CS 213 SOFTWARE METHODOLOGY

Lily Chang

CS Department @ Rutgers New Brunswick

FALL 2023

Design Patterns

- Lecture Note #21

What is design pattern?

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

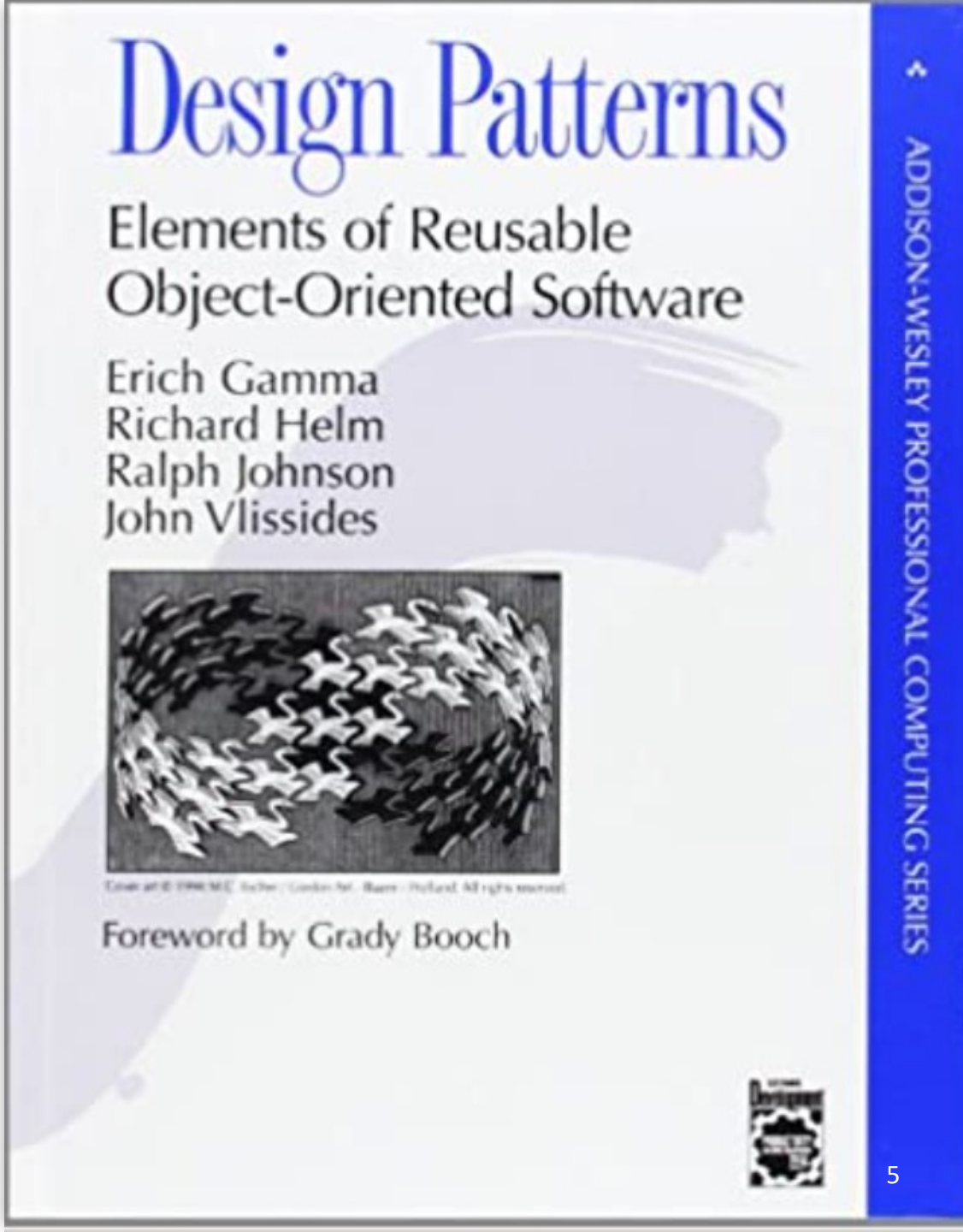
- by Christopher Alexander

What is design pattern?

- Designing object-oriented software is hard, and designing reusable object-oriented software is even harder
  - You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them
  - Your design should be specific to the problem at hand but also general enough to address future problems and requirements
  - You'll find recurring patterns of classes and communicating objects in many object-oriented systems, and these patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable

Gang of Four (GoF)

- The book (1994) has been influential to the field of software engineering
- The authors are often referred to as the Gang of Four (GoF)
- It is considered as an important source for object-oriented design theory and practice (with C++ examples)



Essential components of a design pattern

The pattern name

Use to describe a design problem, its solutions, and consequences in a word or two

A vocabulary for communication

The problem

Describes when to apply the pattern

Explains the problem and its context; such as specific design problems, how to represent algorithms as objects, or symptomatic of an inflexible design

The solution

Describes the elements that make up the design, their relationships, responsibilities, and collaborations.

Doesn't describe a particular concrete design or implementation,

A pattern is like a template that can be applied in many different situations, thus, provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

The consequences

The results and trade-offs of applying the pattern.

Are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

Object Oriented Design Patterns

Reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability

The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design

A design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

Classification of Design Patterns

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them.

