Design Patterns

Session Outcomes

- Introduction to Design patterns
- Creational Patterns
  - Singleton
  - Abstract Factory
- Structural Patterns
  - Flyweight
  - Façade
  - Decorator
- Behavioural Patterns
  - Observer
  - Template Method
  - Visitor

Overview

- Introduction: what is a software pattern?
- Types of Pattern.
- Elements of a Design Pattern.
- Design Patterns:
  - **creational**
    - Singleton, Abstract Factory
  - **structural**
    - Flyweight, Decorator, Façade
  - **behavioural**
    - Visitor, Template, Observer

Introduction to Patterns

- Patterns are an emerging topic in software engineering.
- All **well structured systems are full of patterns**.
- The principle behind patterns is to identify, document, and hence re-use general **solutions to common problems.**
- The aim is to promote re-use in all phases of the software lifecycle, not just implementation (i.e. code).
- Patterns are discovered, they are not invented.
- The idea for patterns is borrowed from a similar technique used by architects.

Introduction

• The principle behind software patterns is to identify, document, and hence re-use general solutions to common problems.

- *A similar technique is used by architects:*
  - *"Vajira Builders" advertise 'patterns' for new houses;*
  - *these patterns can be applied in different contexts . e.g. a small block in Colombo, or a large one in Malabe;*
  - *each time a pattern is used, it is customized.*

A Brief History

- Patterns in Software Engineering were pioneered by the **Gang of Four** in 1994:
  - Erich Gamma,
  - Richard Helm,
  - Ralph Johnson,
  - John Vlissides.
- Their work was inspired by that of an architect:
  - Christopher Alexander (1977) .
- A Pattern Language: Towns/Buildings/Construction.

A Brief History

- Alexander realized that many buildings are variations on a theme - rooms, walls, doors, etc
  - e.g. in most houses all the wet areas are close together;
  - e.g. most houses have an entrance door at the front;
  - e.g. most office buildings have a reception area.

- In OO software there are many different classes:
  - the arrangement of classes determines the structure (and function) of the program.
- Just as there are standard patterns for buildings, so there are also patterns for designing software.

Software Patterns

- Building OO software is hard - building reusable OO software is harder:
  - it is impossible to teach experience; however.
  - it is possible to build on the experience of others.
- Patterns promote re-use in all phases of the software lifecycle, not just the coding phase.
  - there are analysis patterns, design patterns, and others.
- Since 1994 the patterns literature has become vast:
  - new candidate patterns are discovered every day!

Pattern Types

- Software Engineering Patterns can be classified in various ways .
- Classification of patterns used in Software Engineering according to **scale**:

1. **Architectural;**
2. **Design;**
3. **Idiom.**

Pattern Types 1: Architectural

- These patterns provide an overall architecture (i.e. a large scale design) for a software system.
- *Examples:*
  - the **Smalltalk Model View Controller (MVC)** architecture;
  - the **single document interface (SDI)** and **multiple document interface (MDI)** architectures used in many Windows applications;
  - the **pipeline architecture** in Unix;
  - the **ubiquitous client/server architecture**.

Pattern Types 2: Design

- Design Patterns are not analysis patterns, are not descriptions of common structures like linked lists, nor are they detailed application or framework designs.

- Design Patterns are
  - **"Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."**

Pattern Types 3 : Idiom

- These patterns describe a solution to a problem that is **language specific**:
  - the solution is described completely in terms of how it is implemented in a specific programming language.
  - The excellent books by Scott Myers contain lots of C++ idioms...
  - “Effective C++” (50 ways to improve C++ code);
  - “More Effective C++” (35 ways to improve C++ code).

Pattern Types: Idiom

- Example: whenever you create a new class in Java make sure you define the following for it:
  - a constructor where the import argument is an instance of the same class . i.e. a copy constructor:
    - MyClass object2 = new MyClass(object1);
- An equals() method where the import argument is an instance of the same class . i.e. an equals operator:
  - if(object2 == object1) doSomething();
    - a toString() method.

Design Patterns: Why Bother?

- **To identify the common features** that a design has:
  - with other software designs;
  - within itself - i.e. when different sections of a design represent the same basic problem in different contexts.
- **To promote re-use** of common but 'non-obvious‘ designs:
  - if a catalogue of patterns is available then it can be used to provide 'tried and tested' solutions.
- **To provide guidelines** for implementing these solutions

Design Pattern Classification

Creational Patterns

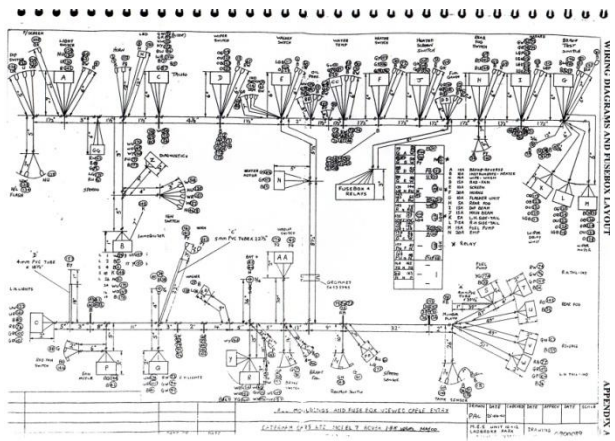
Concerns the process of object creation

Structural Patterns

Deals with the composition of classes /objects

Behavioural Patterns

Characterize the ways in which classes/objects
interact and distribute responsibility



Final Product

Design Pattern Space

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

Defer object creation to another class

Defer object **creation** to another object

Describe **ways to assemble objects**

Describe **algorithms and flow control**

Format Of A Design Pattern

Name	Name and classification - (taxonomy).
Intent	what the pattern is meant to achieve,
Motivation	why you'd want to use it
Applicability	where you'd want to use it
Structure	what the pattern looks like (diagrams).
Participants	classes used to realize the pattern.

Format Of A Design Pattern

Collaborations	how the classes collaborate with each other, and how the system should collaborate with the pattern.
Consequences	good and bad.
Sample Code	
Known Uses	
Related Patterns	

Elements - Pattern Name

- Patterns are collected into catalogues:
  - catalogued patterns will also include a classification.
- The name of a pattern must be meaningful, and consistent with the conventions used in existing pattern catalogues:
  - e.g. you cannot call a new pattern 'Singleton' because there is already a pattern with that name.
- The name can be a single word or a short phrase.
  - . Some patterns have aliases which are documented under the heading *Also Known As*.

Elements - Problem & Context

- **Problem Statement:**
  - describes the goal(s) that the pattern must achieve in the given context.
- **Context:**
  - the generic situation in which the problem occurs,including any pre-conditions;
  - this is often best illustrated with an example;
    - the context describes where the pattern has been applied successfully, and hence where it can be applied in future.

Elements - Solution

- The collection of entities (participants) which achieve the desired goal(s).
- A solution is represented in UML as a collaboration, in other words as a collection of:
  - class diagrams;
  - sequence diagrams;
  - collaboration (communication) diagrams.
- The solution often includes sample code.

Elements - Consequences & Justification

- **Consequences:**
  - a description of the system after the pattern has been applied;
  - this includes any side-effects of the solution . i.e. both the good and bad results of applying the pattern.
- **Justification:**
  - explains how the solution tackles the problem in a way that is consistent with modern design principles such as...
    - maintainability, portability and re-usability.

Creational Patterns

Singleton

Problem

- We need to make sure that only one instance of a class can be created.
- We want that instance to be easy to access anywhere in the application.

Singleton – Intent and Motivation

- **Intent-** Ensure a class has only one instance (i.e. object), and provide a global point of access to it.
- **Motivation**
  - Some classes need exactly one instance:
    - e.g. the Administrator boundary class from the practical exercises should only ever have one instance because there should only be one administrator of the system.
    - Eg: logger
  - A lazy designer might simply document the singleton constraint.
  - A good designer would make the class itself responsible for enforcing the constraint.

Singleton - Implementation

- Q1 - How will you implement singleton design pattern in java?

1. Declare a default private constructor

```
public MyClass {

    private MyClass() {}

}
```

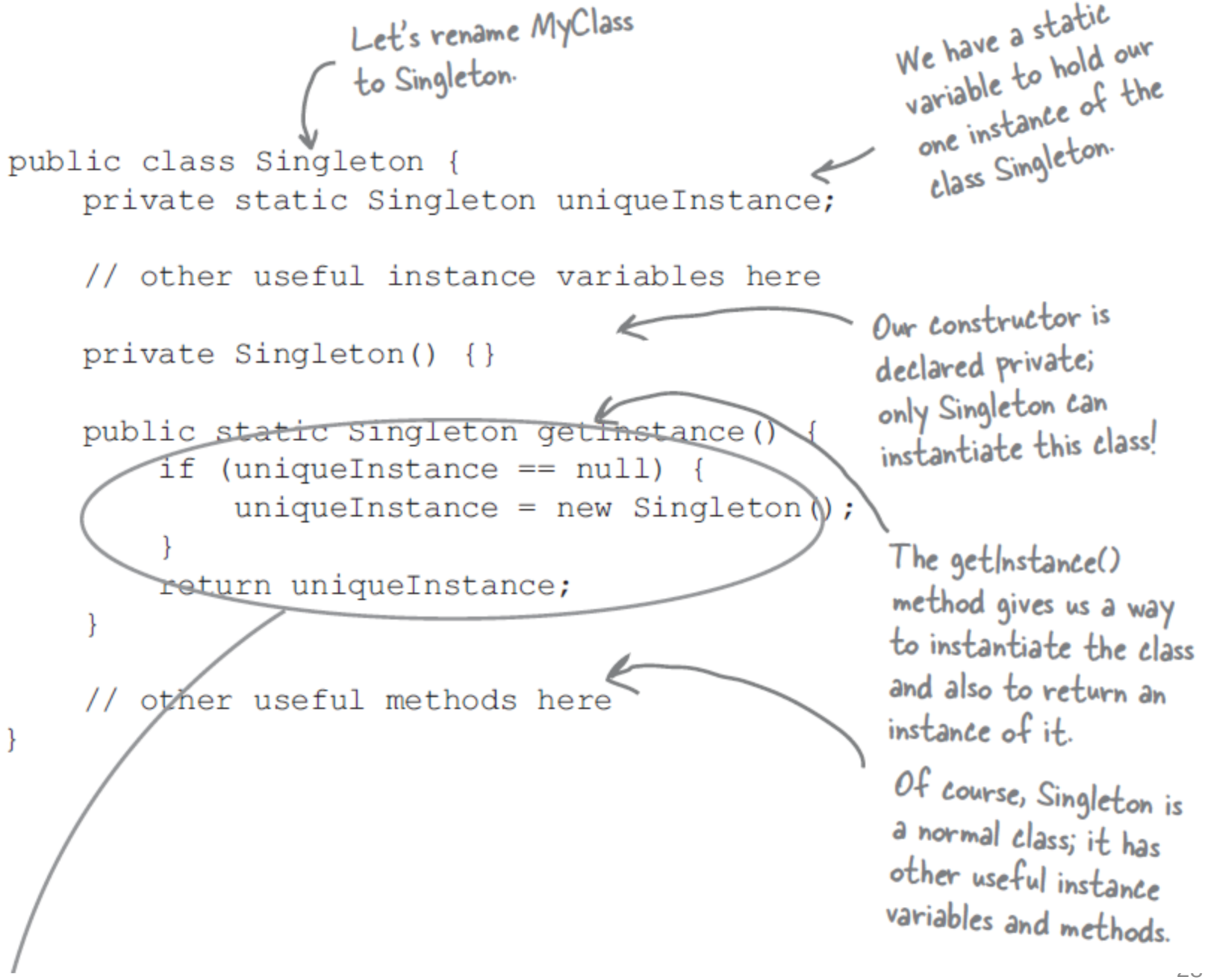
   – Chicken and egg problem

   – How to solve it?

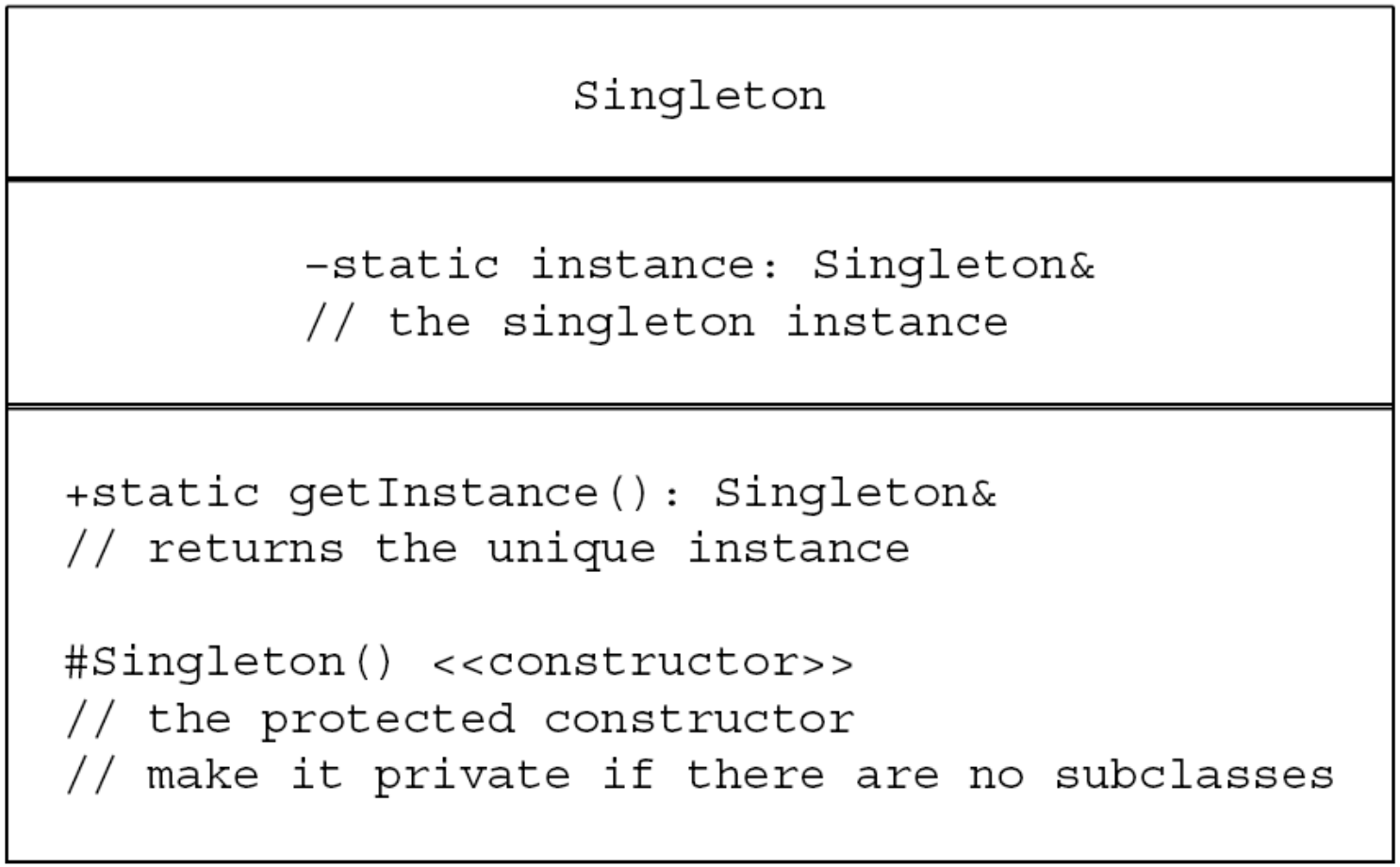
```
public MyClass {

    private MyClass() {}

    public static MyClass getInstance() {
        return new MyClass();
    }
}
```

Singleton- Implementation

2. Declare a private static variable to hold single instance of class.
3. Declare a public static function that returns the single instance of class.
4. Do “lazy initialization” in the accessor function.


