SLIIT

Discover Your Future

Software Engineering (IT2020) - 2025

Lecture 7 - Design Patterns

SLIIT
FACULTY OF COMPUTING

Session Outcomes

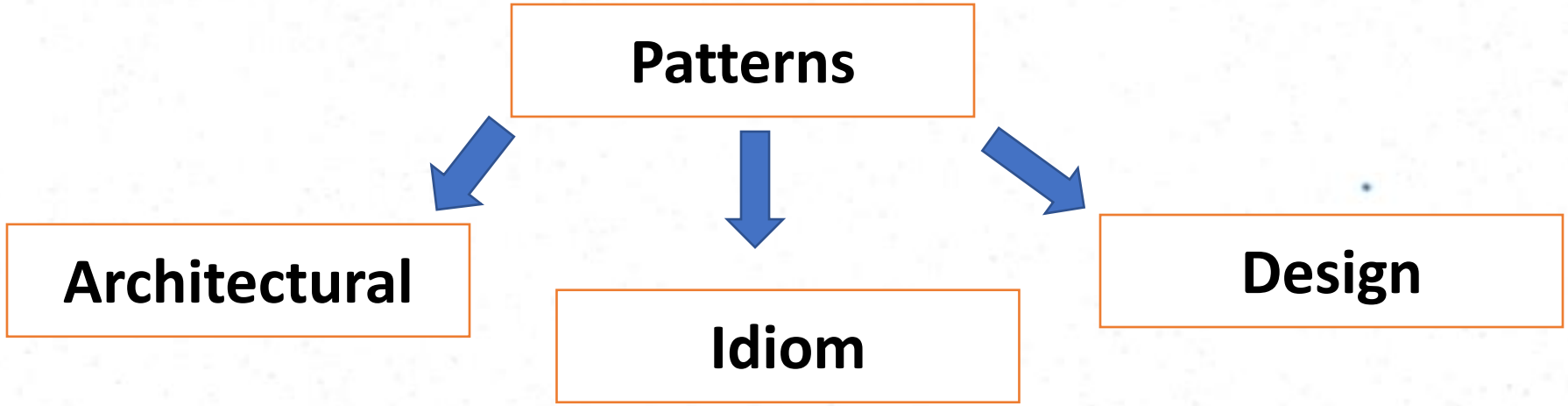
- Introduction to Design Patterns
- Creational Patterns
  - Singleton (Covered in OOP)
- Structural Patterns
  - Façade
  - Decorator
  - Fly weight
- Behavioural Patterns
  - Observer

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.**
- Patterns are discovered, they are not invented.
- Pattern is a template solution to a recurring design problem

Categories of Patterns

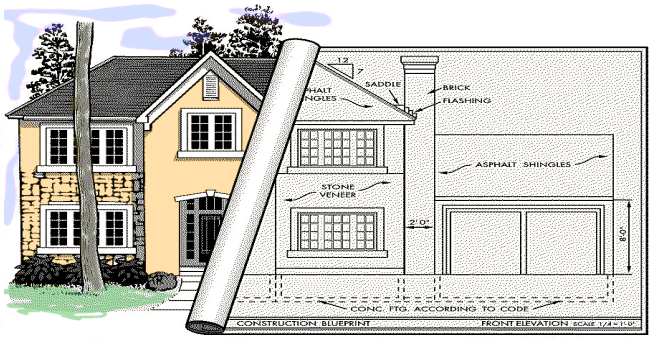
- Software Engineering Patterns can be classified in various ways .
- Classification of patterns used in Software Engineering is shown below.

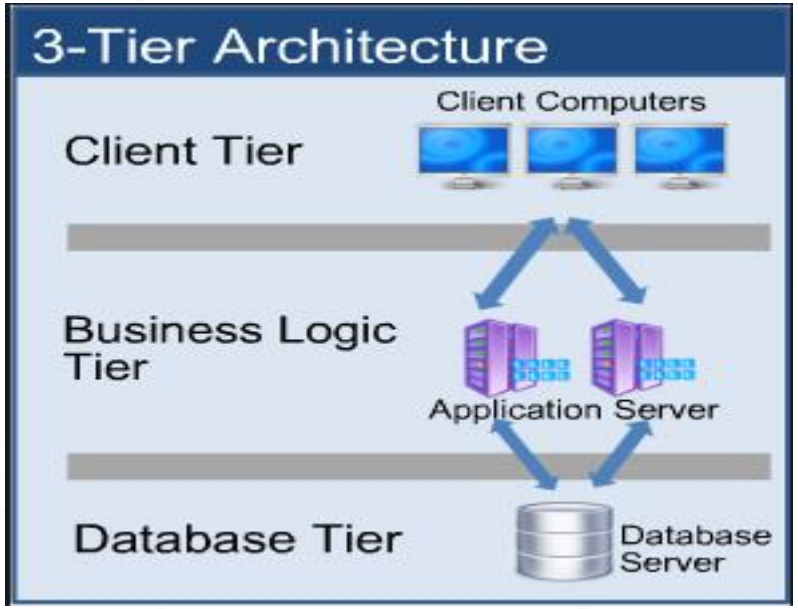


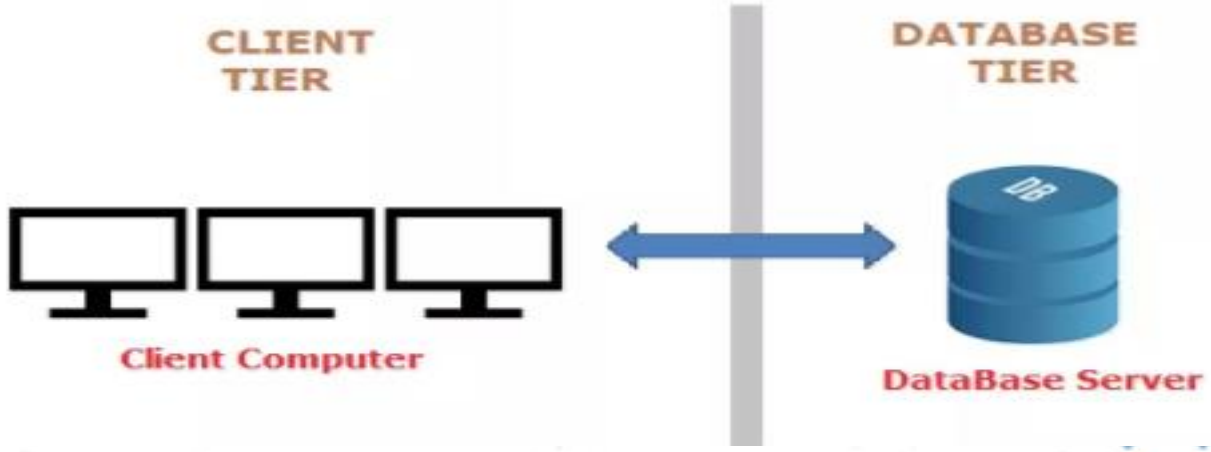
Architectural Patterns

- These patterns provide an overall architecture (a large scale design) for a software system.
- *Examples :*

3-Tier Architecture, 2-Tier Architecture





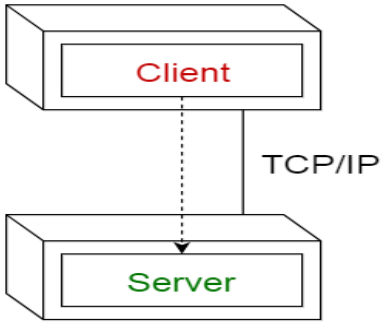


Architectural patterns Examples

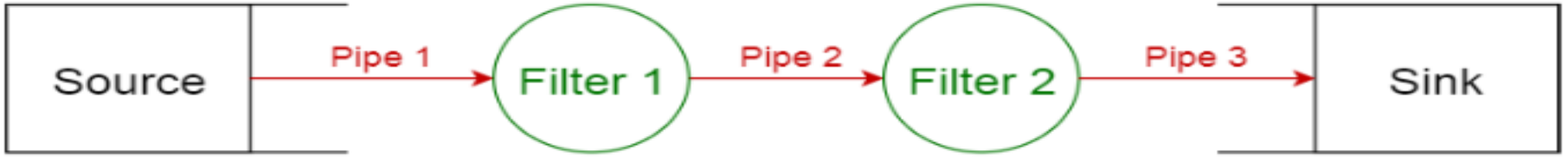
- **Client/server architecture**:

  Online applications such as email, document sharing and banking.

- **Pipeline architecture -**Compilers



Client-server pattern



Idioms

- 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.
  - Idioms represent low-level patterns.
- Eg: **Swapping values between variables**

C/C++/Java	Perl	Python

temp = a;
a = b;
b = temp;
	($a, $b) = ($b, $a);	a, b = b, a

Design Patterns

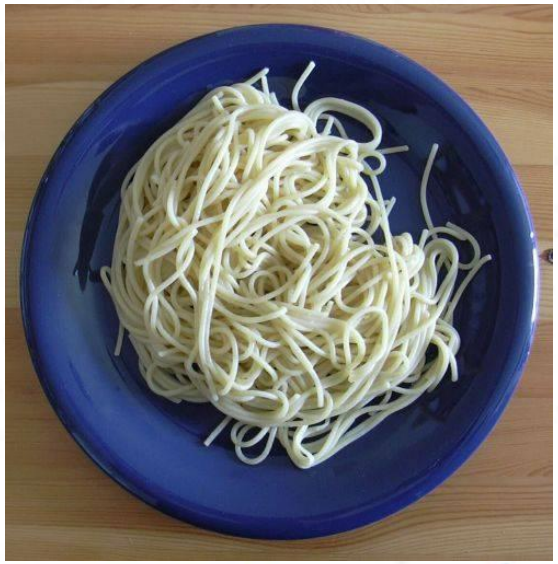
- Quote from Christopher Alexander
    - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (GoF,1995).

- Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work. (They have been proven)
- They are reusable. Patterns provide a ready-made solution that can be adapted to different problems as necessary.

Why use Design Patterns?

Code Reuse is good – Software developers generally recognize the value of reusing code

- reduces maintenance
- reduces defect rate (if reusing good code)
- reduces development time



Elements of a Design Pattern

- A pattern has four essential elements (GoF)
  - Name
    - Describes the pattern
    - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
  - Problem
    - Describes when to apply the pattern
    - Answers - What is the pattern trying to solve?

Elements of a Design Pattern (cont.)

- Solution
  - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
  - Results of applying the pattern
  - Benefits and Costs
  - Subjective depending on concrete scenarios

How to Describe Design Patterns fully

This is critical because the information has to be conveyed to peer developers in order for them to be able to evaluate, select and utilize patterns.

- A format for design patterns
  - Pattern Name and Classification
  - Intent
  - Also Known As
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Patterns

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	

Design Patterns Classification

A Pattern can be classified as

- Creational – Object creation (These patterns will be covered in OOP)
- Structural – Relationship between entities
- Behavioral –Communication between objects

Patterns

Design Patterns		
Creational	**Structural**	**Behavioral**
Singleton (Covered in OOP) Abstract Factory Builder Prototype	**Decorator** **Façade** **Flyweight** Adapter Bridge Composite Proxy	**Observer** Chain of Responsibility Command Iterator Mediator Memento State Strategy Visitor

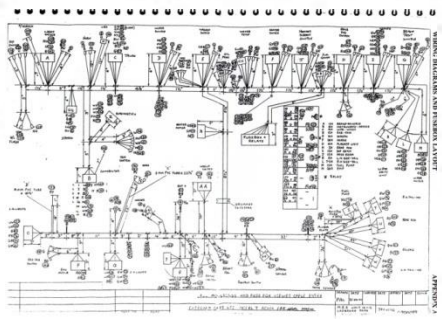
Design Pattern Classification

Creational Patterns

Concerns the process of object creation

Behavioural Patterns

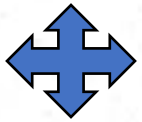
Characterize the ways in which classes/objects
interact and distribute responsibility



Final Product

Structural Patterns

Deals with the composition of classes /objects



Behavioral Patterns

Observer Pattern

Observer Definition

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

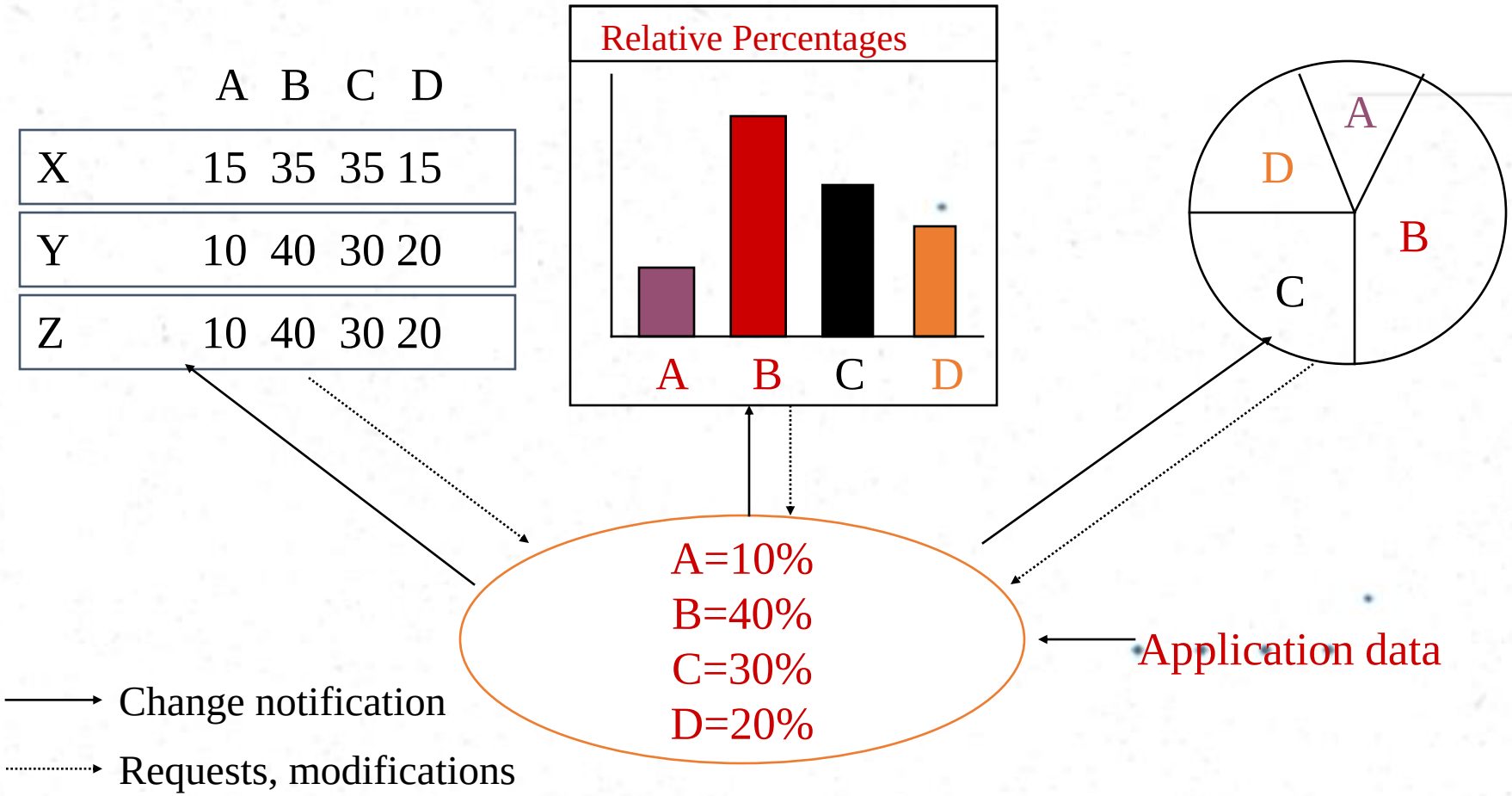
Observer

- Problem :We have an object that changes its state quite often. We want to provide multiple views of the current states

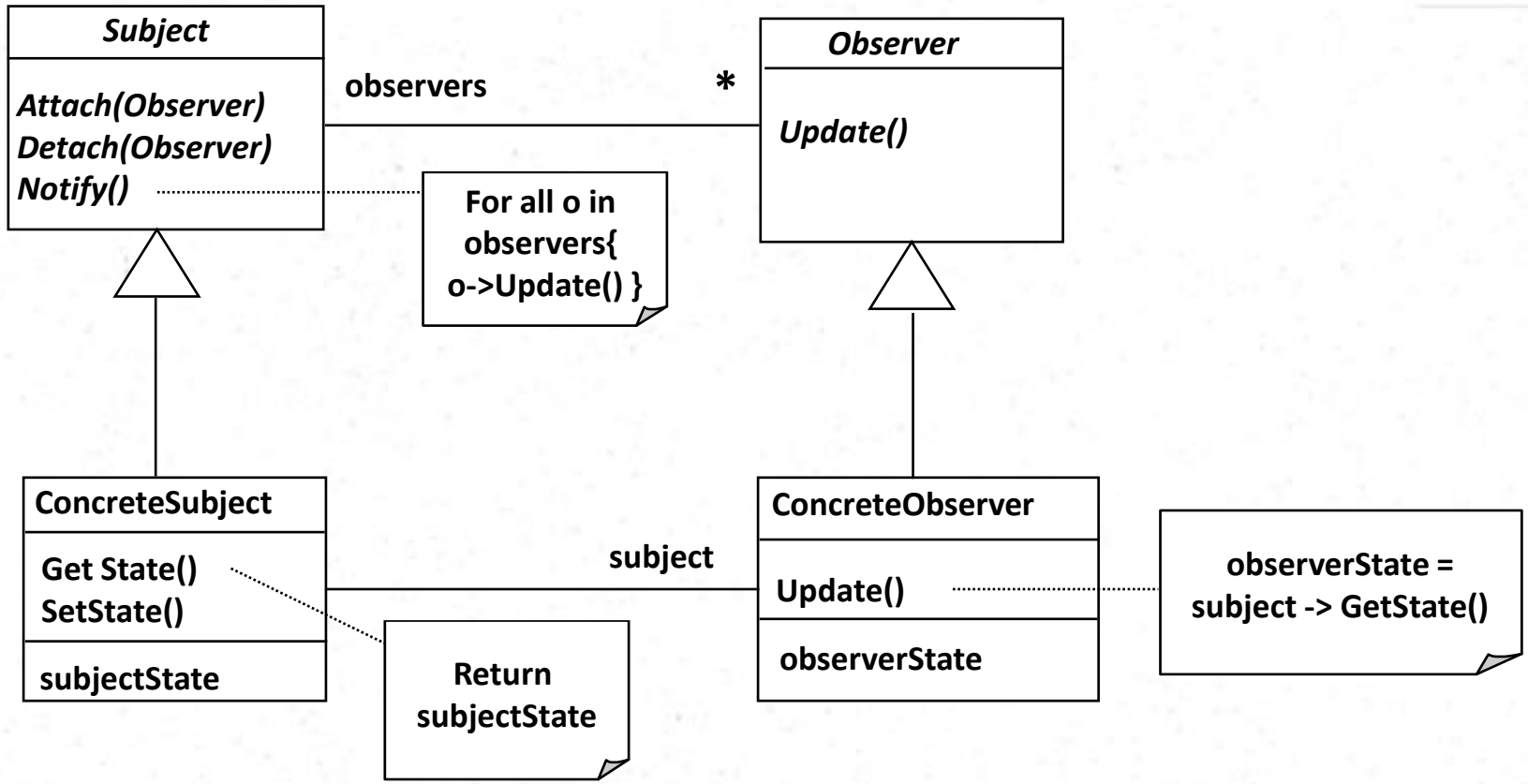
E.g :Histogram view, pie chart view , Bar graph …

- Solution :Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes.
- The system design should be highly extensible
- It should be possible to add new views without having to recompile the observed object or the existing views.