The classification helps you learn the patterns faster, and it can direct efforts to find new patterns as well

The design patterns is classified by two criteria

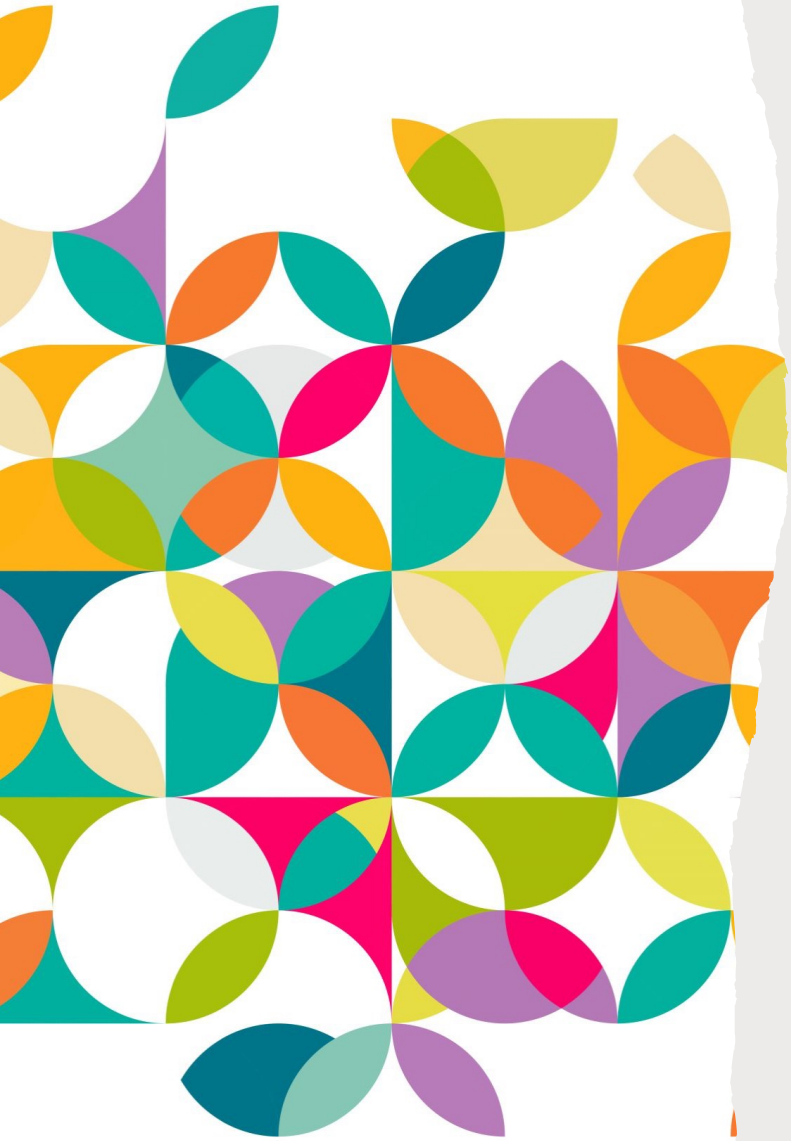
1. **purpose**, reflects what a pattern does.

2. **scope**, specifies whether the pattern applies primarily to classes or to objects.

Classification of design patterns

- Patterns can have either creational, structural, or behavioral purpose.
- Creational patterns concern the process of object creation.
- Structural patterns deal with the composition of classes or objects.
- Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)



Classification of Design Patterns

- Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time.
- Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.
- Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object.
- The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects.
- The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

Design for Change

The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.

To design the system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class redefinition and reimplementation, client modification, and retesting.

Redesign affects many parts of the software system, and unanticipated changes are invariably expensive.

Design patterns help you avoid this by ensuring that a system can change in specific ways. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

Common Causes of Redesign

Creating an object by specifying a class explicitly

- Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes.
- To avoid it, create objects indirectly.
  - The Design patterns Abstract Factory, Factory Method, Prototype can be used to address the problems

Dependence on specific operations

- When you specify a particular operation, you commit to one way of satisfying a request.
- By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.
- Design patterns Chain of Responsibility, Command can be used to address the problems

Common Causes of Redesign

- Dependence on hardware and software platform
  - External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms.
  - Software that depends on a particular platform will be harder to port to other platforms.
  - It may even be difficult to keep it up to date on its native platform. It's important therefore to design your system to limit its platform dependencies.
  - Design patterns Abstract Factory, Bridge can be used to address the problems

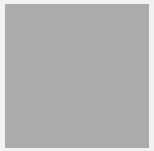
Common Causes of Redesign

Dependence on object representations or implementations

Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes

Hiding this information from clients keeps changes from cascading

Design patterns Abstract Factory, Bridge, Memento, Proxy can be used to address the problems



Algorithmic dependencies

Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes

Algorithms that are likely to change should be isolated.

Design patterns Builder, Iterator, Strategy, Template Method, Visitor can be used to address the problems

Common Causes of Redesign

- Tight coupling
  - Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other.
  - Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain.
  - Loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily.
  - Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems; such as Abstract Factory, Bridge, Chain of Responsibility

Common Causes of Redesign

- Extending functionality by subclassing
  - Customizing an object by subclassing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.).
  - **Defining a subclass also requires an in-depth understanding of the parent class**. For example, overriding one operation might require overriding another. An overridden operation might be required to call an inherited operation. And subclassing can lead to an explosion of classes, because you might have to introduce many new subclasses for even a simple extension.
  - Object composition in general and delegation in particular provide flexible alternatives to inheritance for combining behavior. New functionality can be added to an application by composing existing objects in new ways rather than by defining new subclasses of existing classes. On the other hand, heavy use of object composition can make designs harder to understand.
  - Many design patterns produce designs in which you can introduce customized functionality just by defining one subclass and composing its instances with existing ones.
    - Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

Common Causes of Redesign

- Inability to alter classes conveniently
- Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it (as may be the case with a commercial class library). Or maybe any change would require modifying lots of existing subclasses.
  - Design patterns offer ways to modify classes in such circumstances.
- Adapter, Decorator, Visitor.