Let's rename MyClass to Singleton.
We have a static variable to hold our one instance of the class Singleton.
public class Singleton {
private static Singleton uniqueInstance;
// other useful instance variables here
private Singleton() {}
public static Singleton getInstance() {
if (uniqueInstance == null) {
uniqueInstance = new Singleton();
}
return uniqueInstance;
}
// other useful methods here
}
Our constructor is declared private; only Singleton can instantiate this class!
The getInstance() method gives us a way to instantiate the class and also to return an instance of it.
Of course, Singleton is a normal class; it has other useful instance variables and methods.

Singleton: Structure



Singleton: Participants & Collaborations

- **Participants:**
  - Singleton - creates a unique (static) instance of itself.
  - Note that Singleton defines a getInstance() operation that lets clients access its unique instance.
  - Note that getInstance() is a class operation (i.e. a static method in C++ or Java).
- **Collaborations:**
  - Clients can only access the Singleton instance through the getInstance() operation

Singleton: Implementation so far ….

- Create a public static method that is solely responsible for creating the single instance of the class
  - the getInstance method can be overridden so that it assigns a subclass of Singleton to this variable.
- Should have a **private or protected** constructor, so clients that try to instantiate Singleton directly cause compiler errors
- The instance is not constructed until the first request to access it.
- If sub classing is not required then the constructor should be **private** for greater encapsulation

Singleton: Implementation

- Q2 - In which conditions the above code could lead to multiple instances?

```
public static Singleton getInstance() {
    if(INSTANCE == null) { // 1
                INSTANCE = new Singleton(); // 2
    }
    return INSTANCE; // 3
}
```

  - **If one thread enters getInstance() method and finds the INSTANCE to be null at step 1 and**
    - **enters the IF block and before it can execute step 2 another thread enters the method and**
    - **executes step 1 which will be true as the INSTANCE is still null,**
    - **then it might lead to a situation (Race condition) where both the threads executes step 2 and create two instances of the Singleton class.**

Singleton: Implementation

- Q3 - How do we solve the issues in multi-threading?

  1. **Lazy instantiation using double locking**

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

- Q4 - What is the drawback of synchronizing the getInstance() method?
  - It will decrease the performance of a multithreaded system as only single thread can access the getInstance method at a time. Also synchronizing a method has its own overheads.

Singleton: Implementation

2. An eagerly created instance

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

– **JVM creates the instance when the class is loaded before any thread accesses the instance**

Singleton

- Q4 – What is meant by lazy instantiation and eager instantiation?

Lazy initialization happens when the initialization of the Singleton class instance is delayed till its static getInstance() is called by any client program.

Eager initilization happens when we eagerly initialize the private static variable to hold the single instance of the class at the time of its declaration.

- Q5 - Give some use cases where we can use Singleton pattern?
- Classes implementing Thread Pools, database Connection pools, caches, loggers etc are generally made Singleton so as to prevent any class from accidently creating multiple instances of such resource consuming classes.

Singleton: Known Uses & Related Patterns

- **Known Uses:**
  - Singleton is useful whenever having more than one instance of a class is undesirable.
  - Singleton can be generalised:
    - e.g. a 'coupleton' has exactly two instances, with a single point of access to these instances;
    - a 'key' can be used to specify which instance is required.
- **Related Patterns:**
  - Many patterns can be implemented using the Singleton pattern. See Abstract Factory, Builder, and Prototype, all of which are creational patterns.

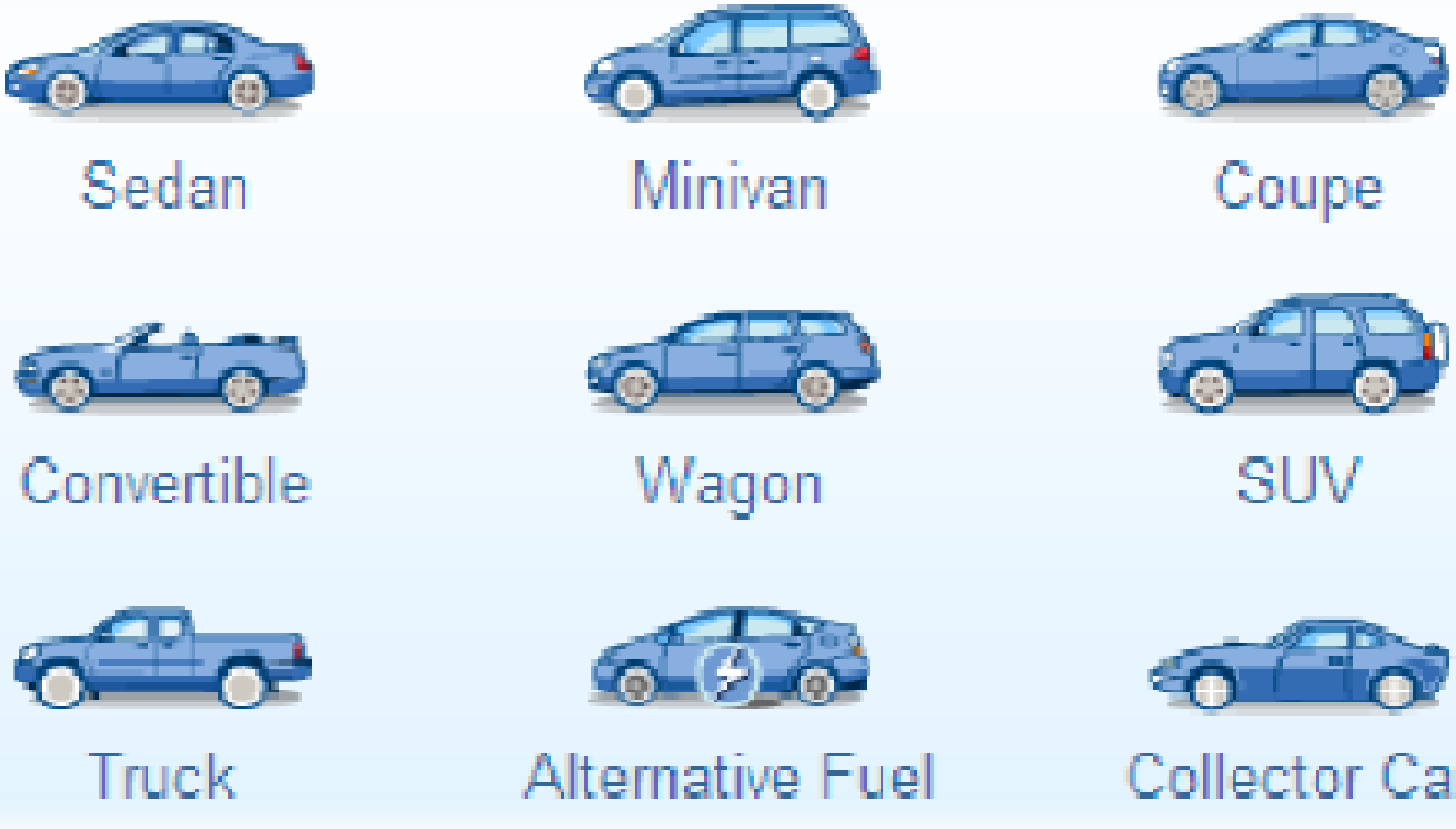
Activity 1

- Write a sample java code to implement Coupleton design pattern.

Creational Patterns

Abstract Factory

Abstract Factory : Problem



Abstract Factory : Intent and Motivation

Intent

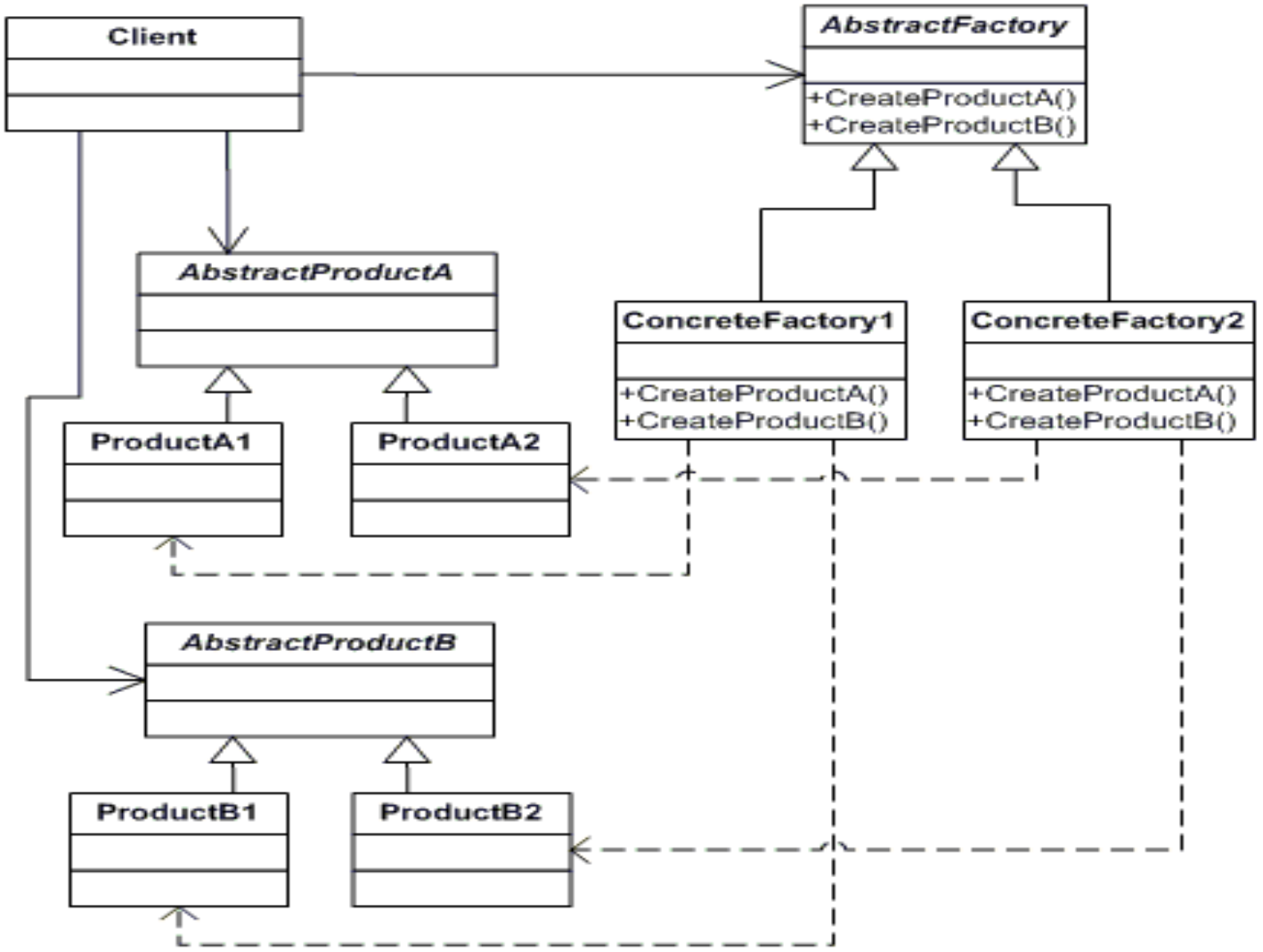
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Motivation