Observer Example



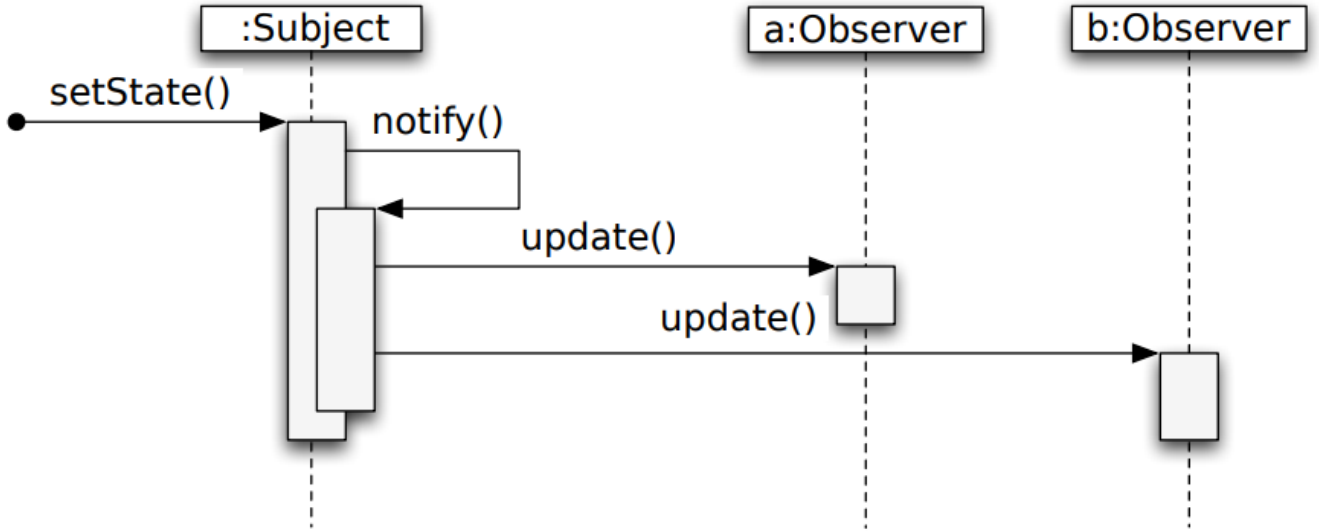
The Structure of Observer

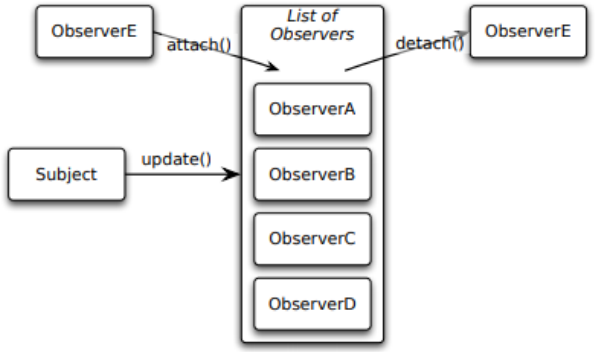


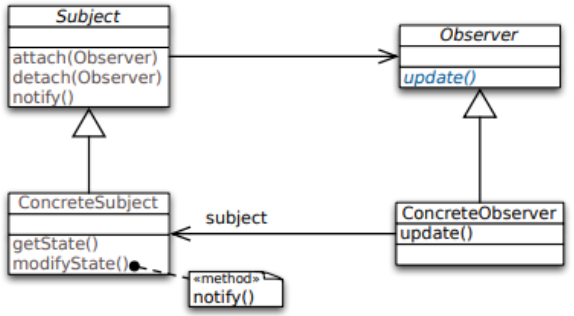
Observer : Participants

- 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


:Subject
a:Observer
b:Observer
setState()
notify()
update()
update()


ObserverE
attach()
List of Observers
detach()
ObserverE
ObserverA
Subject
update()
ObserverB
ObserverC
ObserverD


Subject
attach(Observer)
detach(Observer)
notify()
Observer
update()
ConcreteSubject
getState()
modifyState()
subject
ConcreteObserver
update()
«method»
notify()

Observer Pattern Usage

- **When to use this pattern?**

You should consider using this pattern in your application when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.

- **Real Life Uses:**
  - It is heavily used in GUI toolkits and event listener. In java the button(subject) and onClickListener(observer) are modelled with observer pattern.
  - Social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
  - All users of an app on play store gets notified if there is an update.

Activity- 1

- RDA of Sri Lanka is going to develop a system to model the cities of the country. Consider the partial requirement of the system given below and answer the questions.
- The system intends to find distance between cities by maintaining the set of cities as graphical structure. The application needs to maintain multiple views of the distances between cities via a map. One view provides a table of distances between cities as texts, another view maintains the same information as a 2D graph and another view displays this information as a 3D graph. Assume that the City class provides operations for obtaining the distance between any two cities. When new roads are built, the distances between cities can be changed accordingly.

a) What is the most appropriate design pattern to implement the above requirement? Justify your answer.

b) Draw the class diagram for your selected design pattern in part a).

Observer Pattern Pros & Cons

Pros:

- Supports the principle to strive for loosley coupled designs between objects that interact.
- Allows you to send data to many other objects in a very efficient mannor.
- No modification is need to be done to the subject to add new observers.
- You can add and remove observers at anytime.

Cons

- If not used carefully the observer pattern can add unecessary complexity.
- The order of Observer notifications is undependable.
- Observable protects crucial methods which means you can't even create an instance of the Observable class and compose it with your own objects, you have to subclass.

Structural Patterns

Decorator Pattern

Decorator Definition

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

As a Modern Girl Role

As a warrior character

As a Traditional Character

Decorator : Intent and Motivation

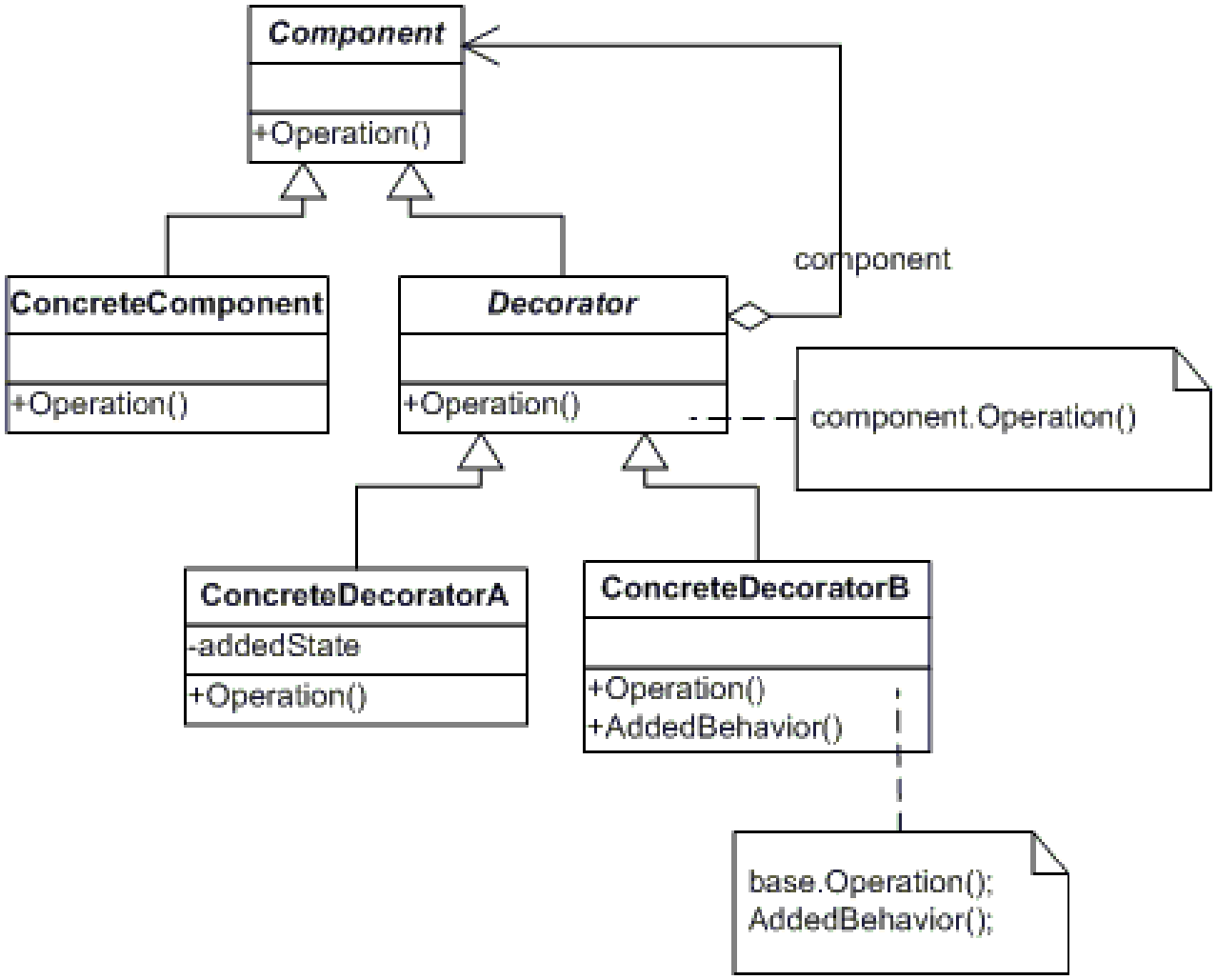
Intent

- 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 : Structure



Decorator

- Component: A component can be an object which is to be decorated as well as an object which decorates. The component to be decorated is wrapped by a component which decorates. Component can be interface or an abstract class
- Concrete Component: An implementation of Component for which we are going to add new responsibilities. This is the component to be decorated.
- Decorator: A Decorator IS-A Component as well as a Decorator HAS-A Component. i.e. Decorator implements a Component as well as it has instance variable of Component. This is the class which serves as abstract class for all concrete decorators.
- Concrete Decorator: This can extend behavior or state of a Component. In above example, ConcreteDecoratorA adds newMethod() and ConcreteDecoratorB adds newMember.