How To Select a Design Pattern

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Design Patterns to be discussed

Object Creational Patterns

- Abstract Factory
- Singleton

Behavioral Patterns

- State (Object Behavioral Pattern)
- Template Method (Class Behavioral Pattern)

Model-View-Controller (MVC)

- Observer (Object Behavioral Pattern)
- Composite (Object Structural Pattern)
- Strategy (Object Behavioral Pattern)

Creational Patterns

Creational design patterns abstract the instantiation process.

They help make a system independent of how its objects are created, composed, and represented.

A class creational pattern uses inheritance to vary the class that's instantiated

An object creational pattern will delegate instantiation to another object.

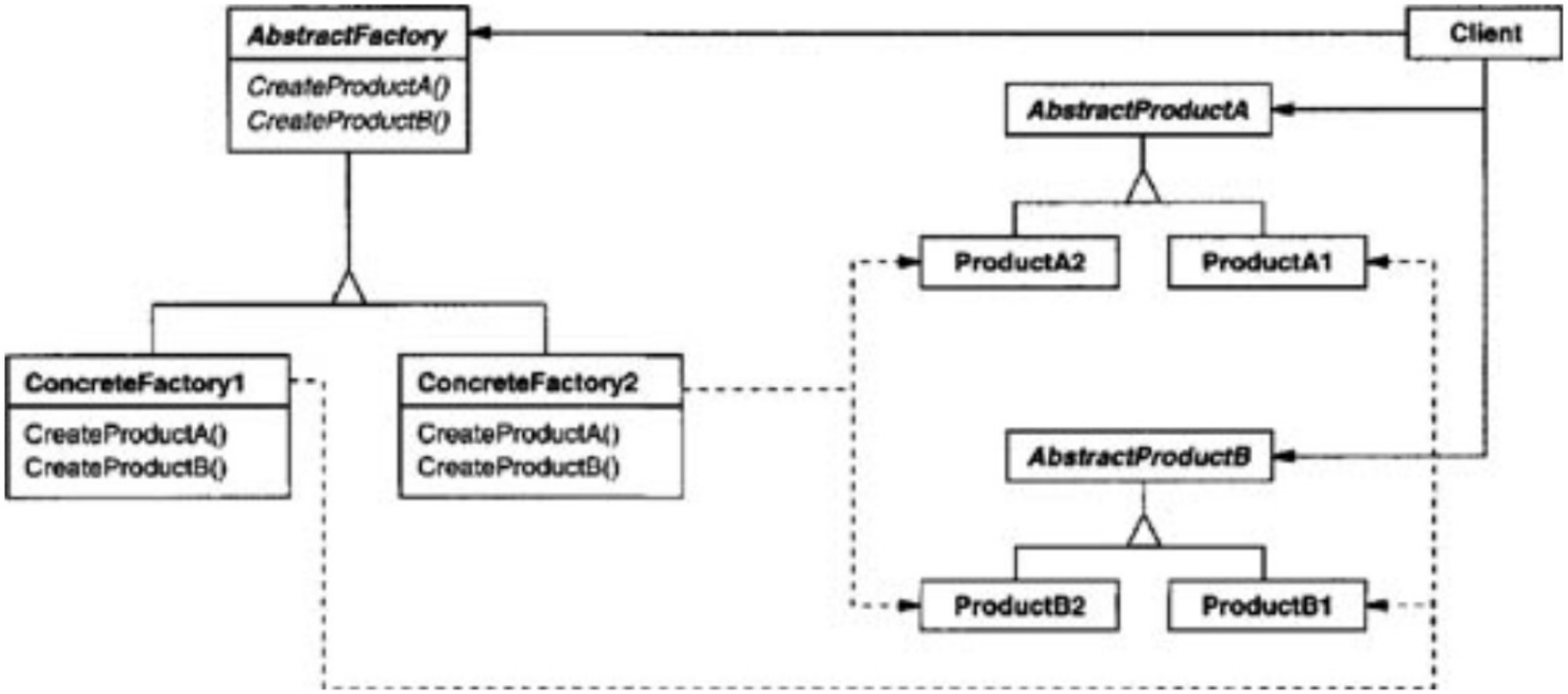
Abstract Factory (object creational)

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes
- This pattern helps us to interchange specific implementations without changing the code that uses them, even at runtime
- This pattern provides a way to encapsulate a group of individual factories that have a common theme. Here a class does not create the objects directly; instead, it delegates the task to a factory object.

Abstract Factory – applicability

- When a system should be independent of how its products are created, composed, and represented; for example, the pizza store.
- When a system should be configured with one of multiple families of products; for example, different types of pizzas with different toppings
- When a family of related product objects is designed to be used together, and you need to enforce this constraint; for example, a car factory

Abstract Factory – Structure



Real-World Example

- Suppose that we are decorating our room with two different tables: one made of wood and one made of steel. For the wooden table, we need to visit to a carpenter, and for the other table, we need to go to a metal shop. Both are table factories, so based on our demand, we decide what kind of factory we need.
- In this context, you may consider two different car manufacturing companies: Honda and Ford. Honda makes models, such as CR-V, Jazz, Brio, and so forth. Ford makes different models, such as Mustang, Figo, Aspire, and so forth. So, if you want to purchase a Jazz, you must visit a Honda showroom, but if you prefer a Figo, you go to a Ford showroom.