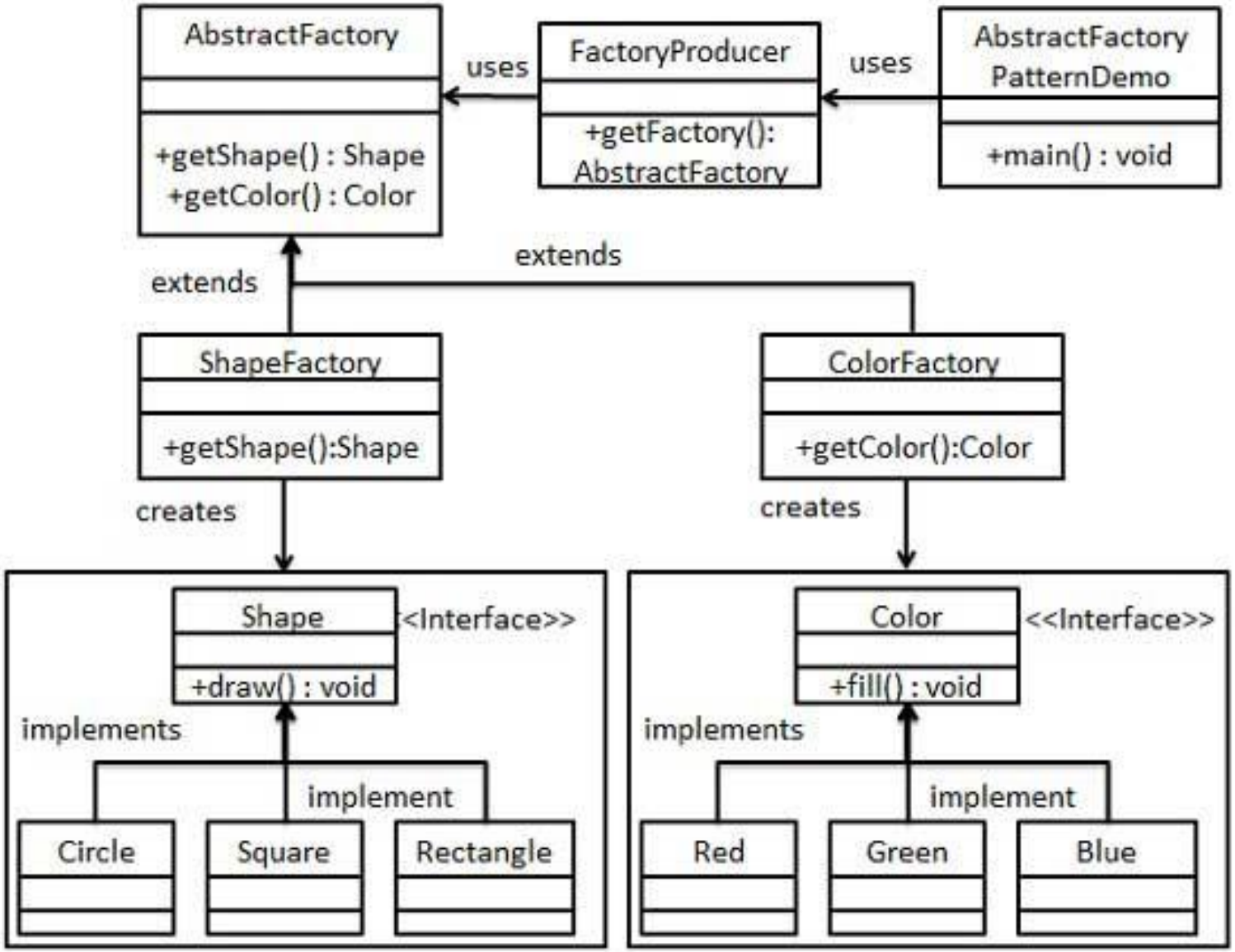
To have multiple looks and feels in a system. eg:

User Interface toolkits with different widgets having different looks and feels

Abstract Factory : Structure



Abstract Factory : Example 1



Abstract Factory : Example 1

- Step 1 - Create an interface for Shapes

  public interface Shape { void draw(); }

- Step 2 -Create concrete classes implementing the same interface.

  public class Rectangle implements Shape {

  public void draw() { System.out.println(“Rect"); } }

  similarly Square and circle should be implemented

- Step 3 - Create an interface for Colors.
- Step4 -Create concrete classes implementing the same interface *(for Red, Green and Blue)*

Abstract Factory : Example 1

- Step 5 - Create an Abstract class to get factories for Color and Shape Objects.

```
public abstract class
    AbstractFactory {
    abstract Color
    getColor(String
    color);
    abstract Shape
    getShape(String
    shape) ; }
```

- Step 6 - Create Factory classes extending AbstractFactory to generate object of concrete class based on given information.

```
public class ColorFactory extends AbstractFactory {
    public Shape getShape(String shapeType){
    return null; }
    Color getColor(String color) {
      if(color == null){ return null; }
      if(color.equalsIgnoreCase("RED")){ return new
                Red(); } else
      if(color.equalsIgnoreCase("GREEN")){ return
                new Green(); } else
      if(color.equalsIgnoreCase("BLUE")){ return new
                Blue(); } return null; } }
```

Abstract Factory : Example 1

- Step 7 - Create a Factory generator/producer class to get factories by passing an information such as Shape or Color.

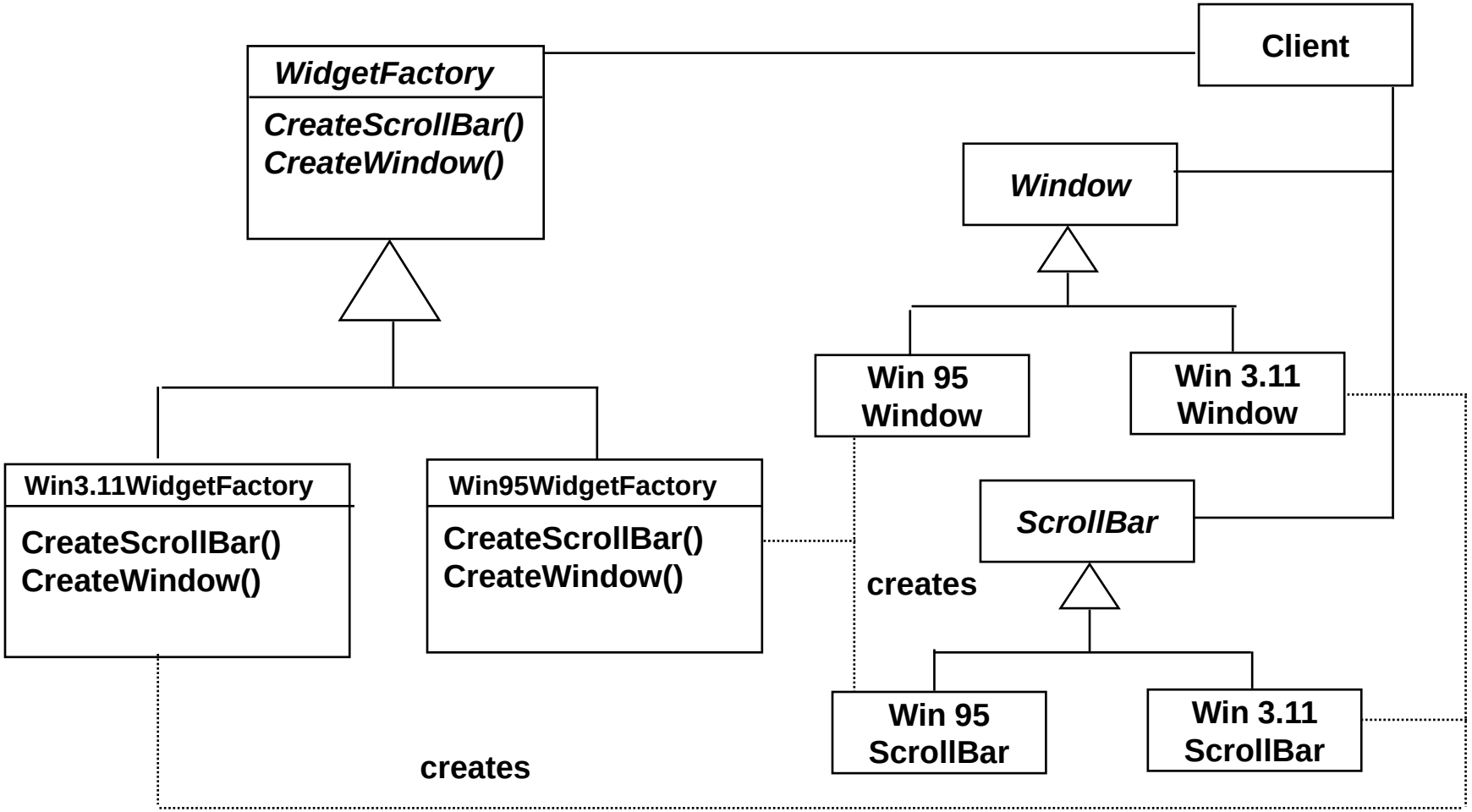
```
public class FactoryProducer {
    public static AbstractFactory getFactory(String choice){
        if(choice.equalsIgnoreCase("SHAPE")){
                return new ShapeFactory(); }
        else
        if(choice.equalsIgnoreCase("COLOR")){
                return new ColorFactory(); }
        return null; }
}
```

Abstract Factory : Example 1

- Step 8 - Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type.

```
public class AbstractFactoryPatternDemo {
    public static void main(String[] args) {
        //get shape factory
        AbstractFactory shapeFactory =
        FactoryProducer.getFactory("SHAPE");
        //get an object of Shape Circle
        Shape shape1 = shapeFactory.getShape("CIRCLE");
        //call draw method of Shape Circle
        shape1.draw();
        ……..
```

Abstract Factory : Example 2



Abstract Factory : Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract products
- **ConcreteFactory** (Win3.1WidgetFactory , Win95 WidgetFactory)
  - implements the operations to create concrete product objects
- **AbstractProduct** (Window, Scrollbar)
  - declares an interface for a type of product object
- **Product** (Win95window, Win95Scrollbar)
  - defines a product object to be created by the corresponding concrete factory
  - implements the AbstractProduct interface
- **Client**
  - uses interfaces declared by AbstractFactory and AbstractProduct classes

Abstract Factory : Applicability

- When a system should be independent of how the products are created, composed and presented,
- When a system requires configuration of multiple families of products,
- When families of product objects need to work together,

Abstract Factory : Implementation

- If you want to construct instances of a class, where the class is selected at run-time, then
  1. Create one abstract factory class for each existing class (or group of related classes) you wish to create.
  2. Have a polymorphic “create instance” method on each abstract factory class, conforming to a common method signature, used to create instances of the corresponding class – ConcreteFactory classes

3.Store and pass around instances of the abstract factory class to control selection of the class to create

4. A given application needs only one instance of a ConcreteFactory classes per product family. So implement it using Singleton.

Abstract Factory : Collaborations

- Single instance of a ConcreteFactory class is created at runtime which instantiates product objects with different implementations.

- AbstractFactory allow its subclasses (ConcreteFactory classes) to create product objects.

Abstract Factory : Consequences

- Isolates clients from implementations of the classes.
- ConcreteFactory class appears only once in an application when it is initiated. By changing ConcreteFactory class different product configurations can be changed easily. (whole product family changes at once)
- Provide Consistency
- Supporting for new kind of products is difficult because the AbstractFactory interface fixes the products that can be created.

Structural Patterns

Flyweight

Flyweight : Problem

- **Consider writing a application like Microsoft Word. It allows user to mix all kinds of objects - sequences of characters, pictures, tables, etc.**
- **Given the complexity, implementers must consider modeling every character as an Object. This sounds right from an object-oriented perspective, however in reality, is too expensive. Imagine allocating memory for every character in a large document!**

Flyweight (Object Structural)

- **Intent:**
  - Use sharing to support large numbers of fine-grained objects efficiently.
- **Motivation:**
  - Some applications could benefit from using objects throughout their design, but a naïve implementation would be prohibitively expensive.

Flyweight: Background

- An image is a 2-D array of pixels, where each pixel specifies an intensity value that typically ranges from 0 (dark) to 255 (light).
  - A monochrome TV image with dimensions 768 by 576 contains over 1 million pixels.
  - A colour TV image of this size contains over 3 million pixels.
- Image processing applications typically use objects to represent images,

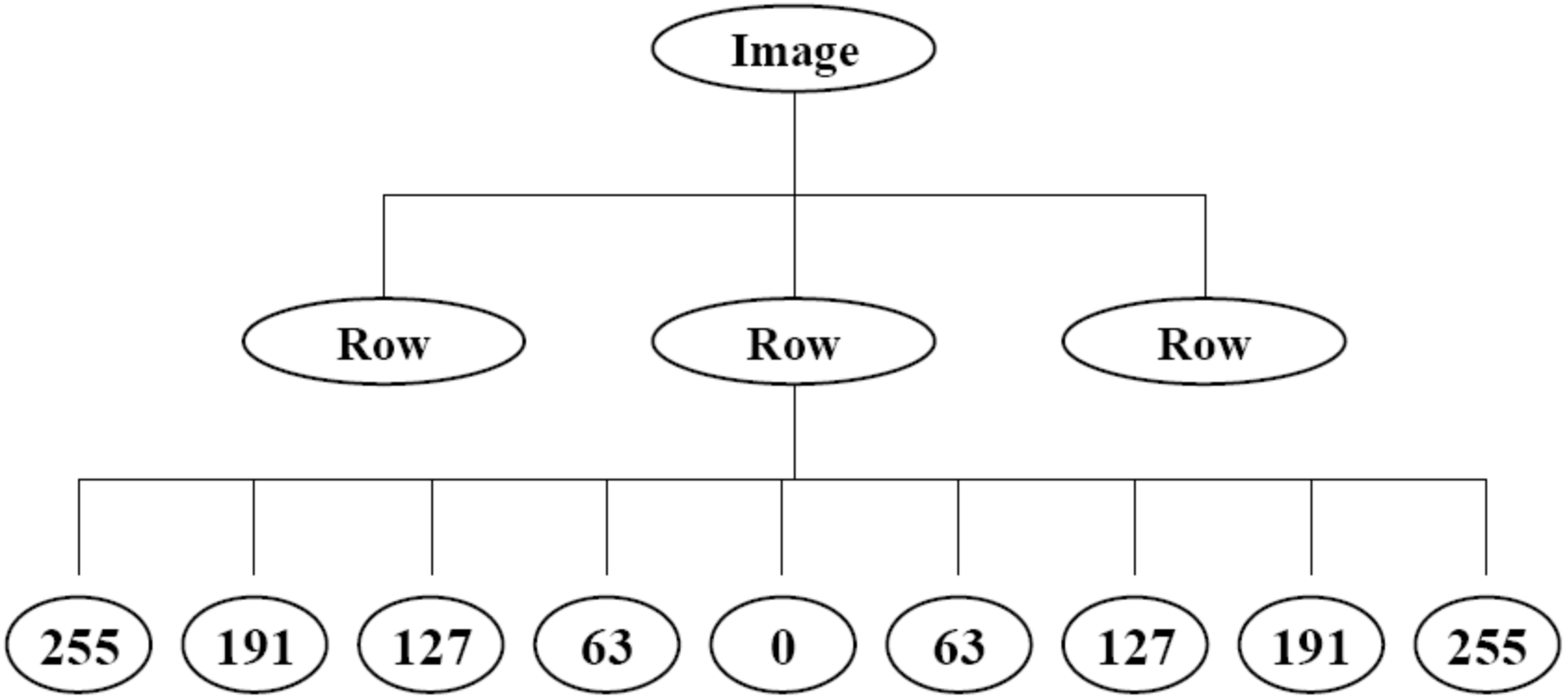
  but stop short of using an object for each pixel in an image.