Decorator : Applicability

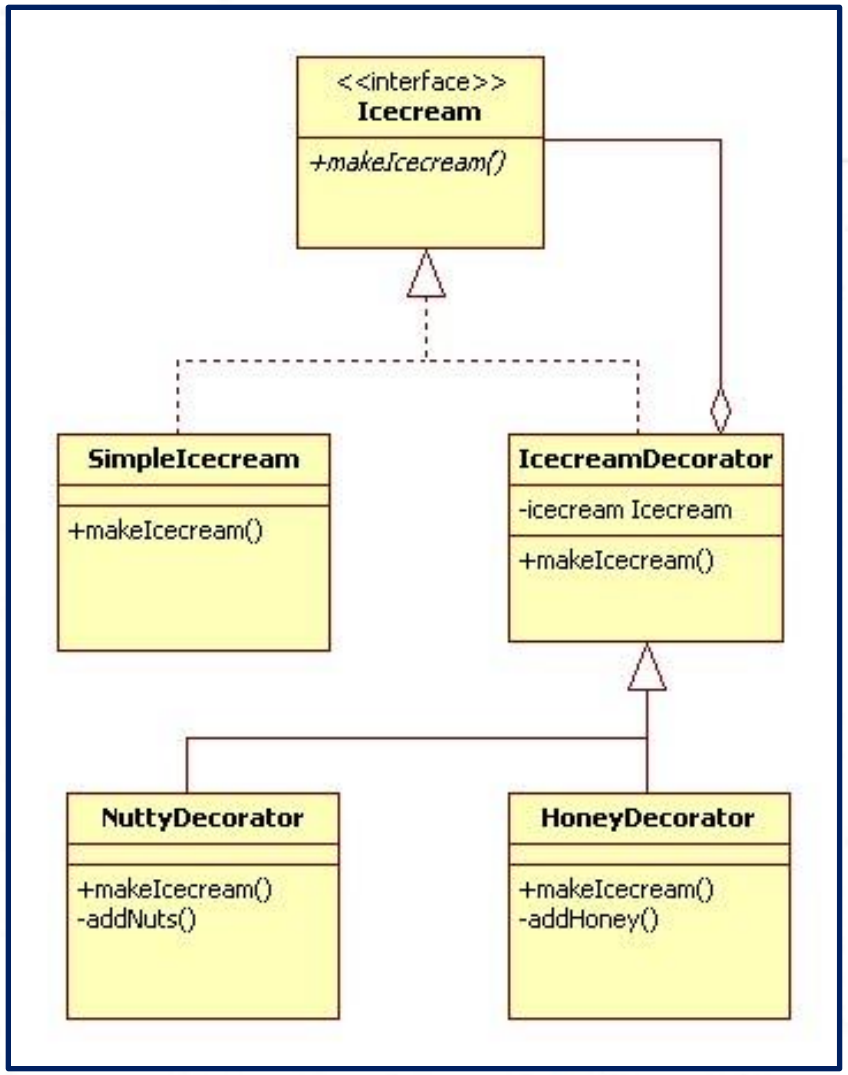
Use Decorator;

- 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 sub classing

Decorator pattern steps

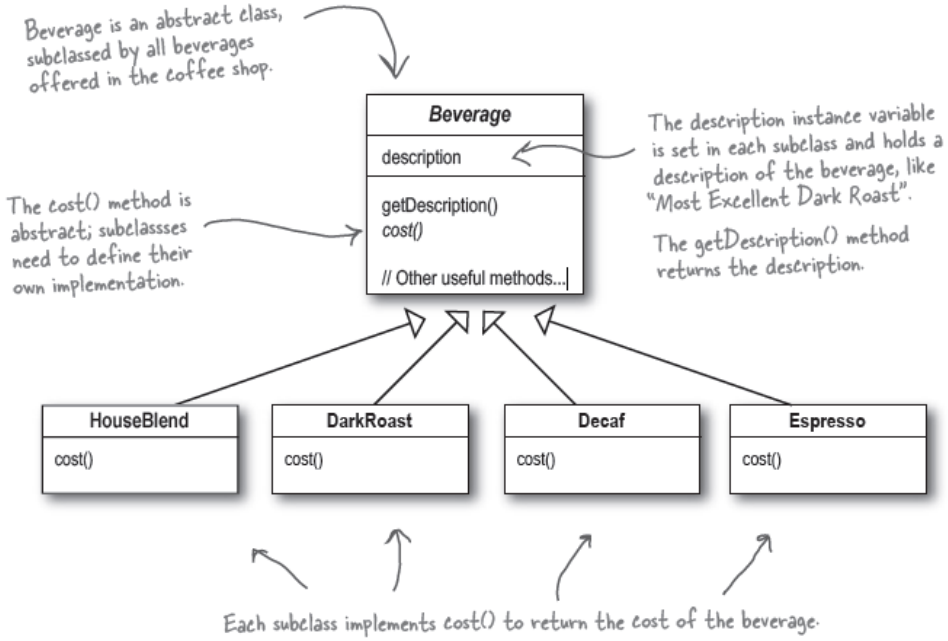
1. Create an interface for the class to be decorated – Icecream
2. Implement that interface with basic functionalities – SimpleIceCream
3. Create an abstract class that contains an attribute type of the interface. - IcecreamDecorator
4. The concrete decorator class will add its own methods. – NuttyDecorator,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



Activity- 2

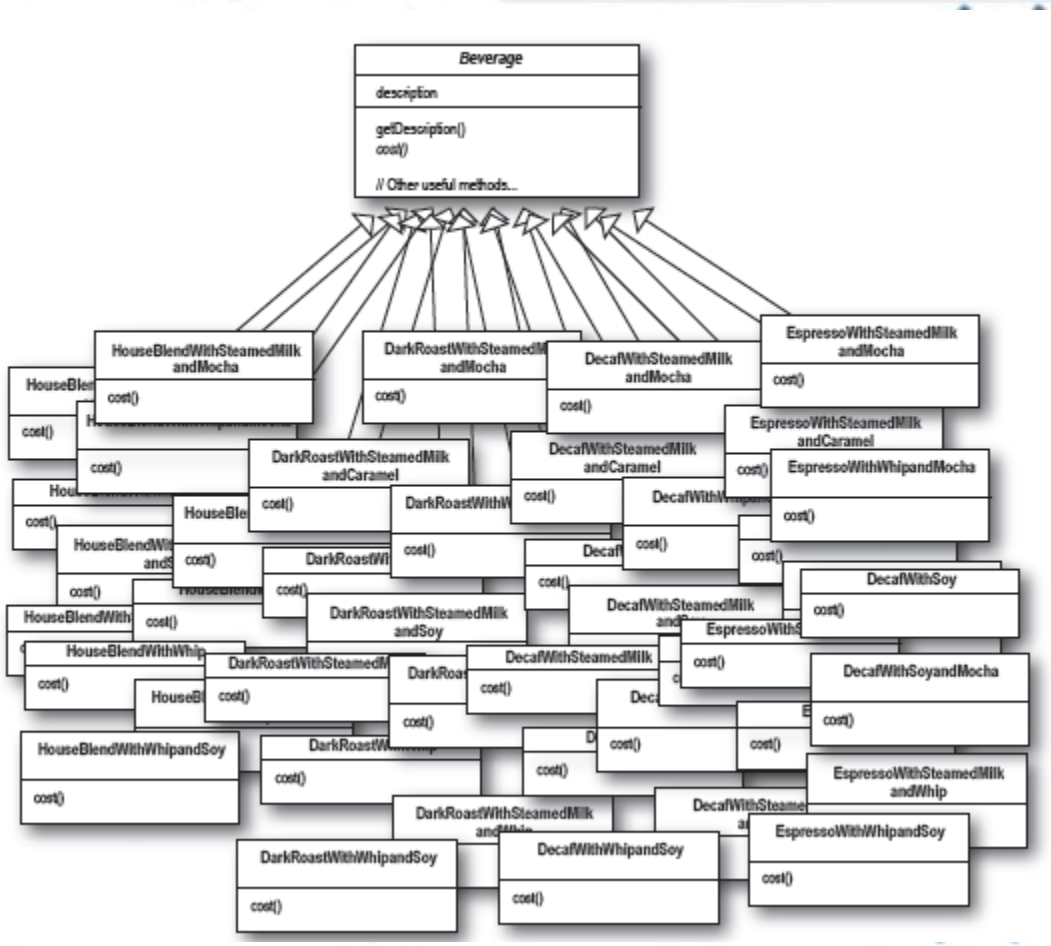
Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