- Let’s assume that we have two factories: one creates dogs and the other creates tigers; and now, you want to categorize dogs and tigers further
- You may choose a domestic animal (dog or tiger) or a wild animal (dog or tiger) through these factories; so let’s introduce two concrete factories: WildAnimalFactory and PetAnimalFactory, where the WildAnimalFactory is responsible for creating wild animals and the PetAnimalFactory is responsible for creating domestic animals, or pets

Computer-World Example

Illustration

- AnimalFactory: An interface that is treated as the abstract factory in the following example
- WildAnimalFactory: A concrete factory that implements AnimalFactory interface. It creates wild dogs and wild tigers.
- PetAnimalFactory: Another concrete factory that implements the AnimalFactory interface. It creates pet dogs and pet tigers.
- Tiger and Dog: Two interfaces that are treated as abstract products in this example.
- PetTiger, PetDog, WildTiger, and WildDog: The concrete products in the following example

```
interface Dog {
    void speak();
    void preferredAction();
}

interface Tiger {
    void speak();
    void preferredAction();
}

class WildDog implements Dog {
    @Override
    public void speak() {.
        System.out.println("Wild Dog says loudly: Bow-Wow.");
    }
    @Override
    public void preferredAction() {
        System.out.println("Wild Dogs prefer to roam freely
          in jungles.\n");
   }
}

class PetDog implements Dog {
    @Override
    public void speak() {
        System.out.println("Pet Dog says softly: Bow-Wow.");
    }
    @Override
    public void preferredAction() {
        System.out.println(
                    "Pet Dogs prefer to stay at home.\n");
   }
}
```

```
class WildTiger implements Tiger {
    @Override
    public void speak() {
        System.out.println(
                    "Wild Tiger says loudly: Halum.");
    }
    @Override
    public void preferredAction() {
        System.out.println("Wild Tigers prefer hunting
          in jungles.\n");
   }
}

class PetTiger implements Tiger {
    @Override
    public void speak() {
        System.out.println(
                    "Pet Tiger says softly: Halum.");
    }
    @Override
    public void preferredAction() {
        System.out.println("Pet Tigers play
          in the animal circus.\n");
   }
}
```

```
//Abstract Factory
interface AnimalFactory {
    Dog createDog();
    Tiger createTiger();
}

//Concrete Factory-Wild Animal Factory
class WildAnimalFactory implements
AnimalFactory {
    @Override
    public Dog createDog() {
        return new WildDog();
    }
    @Override
    public Tiger createTiger() {
        return new WildTiger();
    }
}

//Concrete Factory-Pet Animal Factory
class PetAnimalFactory implements
AnimalFactory {
    @Override
    public Dog createDog() {
        return new PetDog();
    }
    @Override
    public Tiger createTiger() {
        return new PetTiger();
   }
}
```

```
//Client
class AbstractFactoryPatternExample {
    public static void main(String[] args) {
        AnimalFactory myAnimalFactory;
        Dog myDog;
        Tiger myTiger;
        System.out.println(
                  "***Abstract Factory Demo***\n");
        //Making a wild dog through WildAnimalFactory
        myAnimalFactory = new WildAnimalFactory();
        myDog = myAnimalFactory.createDog();
        myDog.speak();
        myDog.preferredAction();
        //Making a wild tiger through WildAnimalFactory
        myTiger = myAnimalFactory.createTiger();
        myTiger.speak();
        myTiger.preferredAction();
        System.out.println("******************");
        //Making a pet dog through PetAnimalFactory
        myAnimalFactory = new PetAnimalFactory();
        myDog = myAnimalFactory.createDog();
        myDog.speak();
        myDog.preferredAction();
        //Making a pet tiger through PetAnimalFactory
        myTiger = myAnimalFactory.createTiger();
        myTiger.speak();
        myTiger.preferredAction();
    }
}
```

Singleton Design Pattern

It's important for some classes to have exactly one instance; for example,

Although there can be many printers in a system, there should be only one printer spooler; there should be only one file system and one window manager; a digital filter will have one A/D converter; an accounting system will be dedicated to serving one company.

This pattern ensures a class only has one instance and provide a global point of access to it.

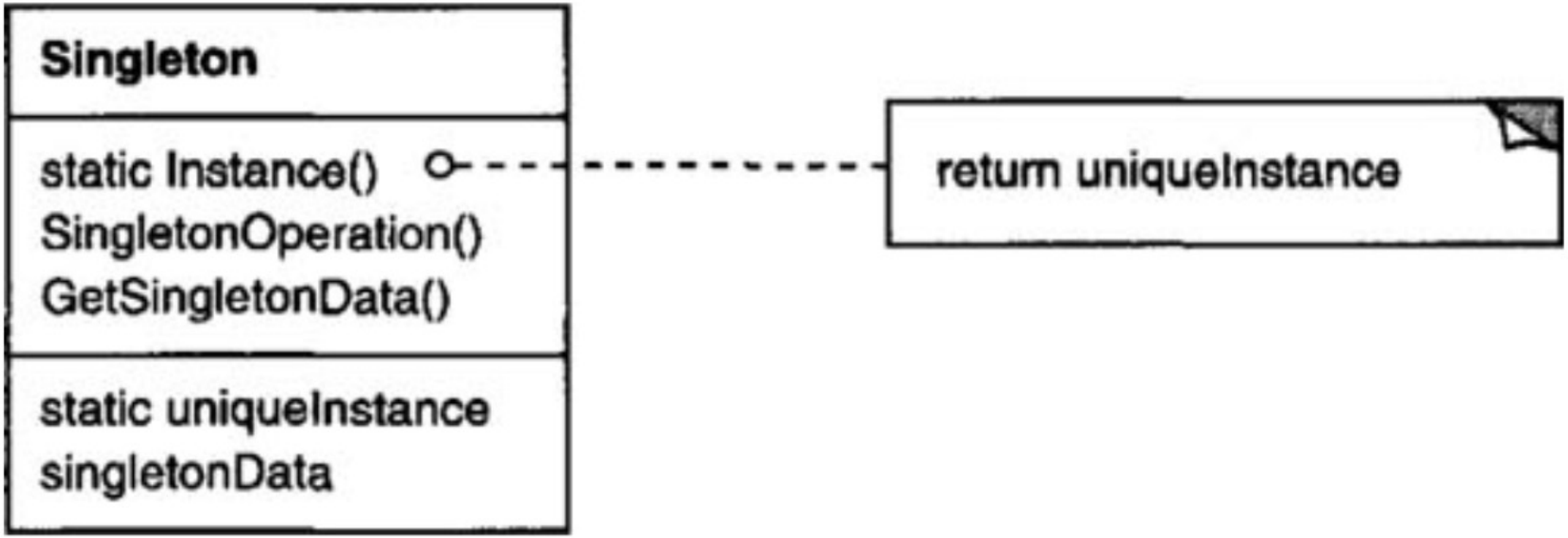
How do we ensure that a class has only one instance and that the instance is easily accessible?