Flyweight: Motivation

- The problem with a pure OO design is its cost:
  - even small images require many pixels objects, which consume lots of memory and may incur unacceptable run-time overheads.
  - **The flyweight pattern describes how to create a large number of objects without prohibitive cost.**
  - A flyweight is a shared object that can be used in multiple contexts simultaneously:
    - a flyweight acts like an ordinary object in each context . it is indistinguishable from a normal unshared object.

Flyweight: Logical Structure

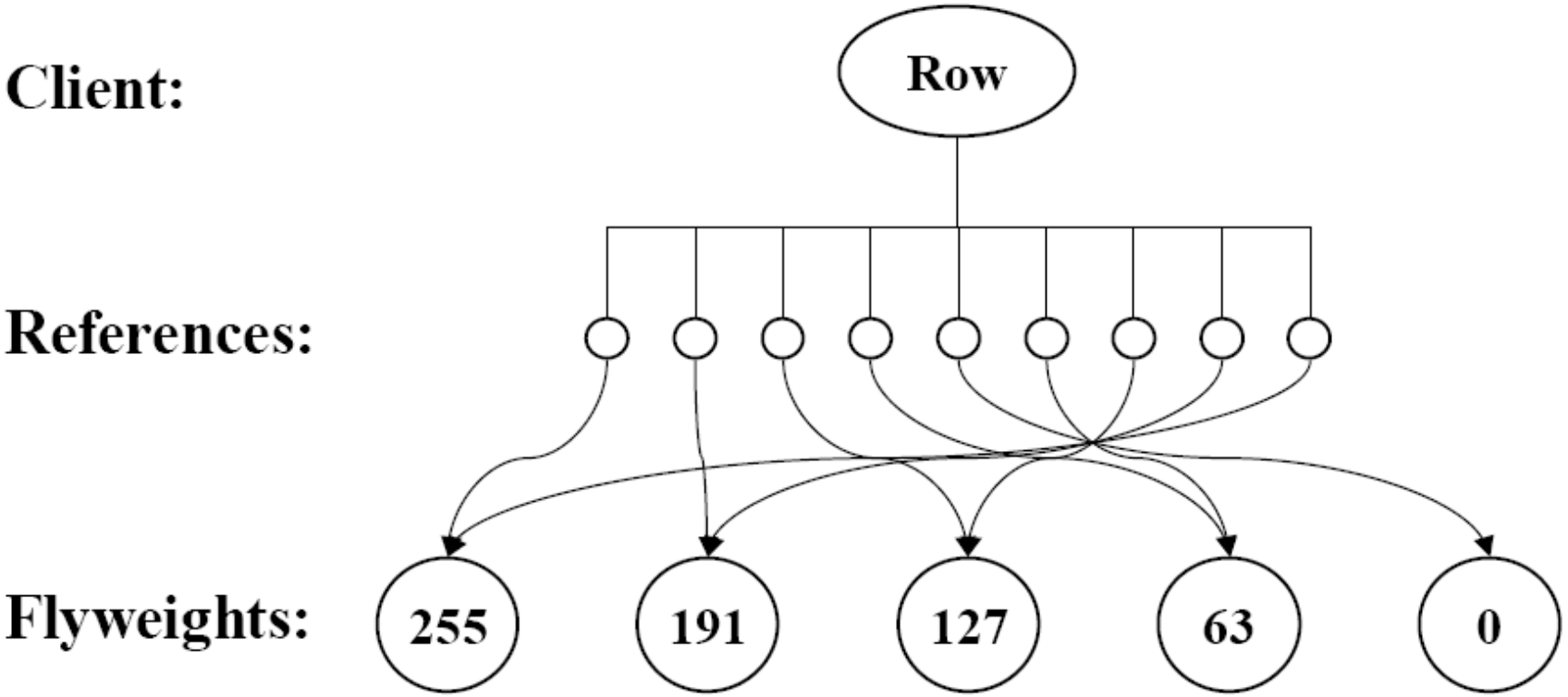
- Logically, every pixel in an image is an object:



Flyweight: Physical Structure

- Physically, there is one shared flyweight object representing all pixels of a given intensity:
  - a flyweight representing a pixel only stores the intensity, not the location of the pixel;
  - it appears in different contexts throughout the image.
- Pixels with the same characteristics all refer to the same instance in a shared pool of flyweight pixels:
  - e.g. the first and last pixels on the previous slide.

Flyweight: Physical Structure



Flyweight: Physical Structure

- The Flyweight pattern relies upon a distinction between intrinsic and extrinsic state.

- **Intrinsic state** is stored in the flyweight; it consists of information that is independent of the flyweight's context, thereby making it sharable .
  - e.g. the intensity of the pixel.

- **Extrinsic state** varies with the context, and so cannot be shared; it must be passed to the flyweight whenever it is needed .
  - e.g. the row and column position of the pixel.

Flyweight: Physical Structure

- Clients supply the context-dependent information that the flyweight needs to draw itself.
  - e.g. an image row knows where its pixels should draw themselves;
  - the row can therefore pass each pixel a location as part of a draw() message.

- The number of unique pixels (256) is far less than the number of pixels in an image:
  - Hence, the total number of objects is substantially less than it would be in a naive implementation.

Activity 2

- Imagine that you need to create a Car Management System to represent all of the cars in a city. You need to store the details about each car (make, model, and year) and the details about each car's ownership (owner name, tag number, last registration date).

If you are to use the Flyweight design pattern for the above system, identify the intrinsic and extrinsic states of the system.

Flyweight: Applicability

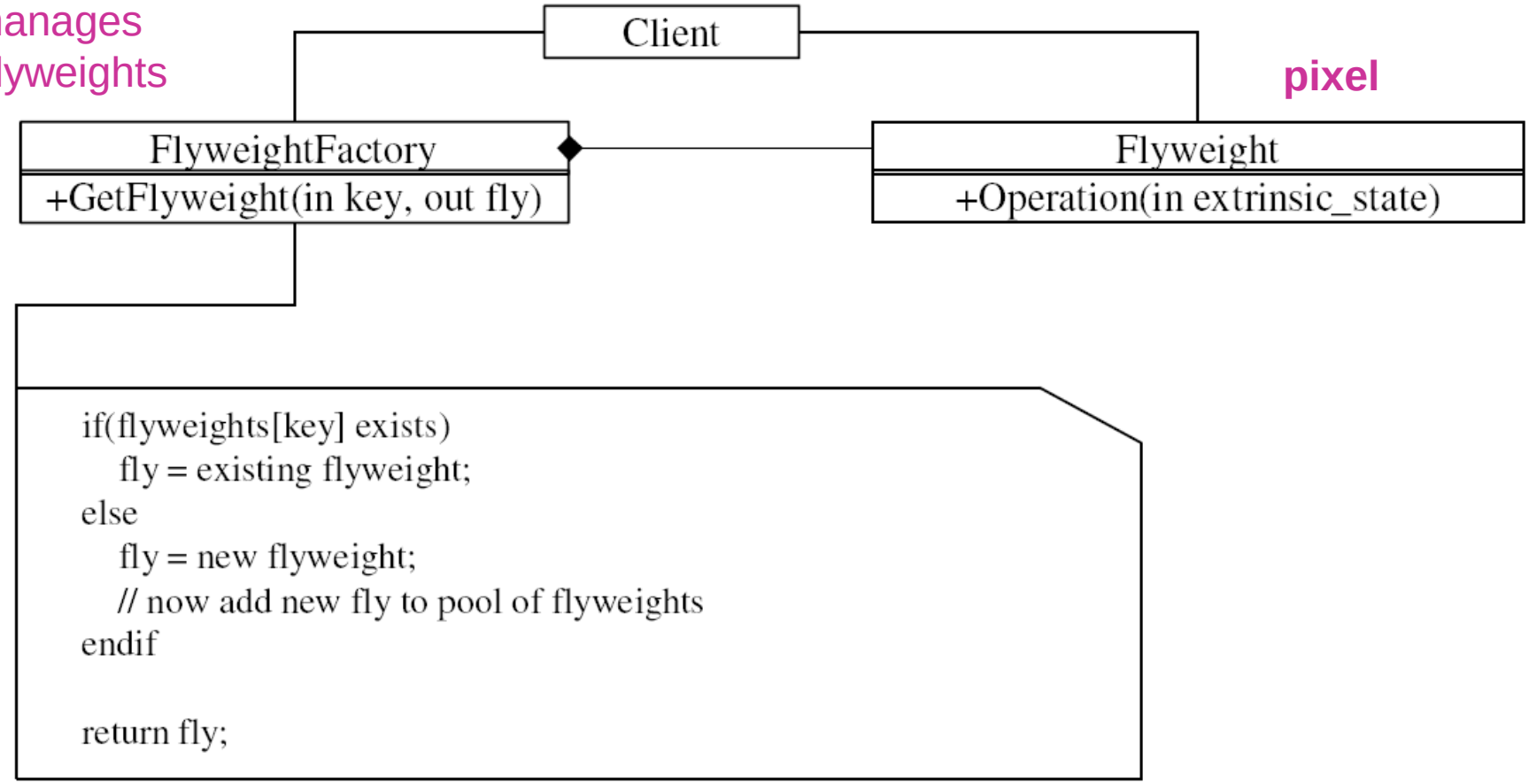
- Apply flyweight when all of the following are true:
  - a naive implementation uses a large number of objects;
  - storage costs are high simply because of the very large number of objects;
  - most object state can be made extrinsic;
  - many groups of objects may be replaced by relatively few shared objects;
  - the application does not depend on object identity .
  - identity tests may return true for logically distinct objects because flyweights are physically shared.

Flyweight: (Simplified) Structure

Creates & manages Flyweights

Row

pixel



Activity 3

- Write the code for the Car Management System in Activity 2 to create flyweight objects using a factory.

Flyweight: Participants

- **Flyweight (Pixel):**
  - Declares an interface through which flyweights can receive and act on extrinsic state.
  - Implements the Flyweight interface and adds storage for intrinsic state, if any.
- **Flyweight Factory:**
  - Creates and manages flyweight objects.
  - Ensures that flyweights are shared properly.
- **Client (ImageRow):**
  - Maintains a reference to one or more flyweights.
  - Computes or stores the extrinsic state of flyweights.

Flyweight: Collaborations

- When a client requests a flyweight object the FlyweightFactory object supplies an existing instance if possible, or creates one if necessary.
- Clients should not instantiate Flyweights directly:
  - they must obtain Flyweights exclusively from the FlyweightFactory to ensure that duplicates are not created, and that Flyweights are shared properly.
- FlyweightFactory is related to Singleton:
  - a FlyweightFactory for pixels would create and manage exactly 256 objects (one per intensity value).

Flyweight: Consequences

- Flyweights may introduce run-time costs associated extrinsic state, especially if it was formerly stored as intrinsic state.
- Storage savings depend on several factors:
  - The reduction in the total number of instances from sharing.
  - The amount of intrinsic state per object (savings increase with the amount of shared state).
  - Whether extrinsic state is computed or stored.
- The more flyweights are shared, the greater the storage savings.
- The greatest savings occur when the objects use substantial quantities of both intrinsic and extrinsic state, and the extrinsic state can be computed rather than stored.

