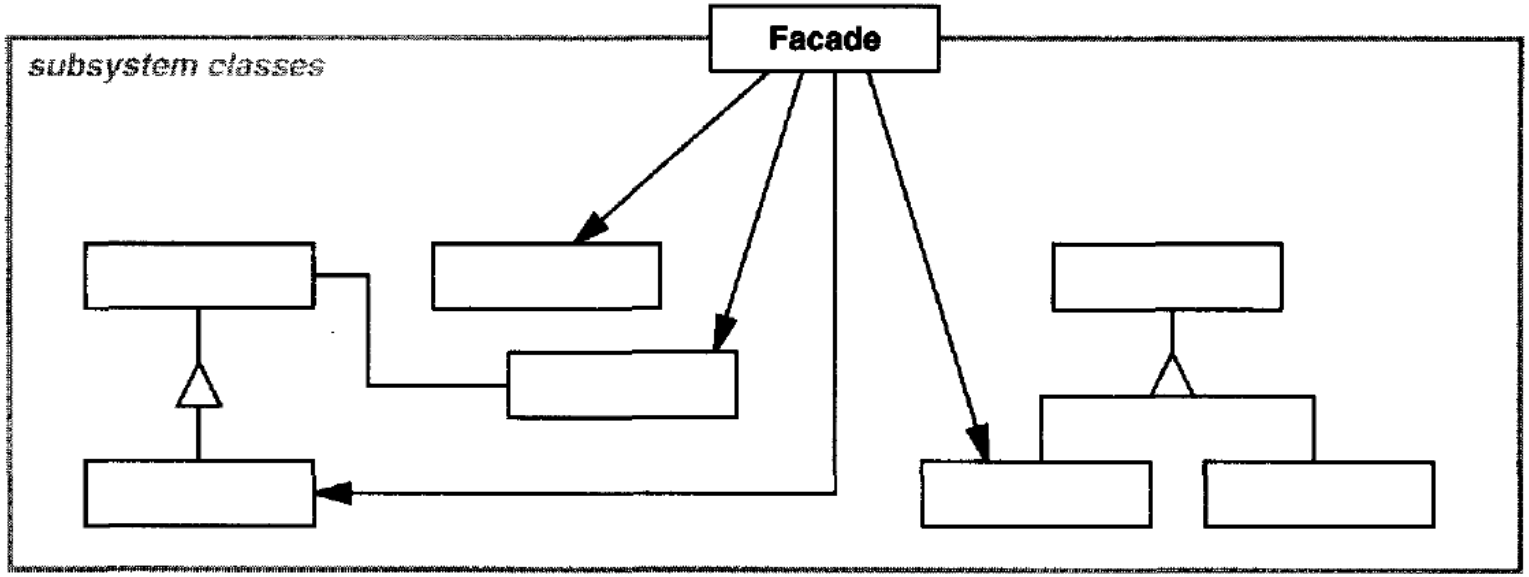
A **facade** object provides a single, simplified interface to the more general facilities of a subsystem

Façade Pattern

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use

- Applicability
  - You want to provide a simple interface to a complex subsystem
  - There are many dependencies between clients and the implementation classes of an abstraction
  - You want to layer your subsystem. Façade would define an entry point to each subsystem level

Façade – Structure



Façade – Participants

- **Facade**
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **Subsystem classes**
  - Implement subsystem functionality.
  - Handle work assigned by the Façade object
  - Have no knowledge of the facade; they keep no references to it.
- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).
    - Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces
  - Clients that use the facade don't have to access its subsystem objects directly

Façade

- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Façade, which forwards them to the appropriate subsystem object(s).
    - Although the subsystem objects perform the actual work, the façade may have to do work of its own to translate its interface to subsystem interfaces
  - Clients that use the facade don't have to access its subsystem objects directly

Façade Pattern - Example

```
class subClass1 {
        public void method1() {
                // method body }
}
class subClass2 {
        public void method2() {
                // method body }
}
```

How about Client?

```
Façade façade = new Façade();
façade.methodA();
```

```
class Façade {
        subClass1 s1;
        subClass2 s2;
        public Façade() {
                s1 = new subClass1();
                s2 = new subClass2();
        }
        public void methodA() {
                s1.method1();
                s2.method2();
        }
}
```

Consequences

- Simplify the usage of an existing subsystem by defining your own interface
- Shields clients from subsystem components, reduce the number of objects that clients deal with and make the subsystem easier to use.
- Promote weak coupling between the subsystem and the clients
  - Vary the components of the subsystem without affecting its clients
  - Reduce compilation dependencies (esp. large systems) – when subsystem classes change
- Does not prevent applications from using subsystem classes if they need to. Choice between ease of use and flexibility.

Façade – A Brief Summary

- Name: **Façade**

- Problem: A common, unified interface to a disparate set of implementations or interfaces is required. There may be undesirable coupling to many things in the subsystem, or the implementation of the subsystem may change

- Solution: Define a single point of contact to the subsystem – a façade object that wraps the subsystem. It represents a single unified interface and is responsible for collaborating with the subsystem components.

Task for Week 12

- Submit weekly exercise on canvas before 23.59pm Saturday
- Well organize time for assignment 3

What are we going to learn on week 13?

- Unit Review

References

- Craig Larman. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.