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance.

Singleton Pattern – applicability

- Use the Singleton pattern when
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

Singleton Pattern – Structure



Real-World Example

- Let's assume that you are a member of a sports team, and your team is participating in a tournament; your team needs to play against multiple opponents throughout the tournament
- Before each of these matches, as per the rules of the game, the captains of the two sides must do a coin toss. If your team does not have a captain, you need to elect someone as a captain.
- Prior to each game and each coin toss, you may not repeat the process of electing a captain if you already nominated a person as a captain for the team.
- Basically, from every team member's perspective, there should be only one captain of the team.

Computer-World Example

- In some specific software systems, you may prefer to use only one file system for the centralized management of resources. Also, this pattern can implement a caching mechanism.
- You notice a similar pattern when you consider the getRuntime( ) method of the java.lang.Runtime class. It is implemented as a Singleton class.
- Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime() method.

Illustration

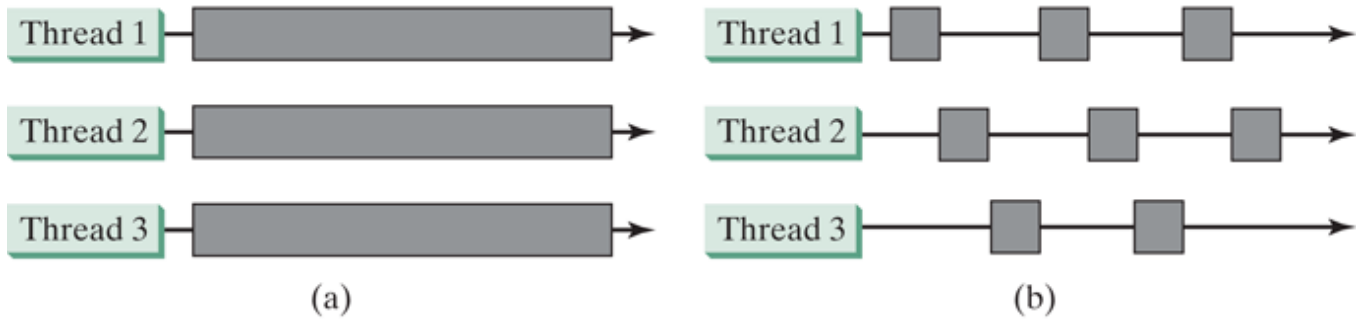
- The constructor is private, so you cannot instantiate the Singleton class(Captain) outside.
  - It helps us to refer the only instance that can exist in the system, and at the same time, you restrict the additional object creation of the Captain class.
  - The private constructor also ensures that the Captain class cannot be extended. So, subclasses cannot misuse the concept.
- The use of the "synchronized" keyword
  - So, multiple threads cannot involve in the instantiation process at the same time
  - It is forcing each thread to wait its turn to get the method, so thread-safety is ensured.
  - Note that, synchronization is a costly operation and once the instance is created, it is an additional overhead.

```
final class Captain {
    private static Captain captain;
    //We make the constructor private to prevent the use of "new" private
    private Captain() { }
     public static synchronized Captain getCaptain() {
        // Lazy initialization; create an object only when it is required.
        if (captain == null) {
             captain = new Captain();
             System.out.println("New captain is elected for your team.");
         }
         else {
             System.out.print("You already have a captain for your team.");
             System.out.println("Send him for the toss.");
         }
        return captain;
    }
}
```

```
// We cannot extend Captain class. The constructor is private in this case.
//For example, class B extends Captain {} will give you an error
public class SingletonPatternExample {
    public static void main(String[] args) {
        System.out.println("***Singleton Pattern Demo***\n");
        System.out.println("Trying to make a captain for your team:");
        //Constructor is private. We cannot use "new" here.
         //Captain c3 = new Captain(); will give you an error
         Captain captain1 = Captain.getCaptain();
         System.out.println("Trying to make another captain for your team:");
         Captain captain2 = Captain.getCaptain();
         if (captain1 == captain2) {
             System.out.println("captain1 and captain2 are same instance.");
         }
    }
}
```