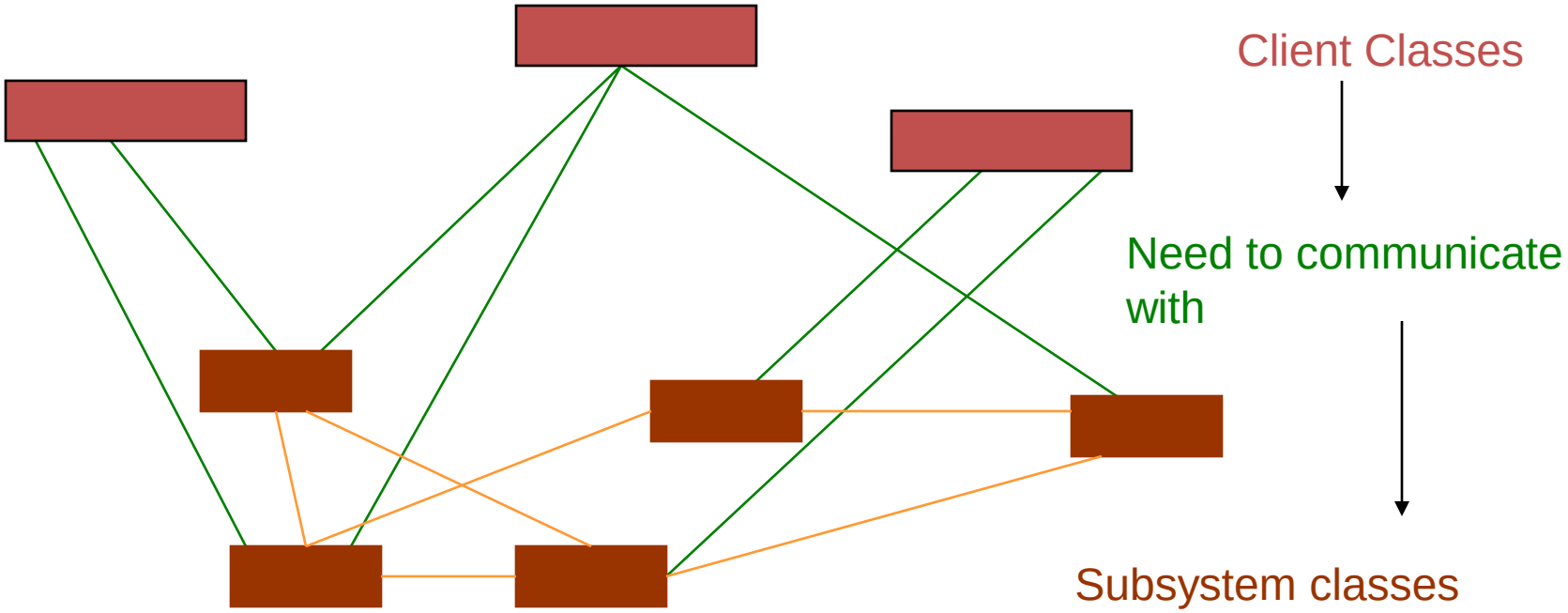
Structural Patterns

Façade

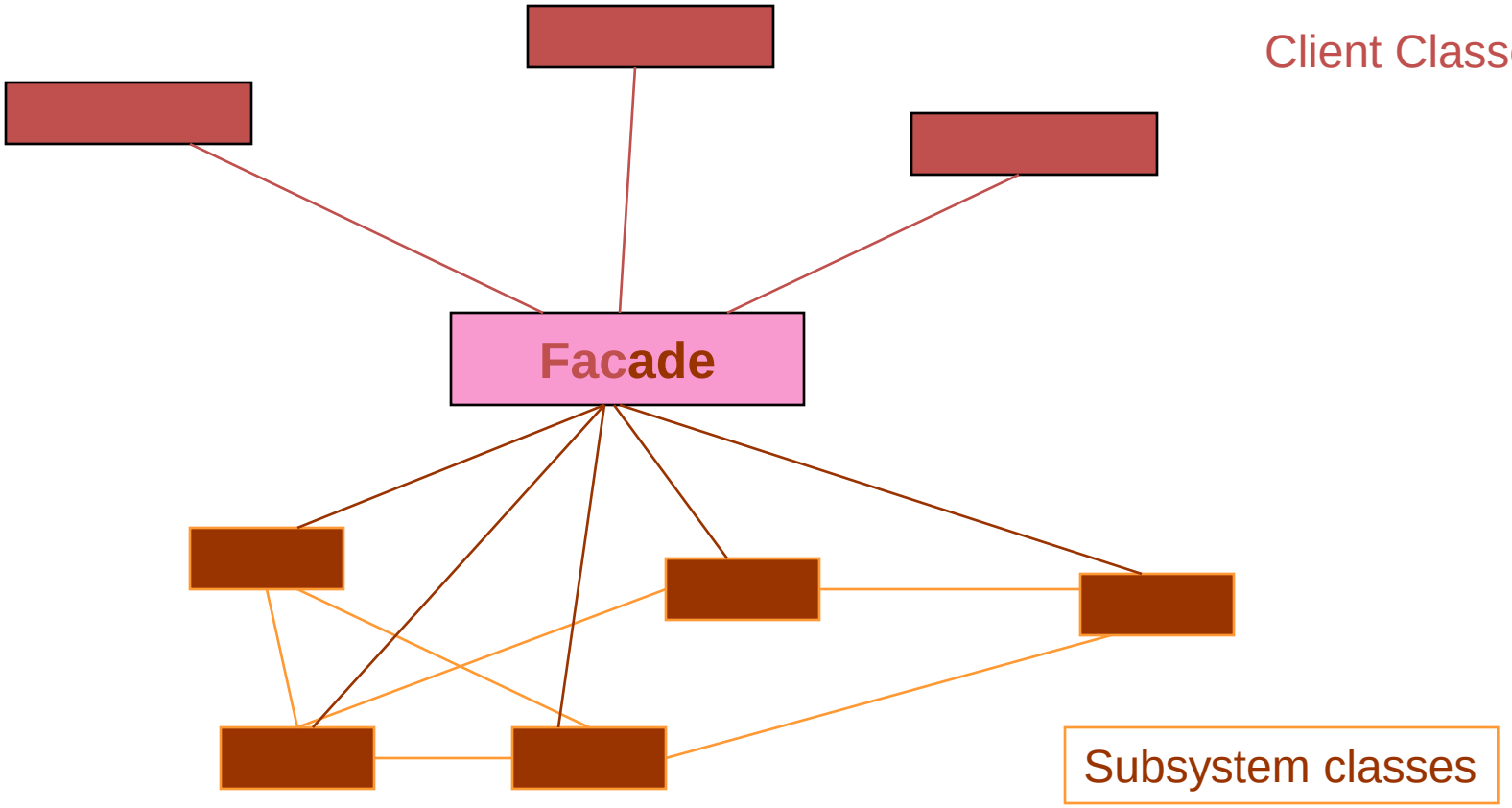
The Façade Design Pattern

- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use
- Motivation
  - Simplifying system architecture by unifying related byt different interfaces via a Façade object that shield this complexity from clients.

Facade Pattern: Problem



Facade Pattern: Solution



Facade:
Encapsulating Subsystems

Name: Facade design pattern

Problem description:

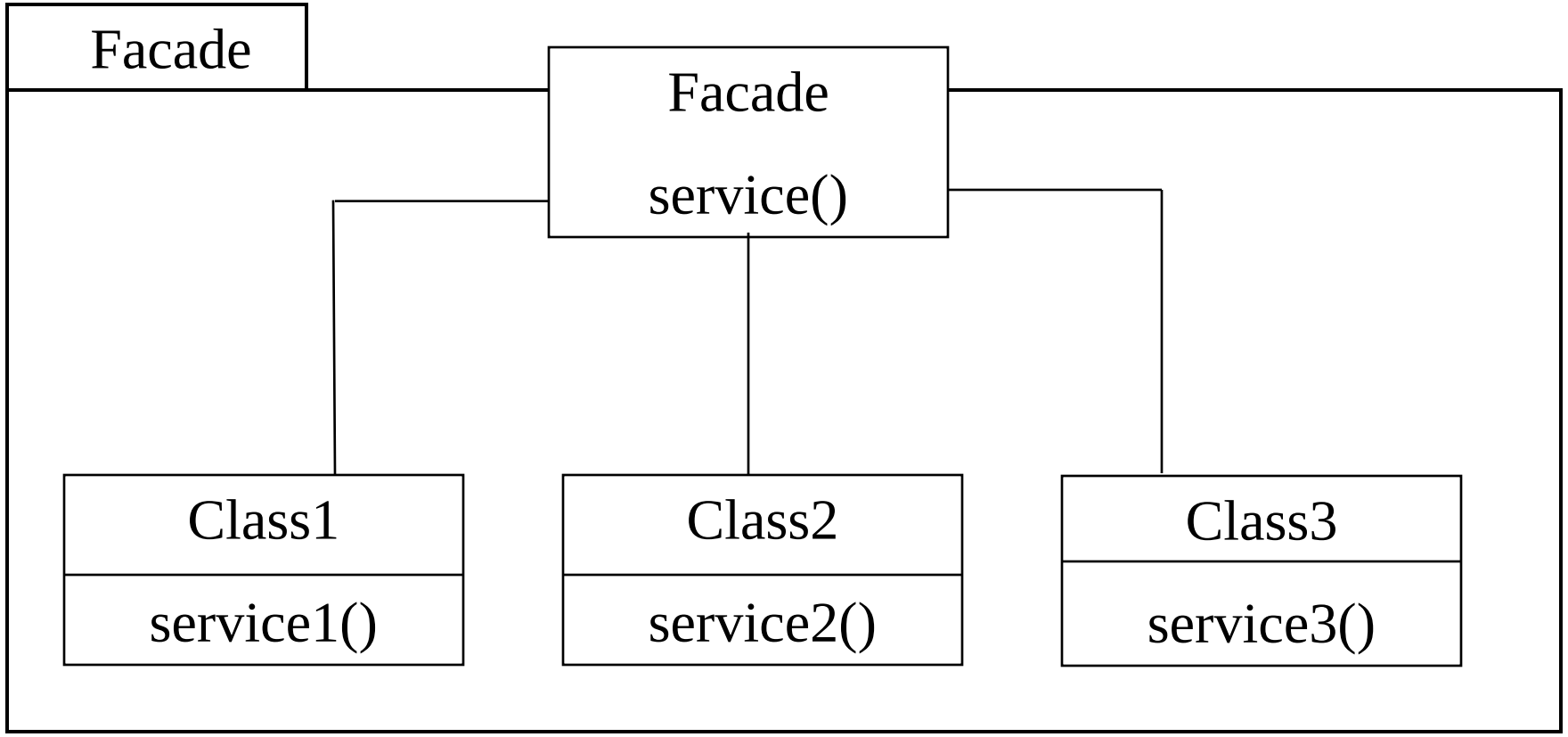
Reduce coupling between a set of related classes and the rest of the system.

Solution:

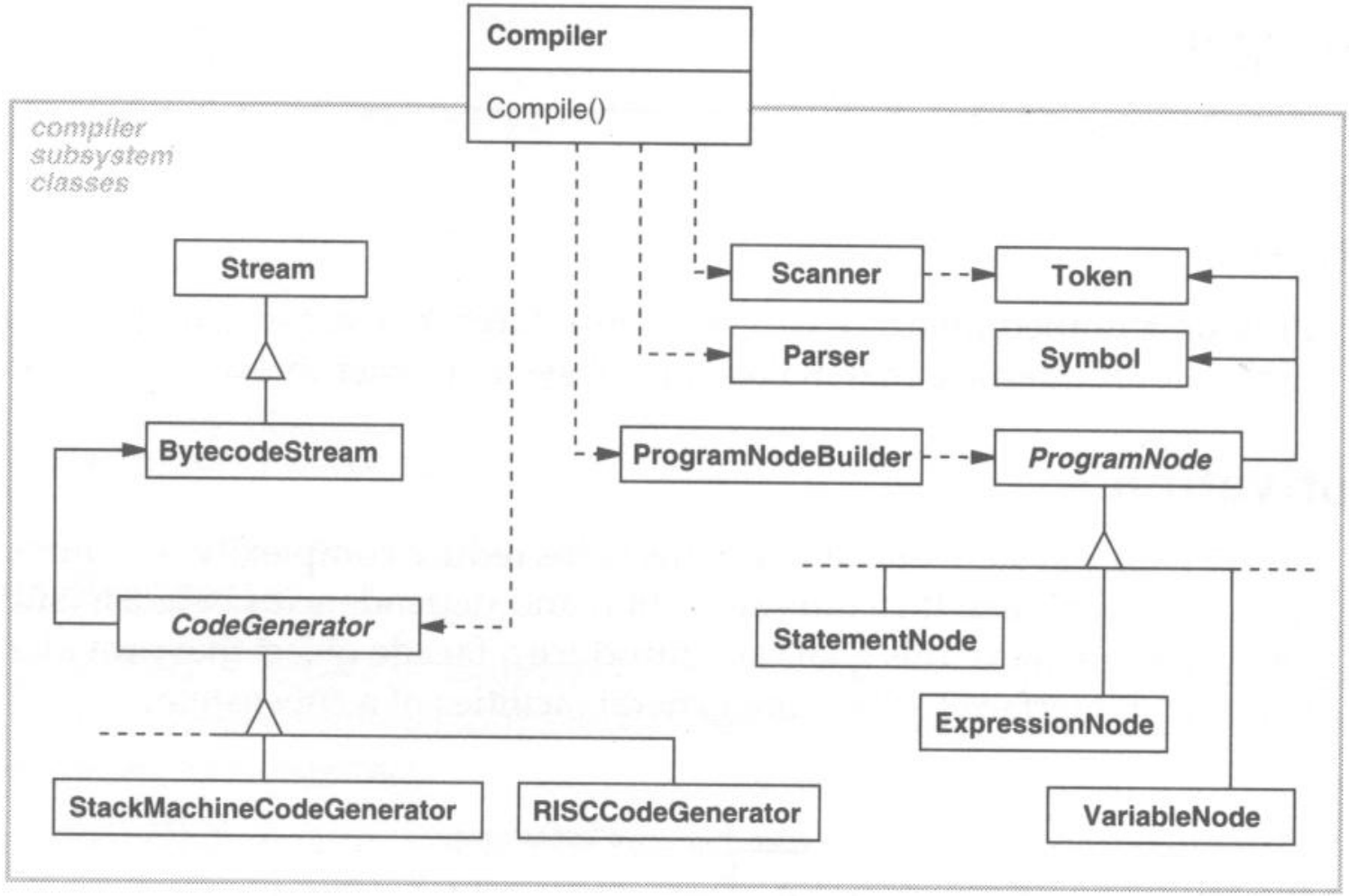
A single **Facade** class implements a **high-level interface** for a subsystem by invoking the methods of the lower-level classes.

Example. A Compiler is composed of several classes: LexicalAnalyzer, Parser, CodeGenerator, etc. A caller, invokes only the Compiler (Facade) class, which invokes the contained classes.

Facade:
Class Diagram



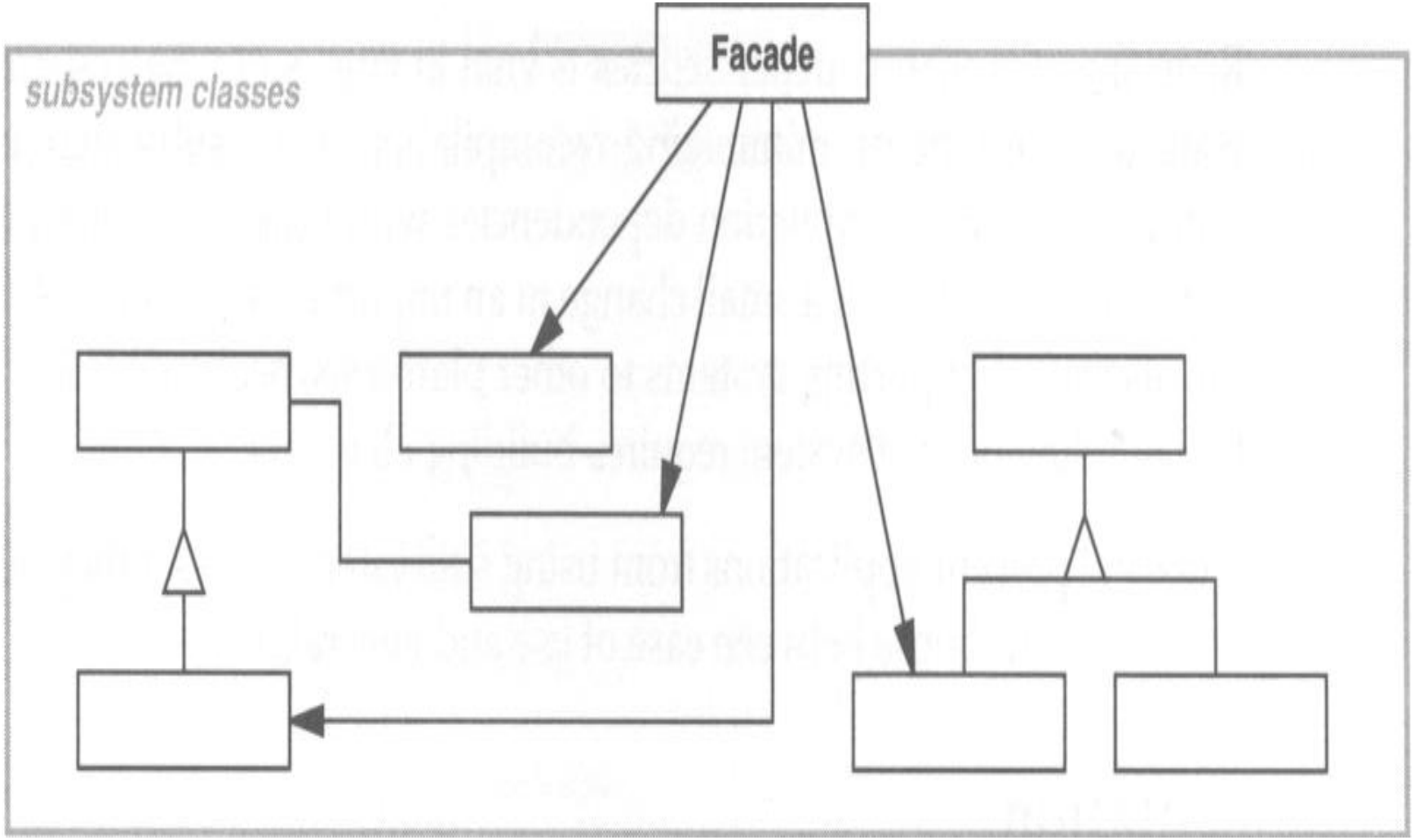
Facade - Motivation



Façade Applicability

- Contrary to other patterns which decompose systems into smaller classes, Façade combines interfaces together to unified same one.
- Separate subsystems from clients via yet another unified interface to them.
- Levels architecture of a system, using Façade to separate the different subsystem layers of the application.