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

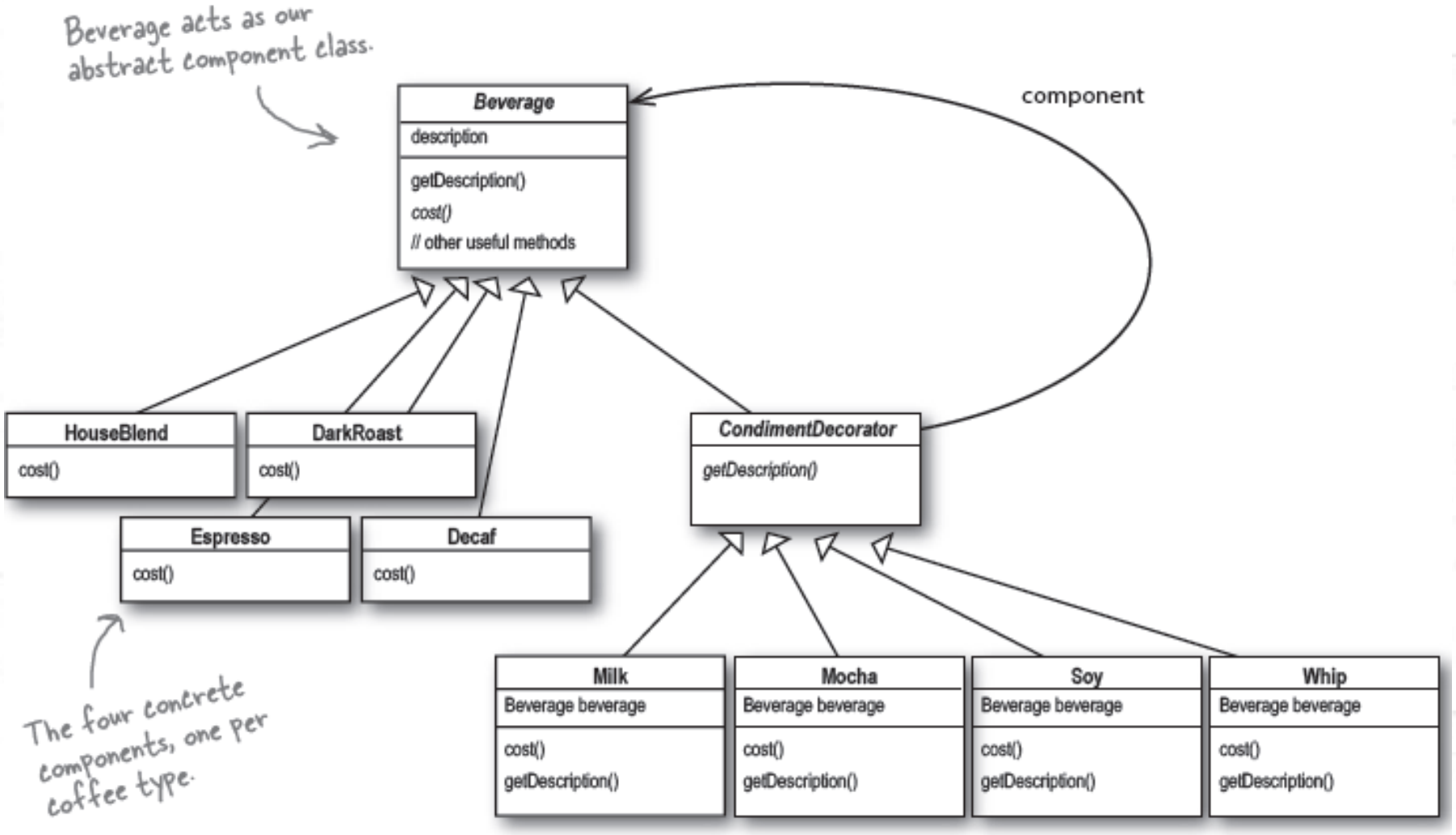


Activity -2

- 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.



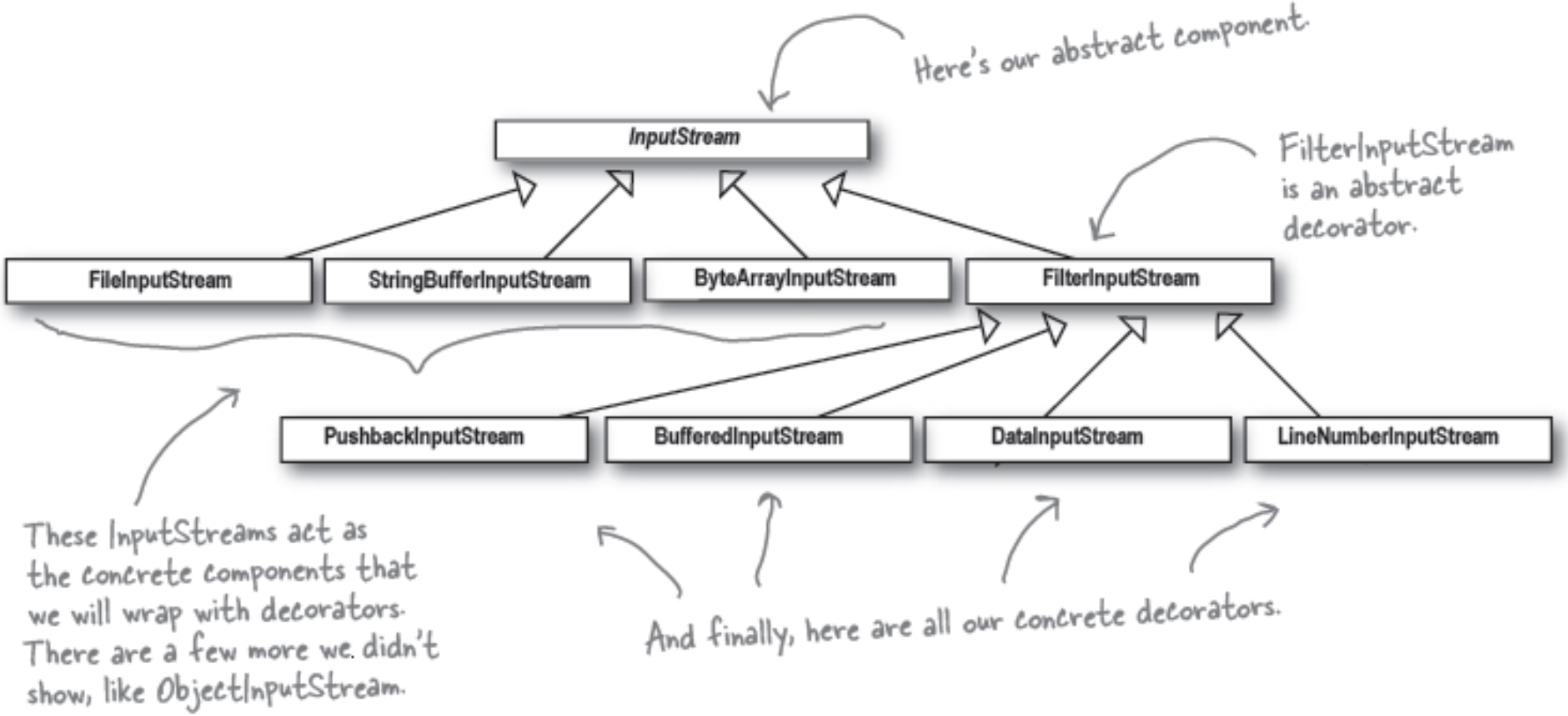
Activity -2 Answer



Activity - Decorator : Participants

- Component (Beverage)- Defines the interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent (HouseBlend, darkRost..)- Defines and 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 in Java I/O classes



Decorator Pros and Cons

Pros

- 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.

Cons

- Decorators can result in many small objects in our design, and overuse can be complex.
- Decorators can complicate the process of instantiating the component because you not only have to instantiate the component but wrap it in a number of decorators

Fcade Design Pattern

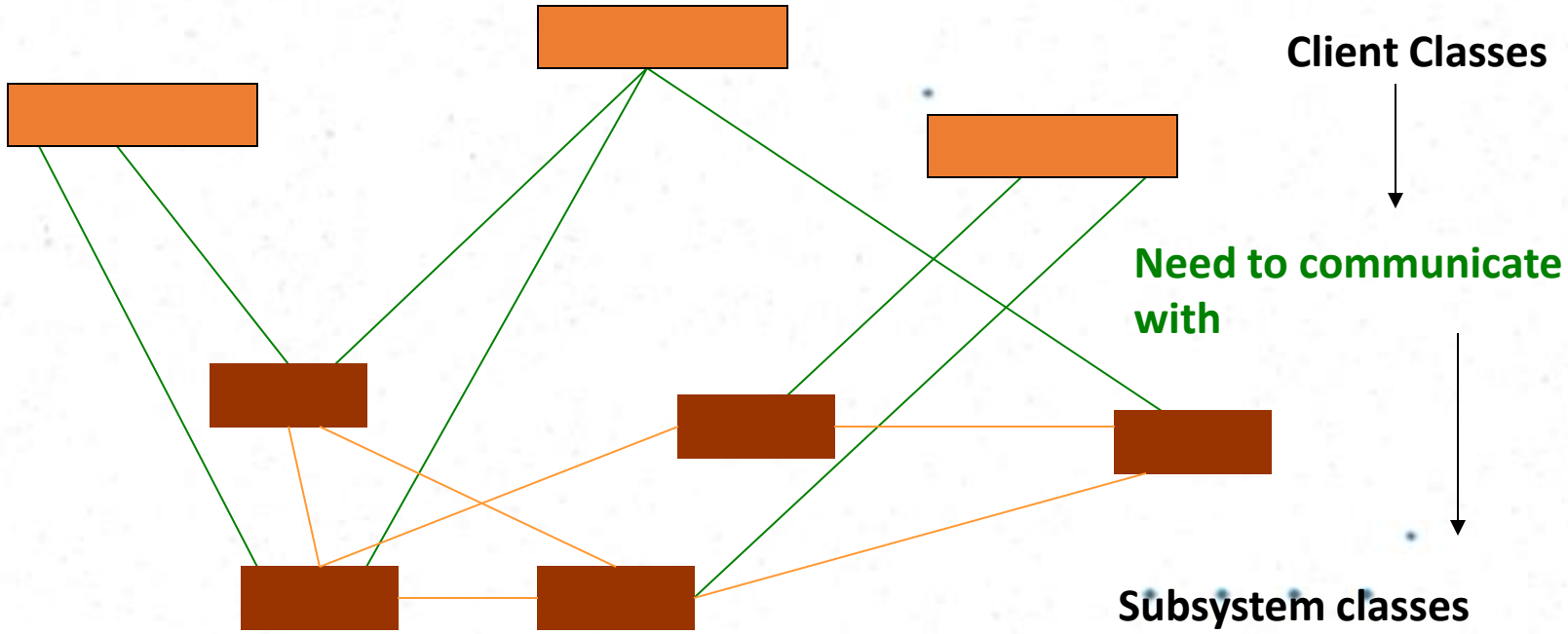
Facade Definition

Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.