Multi-threading in Java

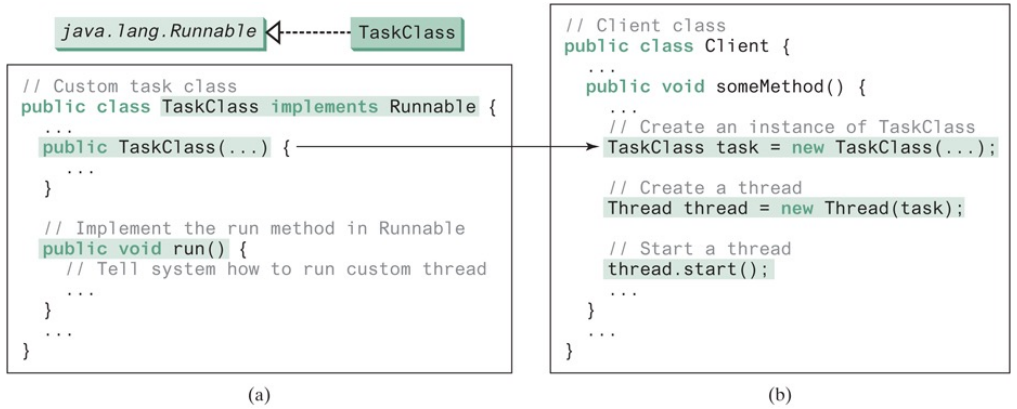
- Multithreading enables multiple tasks in a program to be executed concurrently
- One of the powerful features of Java is its built-in support for multithreading
- A thread is the flow of execution, from beginning to end, of a task.
- A thread provides the mechanism for running a task. With Java, you can launch multiple threads from a program concurrently. These threads can be executed simultaneously in multiprocessor systems



(a) Multiple threads running on multiple CPUs.
(b) Multiple threads share a single CPU.

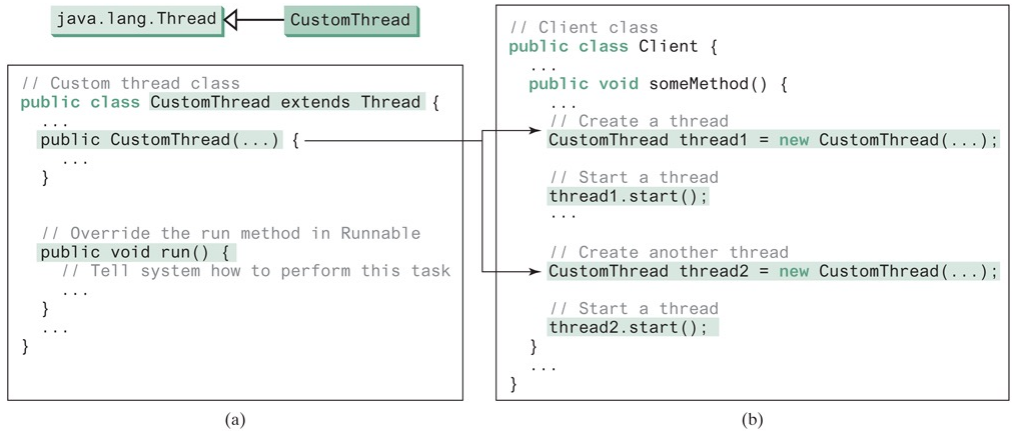
Multi-threading in Java

- A task class must implement the **Runnable interface**.
- A task must be run from a thread; tasks are objects.
- To create tasks, you first **define a class for tasks**, which implements the Runnable interface.
- The Runnable interface contains only the - run() method
- The **run() method** in a task specifies how to perform the task. It is automatically invoked by the JVM when the thread starts



Multi-threading in Java

- Thread class implements Runnable, you could also define a class that extends Thread and implements the run method
- Thread class contains the methods for controlling threads.
- A shared resource may become corrupted if it is accessed simultaneously by multiple threads. Thread synchronization is to coordinate the execution of the dependent threads.
- In multithreaded programs, a class is said to be thread-safe if an object of the class does not cause a race condition in the presence of multiple threads.
- To avoid race conditions, use the keyword synchronized to synchronize the method so that only one thread can access the method at a time.



Behavioral Pattern – State Pattern

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Suppose that you are dealing with a large-scale application where the codebase is rapidly growing. As a result, the situation becomes complex and you may need to introduce lots of if-else blocks/switch statements to guard the various conditions.
- The state pattern fits in such a context. It allows your objects to behave differently based on the current state, and you can define state-specific behaviors with different classes.
- Different types of behaviors can be encapsulated in a state object and the context is delegated to any of these states. When a context's internal states change, its behavior also changes.

Real-World Example

- Consider the scenario of a network connection—a TCP connection. An object can be in various states; for example, a connection might already be established, the connection might be closed, or the object already started listening through the connection. When this connection receives a request from other objects, it responds as per its present state.
- The functionalities of a traffic signal or a television (TV) can also be considered in this category; for example, you can change channels if the TV is already in a switched-on mode. It will not respond to the channel change requests if it is in a switched-off mode.

Computer-World Example

Suppose that you have a job-processing system that can process a certain number of jobs at a time.

When a new job appears, either the system processes the job, or it signals that the system is busy with the maximum number of jobs that it can process at one time.

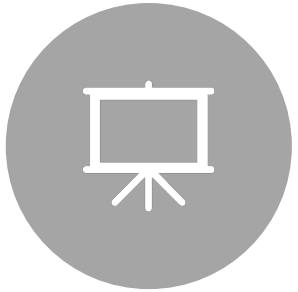
In other words, the system sends a busy signal when its total number of job-processing capabilities has been reached.

Illustration

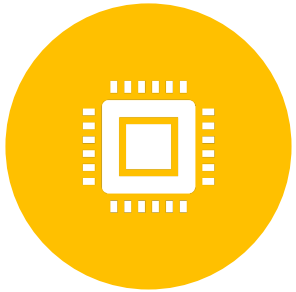
- Suppose that you have a remote control to support the operations of a TV.
- You can simply assume that at any given time, the TV is in either of these three states: On, Off, or Mute.
- Initially, the TV is in the Off state. When you press the On button on the remote control, the TV goes into the On state. If you press the Mute button, it goes into the Mute state.
- You can assume that if you press the Off button when the TV is already in the Off state, or if you press the On button when the TV is already in the On state, or if you press the Mute button when the TV is already in Mute mode, there is no state change for the TV.