Façade Structure



Façade Participants

- Façade (Compiler)
  - Knows which subsystem classes are responsible for a request
  - Delegates client requests to appropriate subsystem objects
- Subsystem classes (Scanner, Parser, ProgramNode, etc.)
  - Implement subsystem functionality
  - Handle work assignment by the Façade object
  - Have no knowledge of the Façade – I.e., no reference upward.

Activity 4

- Every morning when Susan goes for jogging, she has to make the following changes in my Windows phone device:
  - Turn off the wifi
  - Switch on the Mobile Data
  - Turn on the GPS
  - Turn on the Music
  - Start the Sports-Tracker
- And after coming back from jogging, following needs to be done from her part:
  - Share Sports tracker stats on twitter and facebook
  - Stop the Sports Tracker
  - Turn off the Music
  - Turn off the GPS
  - Turn off the Mobile Data
  - Turn on the wifi

- Assume that you can use controllers like GPSController, MobileDataController, MusicController, WifiController, to control the required change.
- What design patterns would you recoomend for this scenario?
- Implement the design pattern you suggested using a sample code.

Façade Collaboration

- Clients
  - Sending requests to the Façade, which forwards them appropriately to the subsystem components.
  - Façade may do some adaptation code
- Separation
  - Clients do not need to know, or ever use the subsystem components directly.

Façade Consequences

- Shielding Clients
  - Reduces the number of objects clients need to deal with
- Promotes weak coupling
  - Between subsystems and clients
  - Although(!) components in the subsystem may be strongly coupled.
  - Help layer the system (also prevents circular dependencies)
  - Reduces compilation dependencies in large systems – more efficient application building (compilation)
- But – permits direct use
  - In case individual components offer meaningful service to clients – the Façade mediates, but does not block access.

Structural Patterns

Decorator

Also Known as Wrapper

As a Modern Girl Role

As a Male Character

As a Traditional Character

Decorator : Intent and Motivation

Intent

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

Motivation

- We want to add individual responsibilities to individual objects, not to a class.

Decorator : Applicability

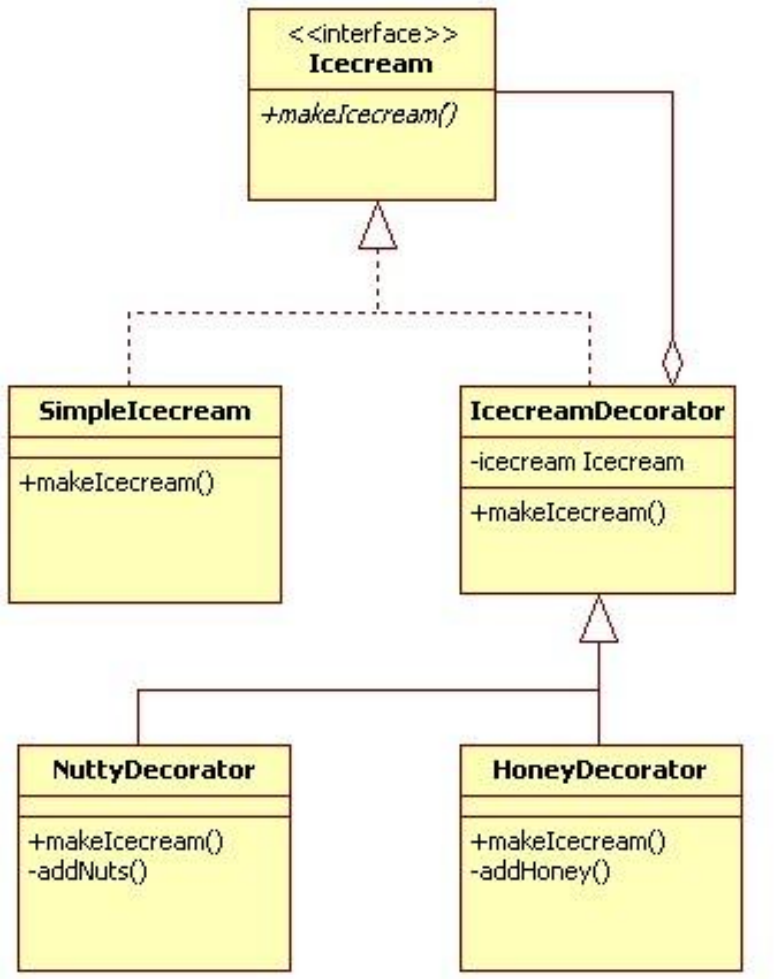
Use Decorator,

- To add responsibilities to individual objects dynamically and transparently (without affecting others)
- For responsibilities that can be withdrawn
- When extension by sub classing is impractical.

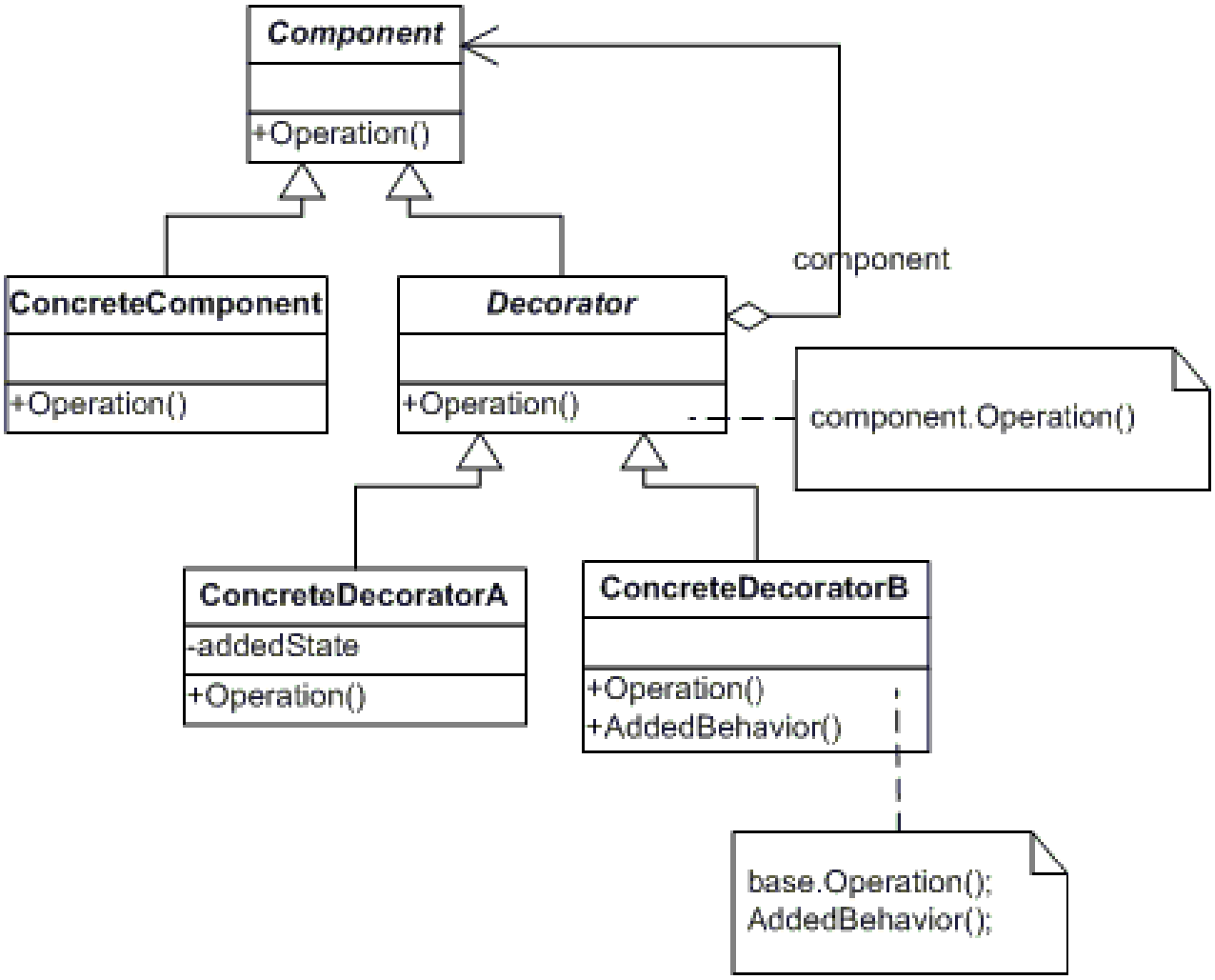
Eg: class definition hidden/unavailable for subclassing

Decorator pattern steps

1. Create an interface for the class to be decorated – Icecream
2. Implement that interface with basic functionalities – SimpleIceCream
3. Create an abstract class that contains an attribute type o the interface. - IcecreamDecorator
4. The concrete decorator clas will add its own methods. – NuttyDecorator, HoneyDecorator