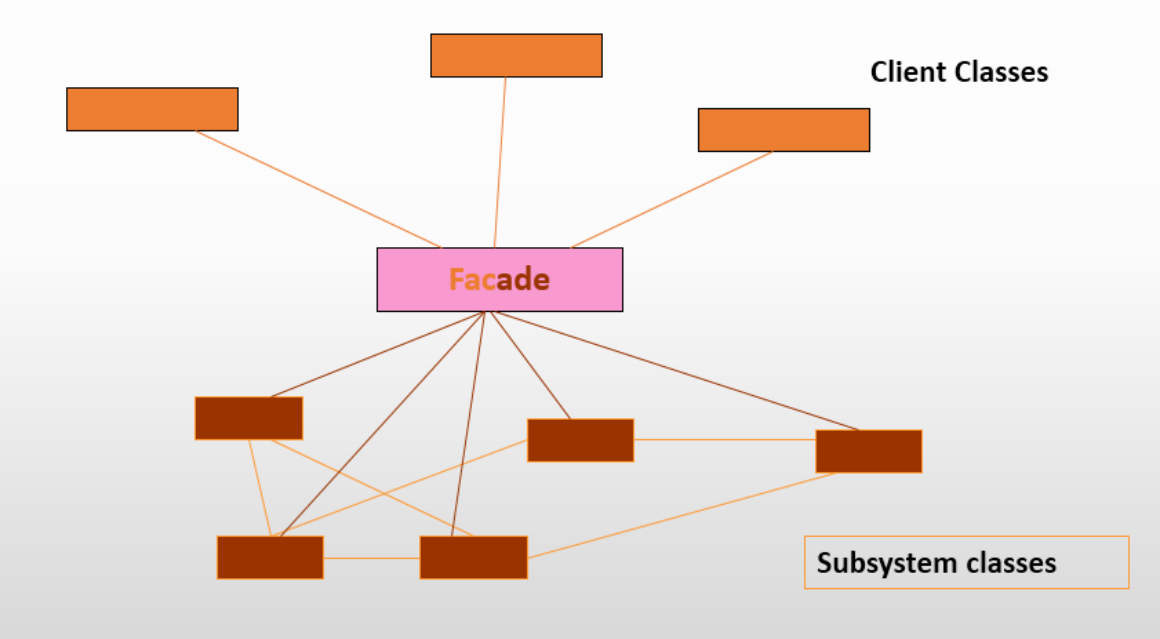
The Facade Design Pattern

- **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 but different interfaces via a Façade object that shield this complexity from clients.
- **Clients**
  - Sending requests to the Façade, which forwards them appropriately to the subsystem components.
  - Façade may do some adaptation code.
  - Clients do not need to know, or ever use the subsystem components directly.

Facade Pattern: Problem



Facade Pattern: Solution



- Create a class that accepts many different kinds of requests, and “forwards” them to the appropriate internal class (Subsystems classes).
- You Might need to write a separate Interface class for each (all implementing the same interface), but the users (Client classes) of your Facade class don’t need to change

Facade

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.

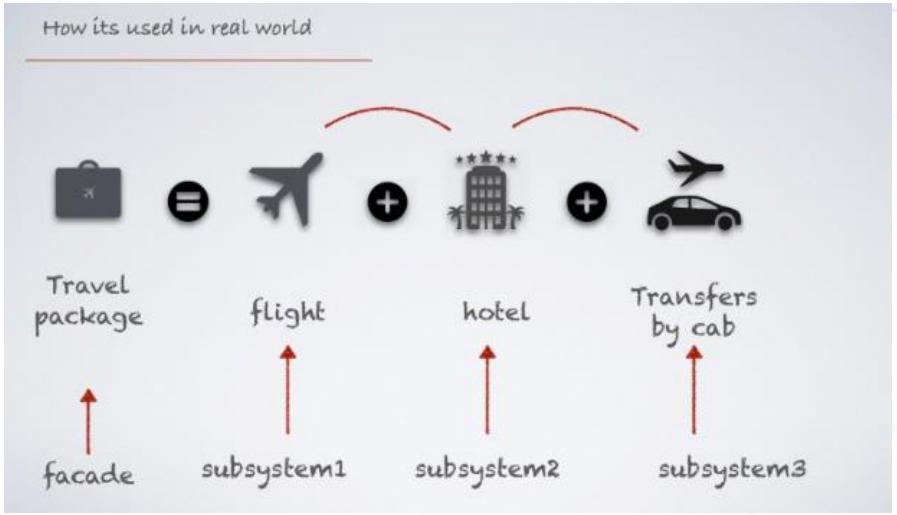
Facade Applicability

- 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.

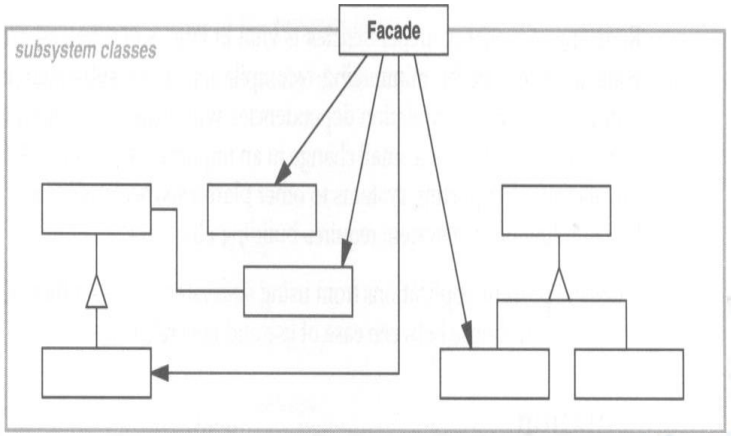
Activity -3

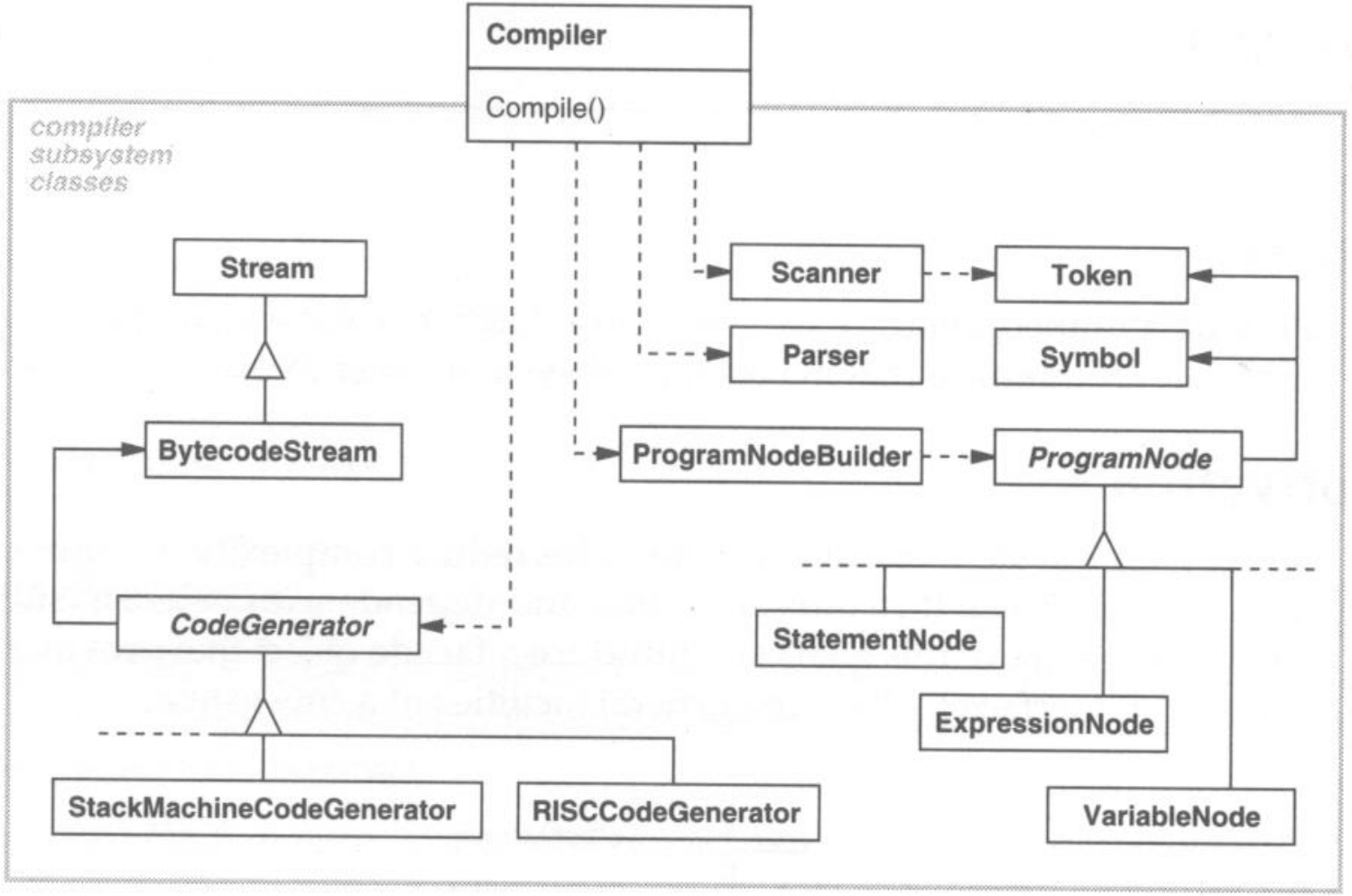
Let us consider an example of booking a package.