Key Characteristics

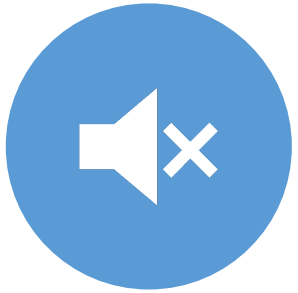
For a state-specific behavior, you have separate classes. For example, here you have classes like On, Off, and Mute.



The TV class is the main class here (the word main does not mean that it includes the main() method) and the client code only talks to it. In design pattern terms, TV is the context class here.



Operations defined in the TV class are delegating the behavior to the current state's object implementation.



PossibleState is the interface that defines the methods/operations that are called when you own an object. On, Off, and Mute are concrete states that implement this interface.

```
interface PossibleState {
    void pressOnButton(TV context);
    void pressOffButton(TV context);
    void pressMuteButton(TV context);
}

//Off state
class Off implements PossibleState {
    //User is pressing Off button when the TV is in Off state
    @Override
    public void pressOnButton(TV context) {
        System.out.println("You pressed On button. Going from Off to On
                            state.");
        context.setCurrentState(new On());
        System.out.println(context.getCurrentState().toString());
    }
    //TV is Off already; user is pressing Off button again
    @Override
    public void pressOffButton(TV context) {
        System.out.println("You pressed Off button. TV is already in Off
                            state.");
    }
    //User is pressing Mute button when the TV is in Off state
    @Override
    public void pressMuteButton(TV context) {
        System.out.println("You pressed Mute button. TV is already in Off
                           state, so Mute operation will not work."); }

    public String toString() {
        return "\t**TV is switched off now.**";
   }
}
```

```
//On state
class On implements PossibleState {
    //TV is On already, user is pressing On button again
    @Override
    public void pressOnButton(TV context) {
        System.out.println("You pressed On button. TV is already in On
                           state.");
    }
    //User is pressing Off button when the TV is in On state
    @Override
    public void pressOffButton(TV context) {
        System.out.println("You pressed Off button. Going from On to Off
                            state.");
        context.setCurrentState(new Off());
        System.out.println(context.getCurrentState().toString());
    }
    //User is pressing Mute button when the TV is in On state
    @Override
    public void pressMuteButton(TV context) {
        System.out.println("You pressed Mute button. Going from On mode.");
        context.setCurrentState(new Mute());
        System.out.println(context.getCurrentState().toString());
    }
    public String toString() {
        return "\t**TV is switched on now.**";
    }
}
```

`class Mute implements PossibleState { }` works in a similar way.

```
class TV {
    private PossibleState currentState;
    public TV() {
        //Initially TV is initialized with Off state
        this.setCurrentState(new Off());
    }
    public PossibleState getCurrentState() {
        return currentState;
    }
    public void setCurrentState(PossibleState currentState) {
        this.currentState = currentState;
    }
    public void pressOffButton() {
        currentState.pressOffButton(this);//Delegating the state
    }
    public void pressOnButton() {
        currentState.pressOnButton(this);//Delegating the state }
    public void pressMuteButton() {
        currentState.pressMuteButton(this);//Delegating the state }
}

//Client
public class StatePatternExample {
    public static void main(String[] args) {
        System.out.println("***State Pattern Demo***\n");
        TV tv = new TV();
        System.out.println("User is pressing buttons in the following
                            sequence:");
        System.out.println("Off->Mute->On->On->Mute->Mute->Off\n");
        tv.pressOffButton(); //TV is initially off already.
        tv.pressMuteButton();
        tv.pressOnButton(); //turn on the TV
        tv.pressOnButton(); //TV is already on
        tv.pressMuteButton(); //Putting the TV in Mute mode
        tv.pressMuteButton();
        tv.pressOffButton(); //turn off the TV
    }
}
```

Behavioral Pattern – Template Method Pattern

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

In a template method, you define the minimum or essential structure of an algorithm. Then you defer some responsibilities to the subclasses.

The key intent is that you can redefine certain steps of an algorithm, but those changes should not impact the basic flow of the algorithm.

So, this design pattern is useful when you implement a multistep algorithm and you want to allow customization through subclasses.

Real-World Example

- Suppose that you are ordering a pizza from a restaurant.
- For the chef, the basic preparation of the pizza is the same; he includes some final toppings based on customer choice.
- For example, you can opt for a veggie pizza or a non-veggie pizza. You can also choose toppings like bacons, onions, extra cheese, mushrooms, and so forth. The chef prepares the final product according to your preferences.

Computer-World Example

- Suppose that you are making a program to design engineering courses.
- Let's assume that the first semester is common for all streams. In subsequent semesters, you need to add new papers/subjects to the application based on the course.
- This pattern makes sense when you want to avoid duplicate codes in your application. At the same time, you may want to allow subclasses to change some specific details of the base class workflow to provide varying behaviors in the application.

Illustration

Assume that each engineering student needs to complete the course of Mathematics and then Soft skills (This subject may deal with communication skills, character traits, people management skills etc.) in their initial semesters to obtain the degree. Later you will add special papers to these courses (Computer Science or Electronics)

To serve the purpose, a method completeCourse() is defined in an abstract class BasicEngineering; this method is final, so that subclasses of BasicEngineering cannot override the completeCourse() method to alter the sequence of course completion order.