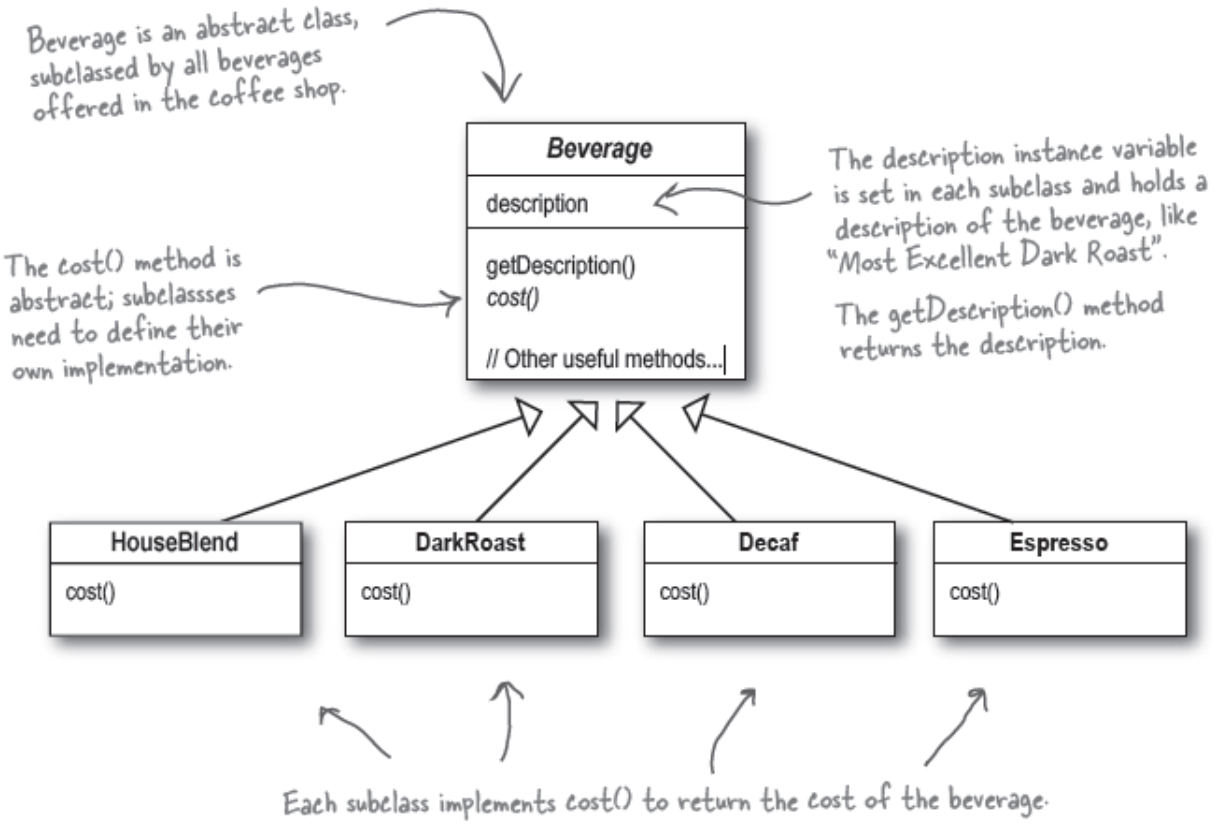
Decorator : Structure



Activity 5

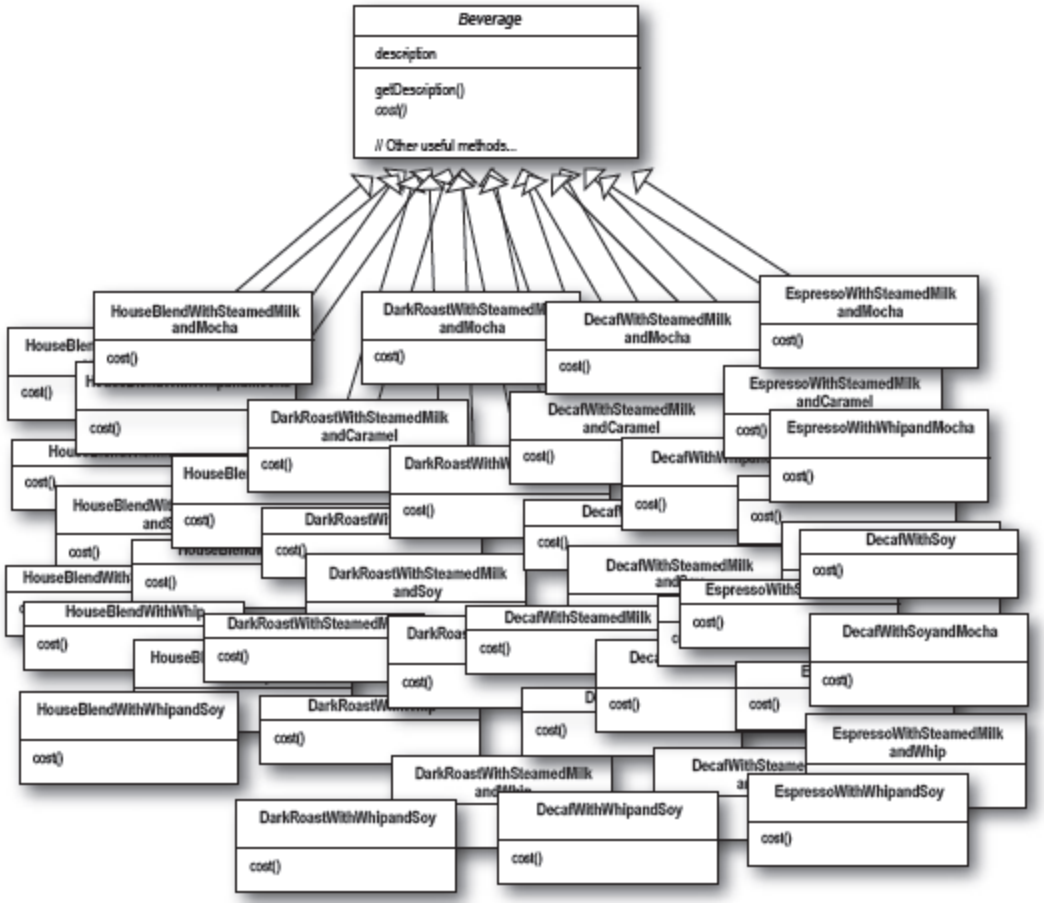
Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

When they first went into business they designed their classes like this...



Activity 5 (cont.)

- In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these. Here's their first attempt...
- Use the Decorator pattern to make this design better.



Decorator : Participants

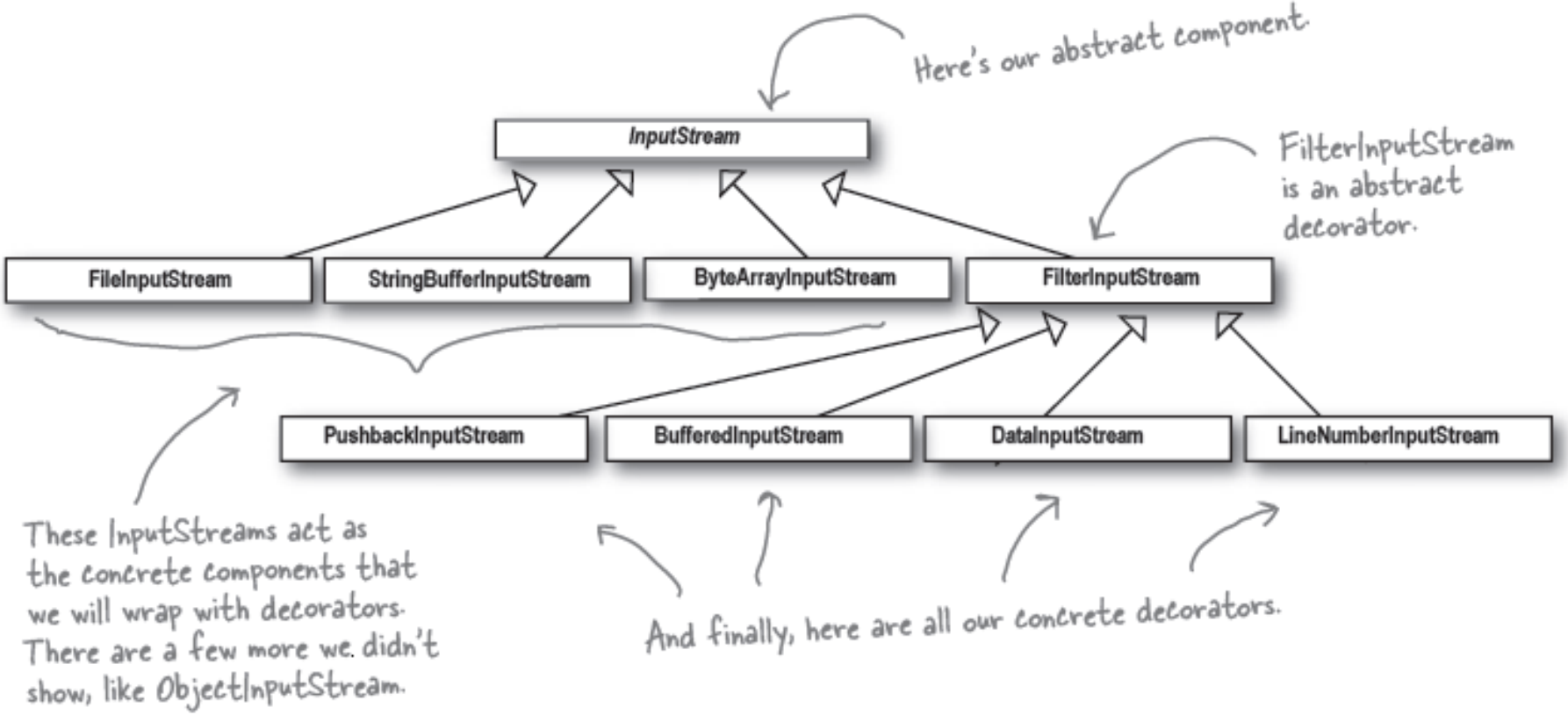
- Component (Beverage)- Defines the interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent (HouseBlend, darkRost..)- Defines an object to which additional responsibilities can be attatched.
- Decorator (CondimentDecorator) - Maintains a reference to a Component object that defines an interface that confirms to Component's interface
- ConcreteDecorator (Milk, Mocha…)- Adds Responsibilities to the component

Decorator : Collaborations

- Decorator forward requests to Component object.

- Before and after the request can perform additional operations – optional

Decorator in Java I/O classes



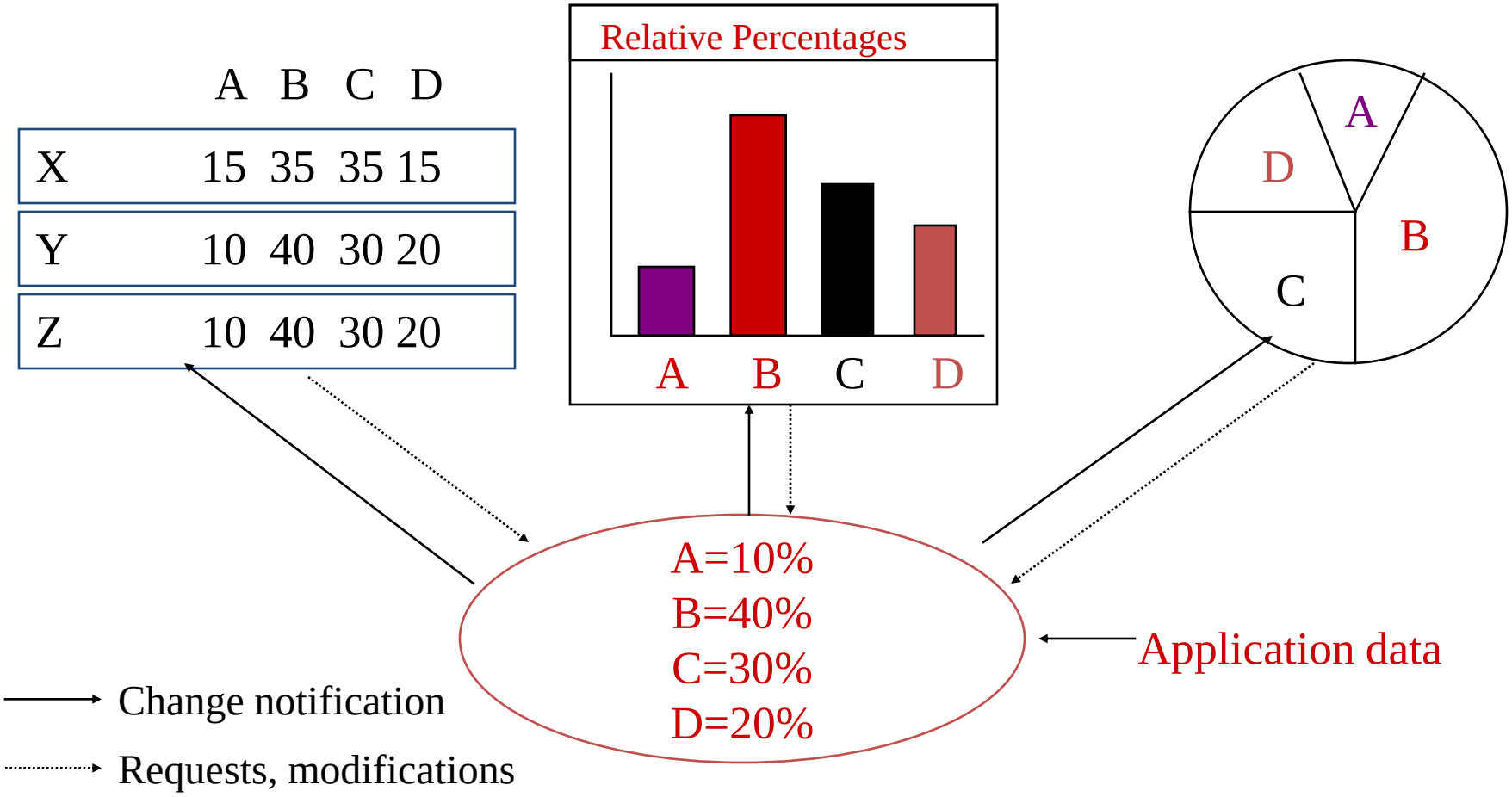
Decorator : Consequences

- A more flexible way to add or remove responsibilities at runtime.
- Rather than having a high complex class, we can define a simple class and add functionality incrementally with Decorator objects.
- A Decorator object is not as same as a component object. So we cant rely on object identity when using decorators.
- Large number of little objects.

Behavioural Patterns

Observer

Patterns by Example:
Multiple displays enabled by Observer



Observer (Object Structural)

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Observer has
  - Purpose - Behavioural
  - Scope - Objects
- Example - different views of data in a spreadsheet.
  - Table view
  - Bar graph view
  - Pie chart view

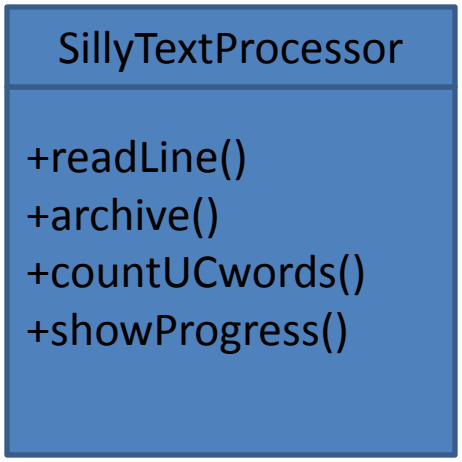
Observer : Applicability

- Use Observer in the following situations:
  - When an abstraction has two aspects, one dependent upon the other. Encapsulating these aspects in separate objects lets you vary and reuse then independently
  - When a change to one object requires changing others, and you don’t know how many objects need to be changed
  - When an object should be able to notify other objects without making assumptions about what these objects are, i.e. you don’t want them tightly coupled

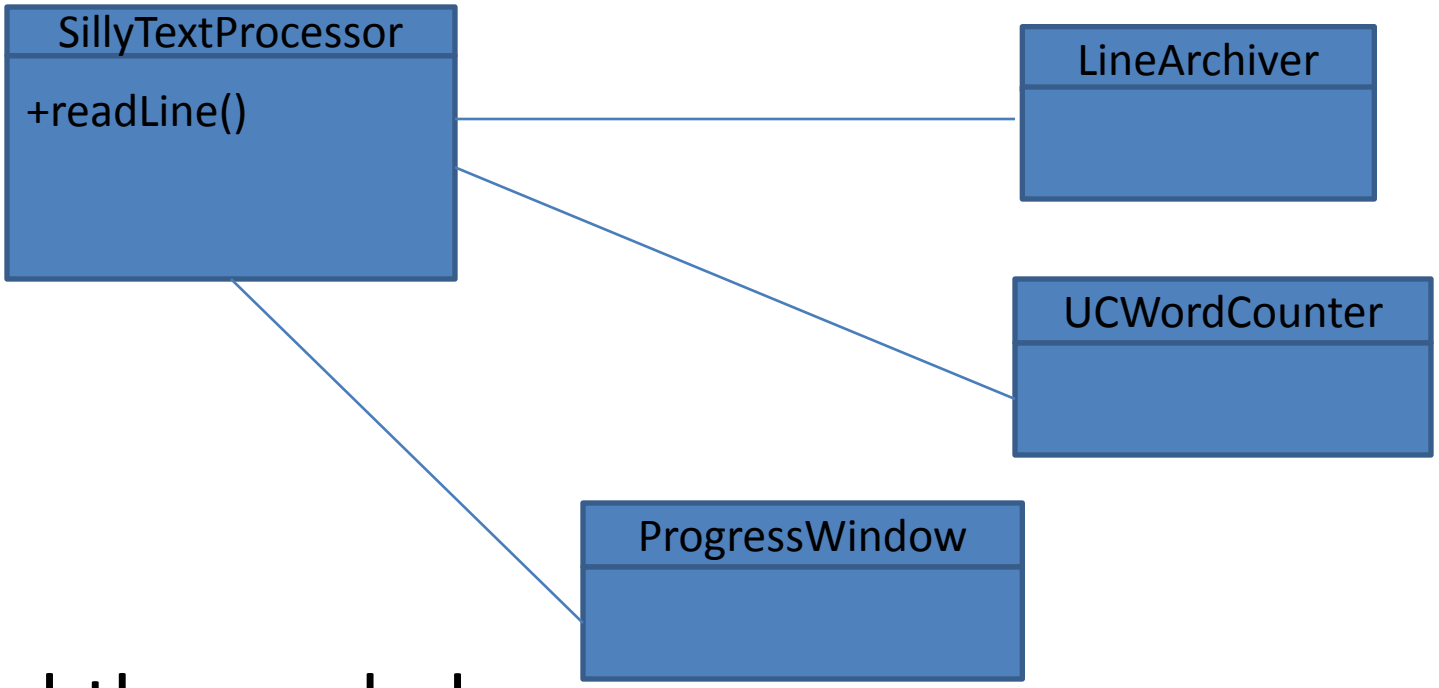
Observer : Example

- A Silly Text Processor:
  - Counts the number of words that start with an uppercase letter
  - Save the lines to a file
  - Shows the progress (e.g., the number of lines processed)