- Usually when you try to book a package , the ticket booking system interacts with many of subsystems.
- The various sub-systems may be flight , hotel and cab booking .
- In addition this may also interact with many other sub systems.
- In this case instead of client having the overhead of interacting with various other subsystems , we can introduce a facade layer which interacts will all these subsystems.
- Finally once it get the response from all the subsystems, it aggregates all these response and send the response back to the client.



Facade - Example





Facade 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.

Facade Summary

- Ideally our **facade provides a unified interface for interacting with all other subsystems** .
- And also the **facade layer which we have introduced acts a higher level interface** which interacts with other subsystems.

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!

Definitions

- Flyweight
  - A shared object that can be used in multiple contexts simultaneously.
- Intrinsic
  - State information that is independent of the flyweights context. Shareable Information.
- Extrinsic
  - State information that depends on the context of the flyweight and cannot be shared.

Flyweight : Example

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

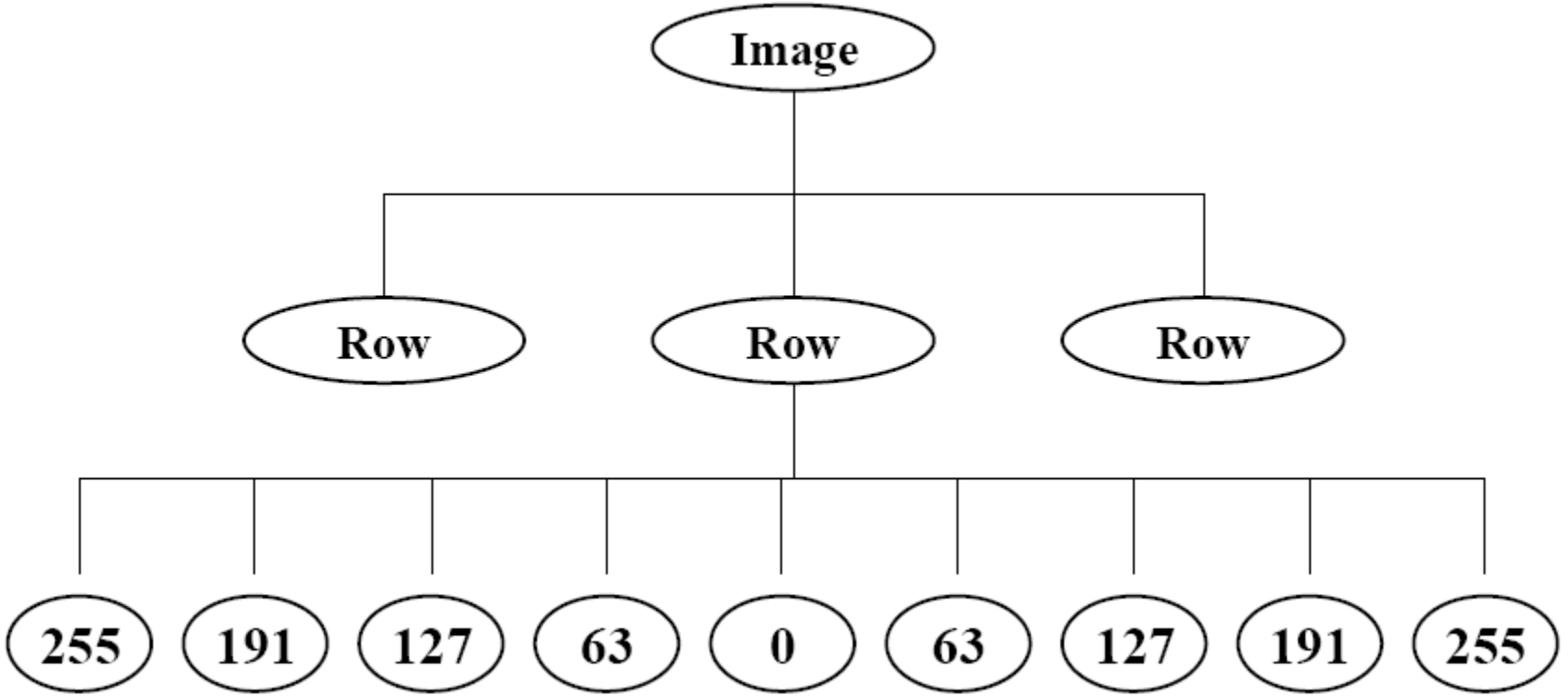
- An image is a 2-D array of pixels, where each pixel specifies an intensity value that typically ranges from 0 (dark) to 255 (light).
  - A monochrome TV image with dimensions 768 by 576 contains over 1 million pixels.
  - A colour TV image of this size contains over 3 million pixels.
- Image processing applications typically use objects to represent images, but using an object for each pixel in an image is not the best approach. Why ??

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

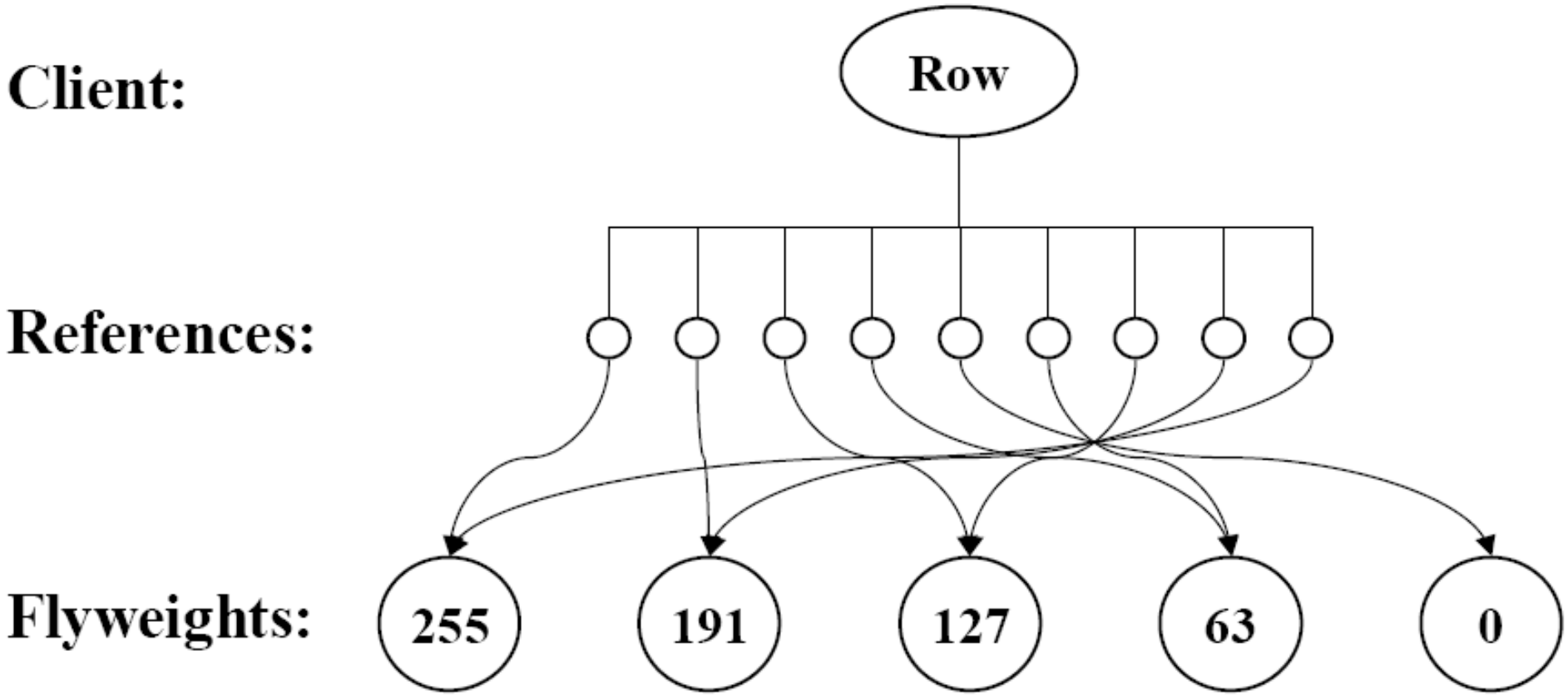
- Logically, every pixel in an image is an object:



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

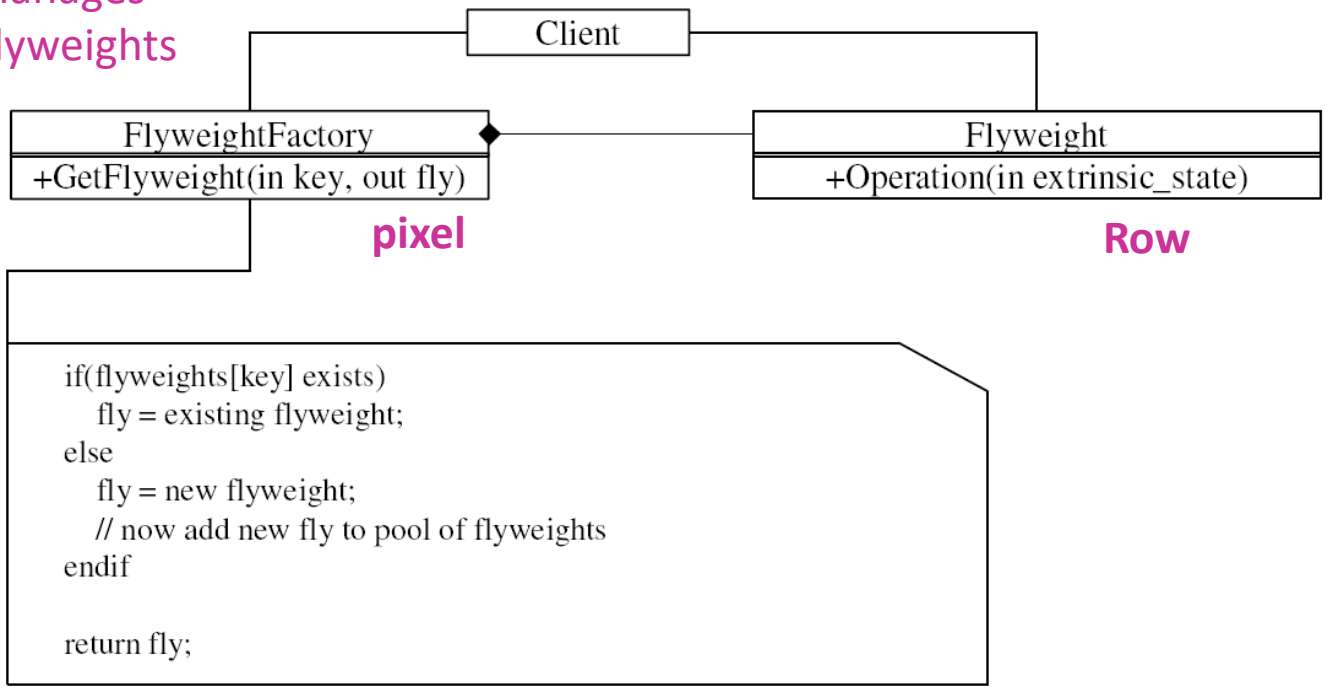
- The Flyweight pattern relies upon a distinction between intrinsic and extrinsic state.

- **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: (Simplified) Structure

Creates & manages Flyweights



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.

Activity

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.

Applicability

- Use the Flyweight when *all* of the following are true:
  - Application has a large number of objects.
  - Storage costs are high because of the large quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few once you remove their extrinsic state.
  - The application doesn't depend on object identity.

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 state is separated into intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; extrinsic state is stored or computed in the client objects.
- Clients pass the extrinsic state to the flyweight when they invoke its methods.

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.

Singleton (Covered in OOP)

Singleton Definition – Covered in OOP

Ensure a class only has one instance and provide a global point of access to it.

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.
- That instance should not be resource intensive and it should be able to create only when it is needed.

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 should only ever have one instance because there should only be one administrator of the system.

  - Otherwise, by instantiating more than one would run into problems such as incorrect program behavior, overuse of resources or inconsistent results. To avoid them, we use Singleton.

Singleton - Implementation

- To create an object from any class, we need to call the constructor of that class.

  E.g.: new MyClass();

- Can instantiate this class more than once.

```
public MyClass {

    public MyClass() {}

}
```

Singleton - Implementation

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

Step 1 : Declare a default private constructor.

```
public MyClass {

    private MyClass() {}

}
```

Can we create at least one object from this class?

Singleton - Implementation

- Since the constructor is private, instantiating an object from MyClass is impossible.

- It's a chicken and egg problem.

How to solve it?

Singleton - Implementation

- Create a 'public static' method to return an object of type MyClass.

```
public MyClass {

    private MyClass() {}

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

- Since it is a public method any class can access it. Since it is a static method (Class method) , can access it through class name without instantiating an object.

  e.g. : MyClass.getInstance

Singleton- Implementation

- Still the above code does not satisfy the Singleton problem since it can initialize more than one object.

- To complete the code, follow the steps below:

  Step 2 : Declare a private static variable to hold single instance of class.

  Step 3 : Declare a public static function that returns the single instance of class.

  Step 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!

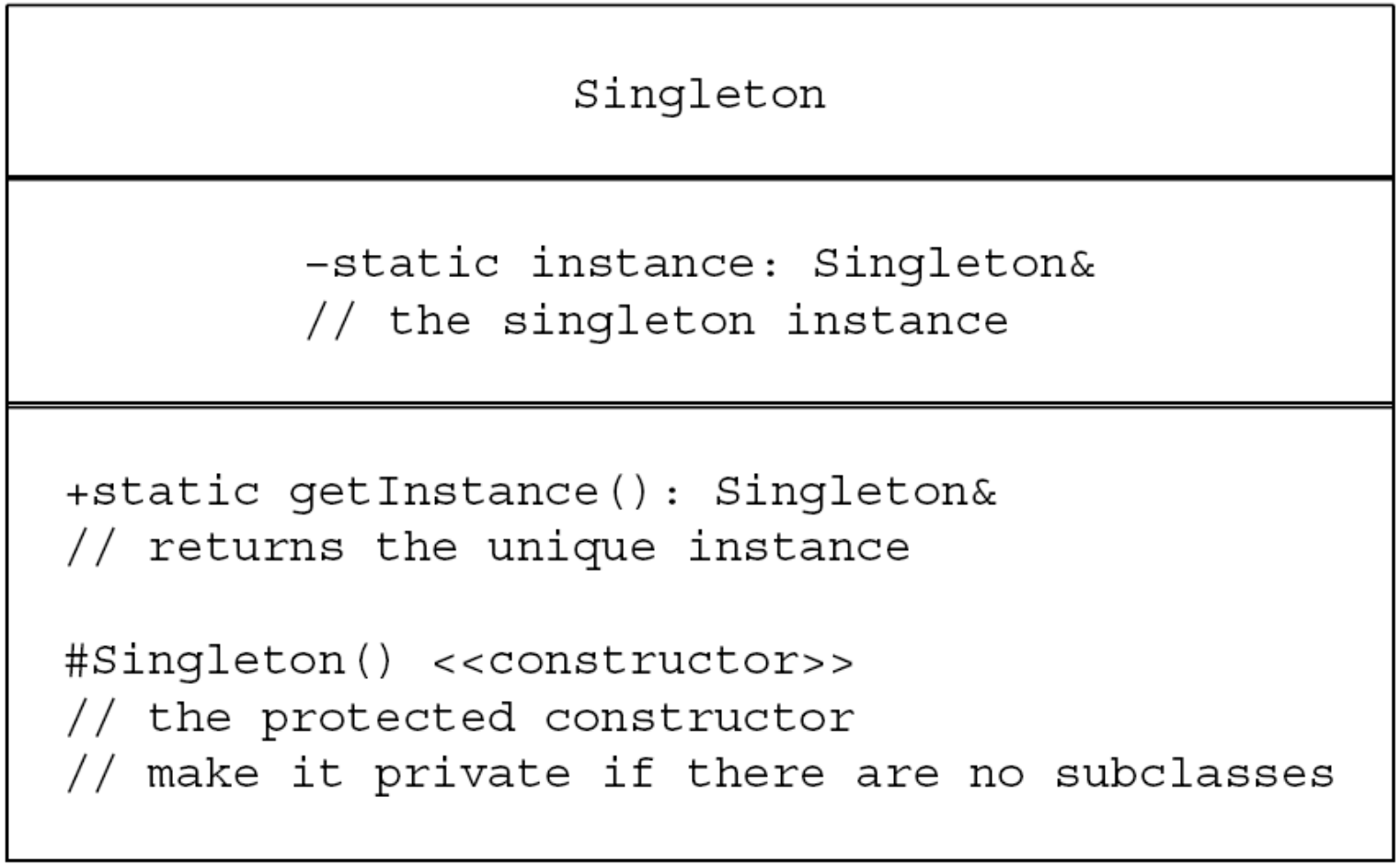
The getInstance() method gives us a way to instantiate the class and also to return an instance of it.

Of course, Singleton is a normal class; it has other useful instance variables and methods.

What is meant by Lazy Initialization?

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

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.

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 when trying 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 class Singleton {

    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance()
    {
        if (uniqueInstance == null)  //1
        {
            uniqueInstance = new Singleton(); //2
        }

        return uniqueInstance; //3
    }
}
```

Singleton: Implementation –Multithreading

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

    return uniqueInstance; //3
}
}
```

- If one thread enters 'getInstance()' method and finds the 'uniqueInstance' to be null at step 1.
- Then it enters the IF block.
- Before it can execute step 2 another thread enters the method and executes step 1 which will be true as the uniqueInstance 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 – Multithreading

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

1. By making getInstance() a synchronized method.

```
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.

Singleton: Implementation – Multithreading

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.
- Once we have set the uniqueInstance variable to an instance of Singleton, we have no further need to synchronize this method.
- Therefore, after the first time through, synchronization is totally unneeded overhead.

Singleton: Implementation – Multithreading

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 uniqueInstance variable when the class is loaded and before any thread accesses the static instance variable.

What is meant by Eager Initialization?

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

Singleton: Implementation – Multithreading

3. Use 'double – checked locking'

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

    private Singleton() {}

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

Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

* The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

Singleton: Known Uses & Related Patterns

- **Known Uses:**
  - Singleton is useful whenever having more than one instance of a class is undesirable.
  - For example:
    - Classes implementing Thread Pools
    - Database connection pools,
    - Caches
    - Objects used for logging
    - Objects that act as device drivers to devices such as printers and graphic cards etc.
- Generally use 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 can be generalized:
    - 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.

Thank you