A Bad Design

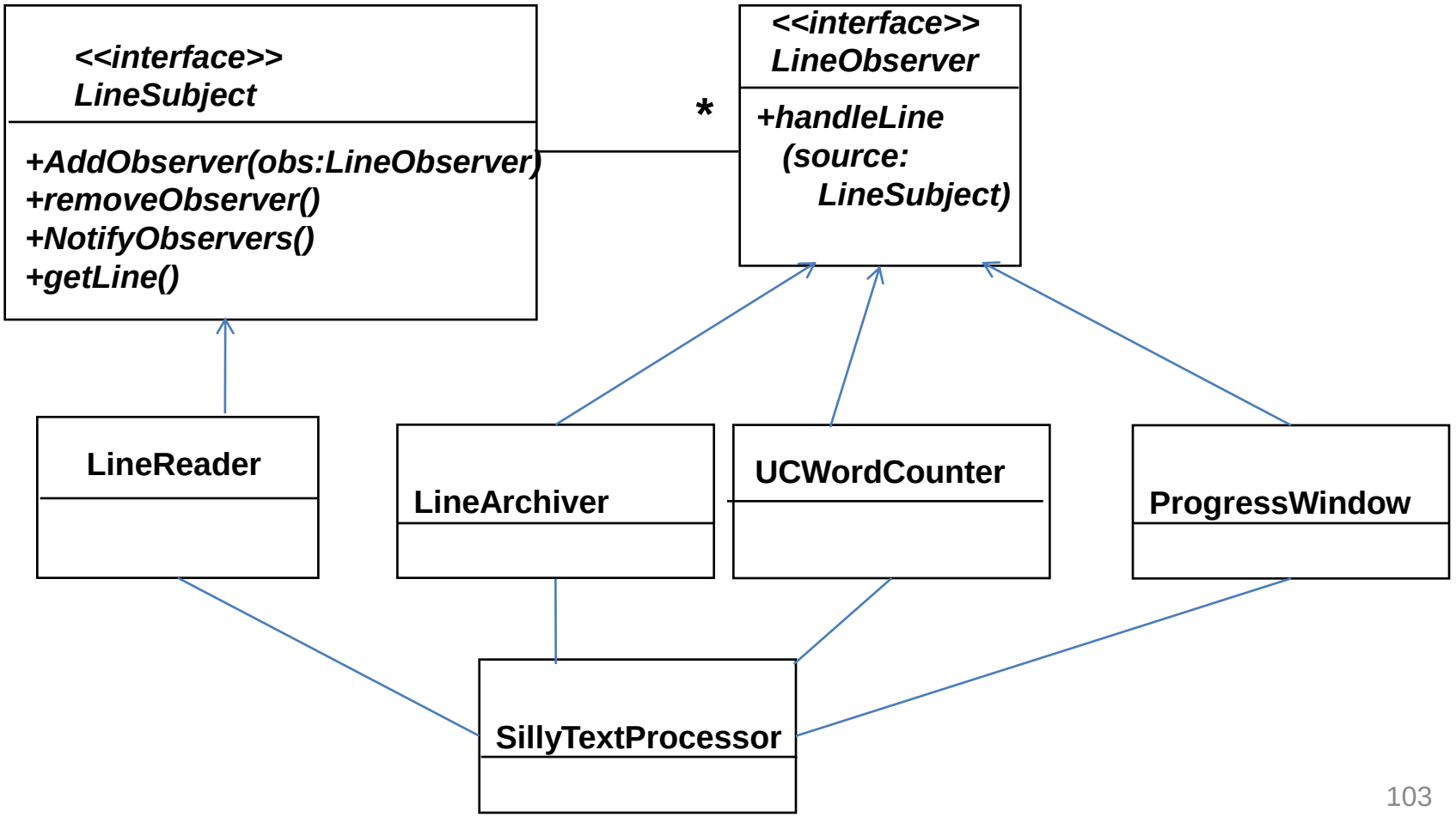

SillyTextProcessor
+readLine()
+archive()
+countUCwords()
+showProgress()

A better design

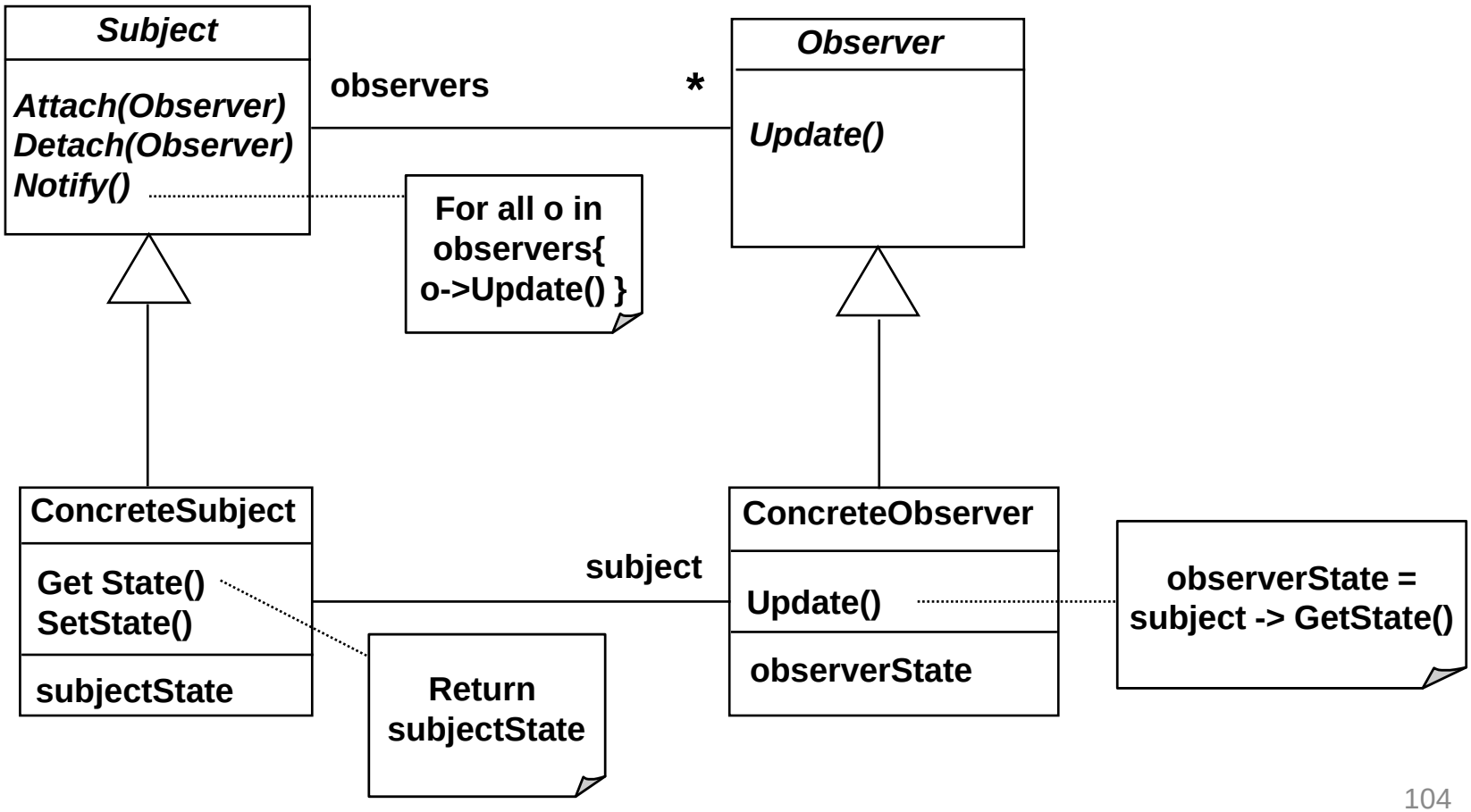


- Tightly coupled

Best design


<<interface>>
LineSubject
+AddObserver(obs:LineObserver)
+removeObserver()
+NotifyObservers()
+getLine()
*
<<interface>>
LineObserver
+handleLine
(source:
LineSubject)
LineReader
LineArchiver
UCWordCounter
ProgressWindow
SillyTextProcessor

The Structure of Observer



Observer : Participants

- Subject
  - Knows its observers
  - Any number of Observer objects may observe a subject
  - provides an interface for attaching and detaching Observer objects
- Observer
  - defines an updating interface for objects that should be notified of changes in a subject
- ConcreteSubject
  - stores state of interest to ConcreteObserver objects
  - sends a notification to its observers when its state changes
- ConcreteObserver
  - maintains a reference to ConcreteSubject object
  - stores state that should stay consistent with subject's
  - implements the Observer updating interface to keep its state consistent with the subject's.

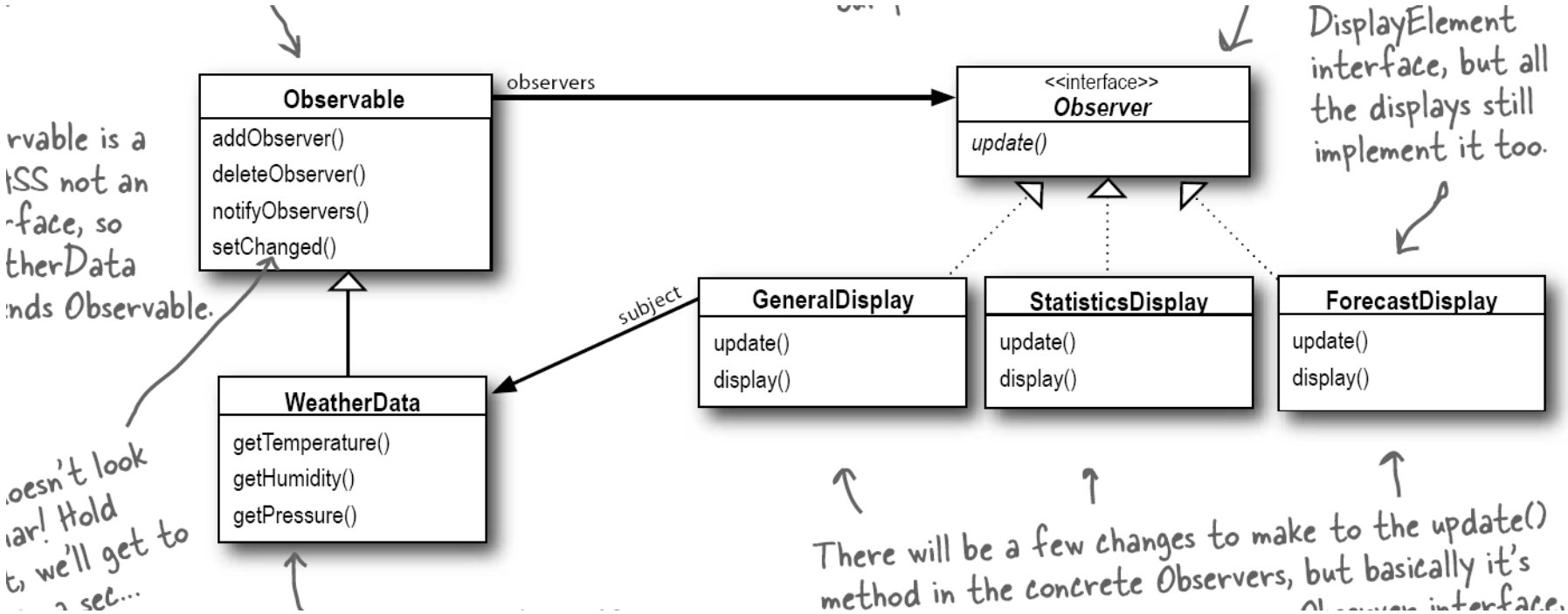
Observer : Consequences

- Abstract Coupling between Subject and Observer
  - All a subject knows is it has a list of observers, each conforming to an abstract and simple interface
  - Subject doesn't know the concrete class of any observer
  - Coupling is abstract and minimal
  - Subject and Observer can belong to different layers of the system as they are not tightly coupled
- Support for Broadcast Communication
  - Subject need not specify the receiver(s) for its message
  - Message is sent to all interested parties who are subscribed
  - Only responsibility of subject is to notify observers
  - Can add or remove observers at will

Observer : Consequences (2)

- Unexpected Updates
  - As observers are unaware of each other, they cannot know the cost of changing the state of the subject
  - A seemingly innocuous operation on the subject could cause a cascade of updates to observers and their dependent objects
  - Dependency criteria which are not well-defined or maintained often lead to spurious updates
  - These can be hard to track down, especially with a simple Update protocol, which doesn't provide details on what changed in the subject

Observer - example



References

Head First Design Patterns by Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra (Author), Elisabeth Robson