Two other concrete classes - ComputerScience and Electronics are the subclasses of BasicEngineering class and they are completing the abstract method completeSpecialPaper() as per their needs.

```
abstract class BasicEngineering {
    //Making the method final to prevent overriding.
    public final void completeCourse() {
        //The course needs to be completed in the following sequence
        //1.Math-2.SoftSkills-3.Special Paper
        //Common Papers:
        completeMath();
        completeSoftSkills();
        completeSpecialPaper(); //Specialization Paper:
    }
    private void completeMath() {
        System.out.println("1.Mathematics");
    }
    private void completeSoftSkills() {
        System.out.println("2.SoftSkills"); }
    public abstract void completeSpecialPaper();
}

class ComputerScience extends BasicEngineering {
    @Override
    public void completeSpecialPaper() {
        System.out.println("3.Object-Oriented Programming");
    }
}

class Electronics extends BasicEngineering {
    @Override
    public void completeSpecialPaper() {
        System.out.println("3.Digital Logic and Circuit Theory");
    }
}
```

```
public class TemplateMethodPatternExample {
   public static void main(String[] args) {
       System.out.println("***Template Method
                                Pattern Demo***\n");
       BasicEngineering preferrredCourse =
                          new ComputerScience();
       System.out.println("Computer Sc. course
          will be completed in following order:");
        preferrredCourse.completeCourse();
        System.out.println();
         preferrredCourse = new Electronics();
         System.out.println("Electronics course
           will be completed in following order:"
         preferrredCourse.completeCourse();
    }
}
```

Structural patterns

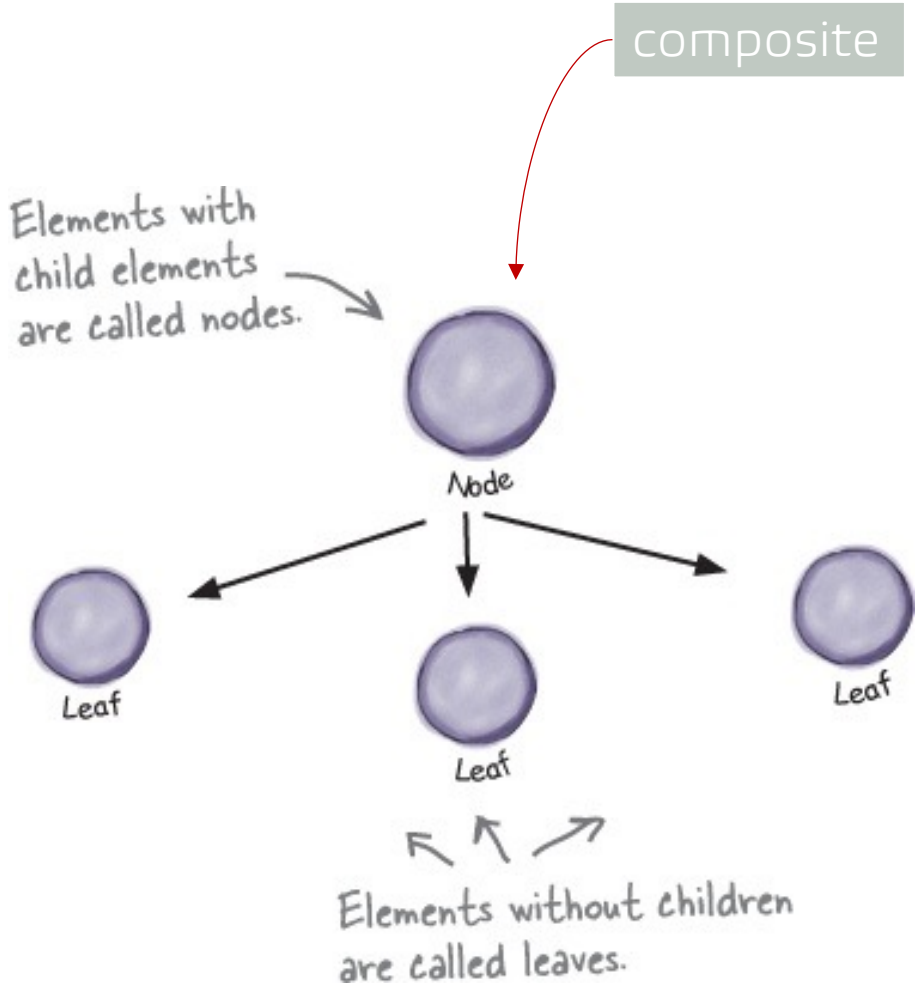
Structural patterns are concerned with how classes and objects are composed to form larger structures

This pattern is particularly useful for making independently developed class libraries work together

Composite Pattern

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies.

Composite lets clients treat individual objects and compositions of objects uniformly.



Composite Pattern - applicability

- Use the Composite pattern when
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.
  - Consider a shop that sells different kinds of dry fruits and nuts; let's say cashews, dates, and walnuts. Each of these items is associated with a certain price. Let's assume that you can purchase any of these individual items or you can purchase "gift packs" (or boxed items) that contain different items. In this case, the cost of a packet is the sum of its component parts. The composite pattern is useful in a similar situation, where you treat both the individual parts and the combination of the parts in the same way so that you can process them uniformly.

Real-World Example

You can also think of an organization that consists of many departments.

In general, an organization has many employees. Some of these employees are grouped together to form a department, and those departments can be further grouped together to build the final structure of the organization.

Computer-World Example

- Any tree data structure can follow this concept. Clients can treat the leaves of the tree and the non-leaves (or branches of the tree) in the same way.
- This pattern is commonly seen in various UI frameworks.
- In Java, the generic Abstract Window Toolkit (AWT) container object is a component that can contain other AWT components. For example, in java.awt.Container class (which extends java.awt.Component) you can see various overloaded version of add(Component comp) method
- JavaFX, scene graph, panes and UI controls
- Android, component tree, view groups and views

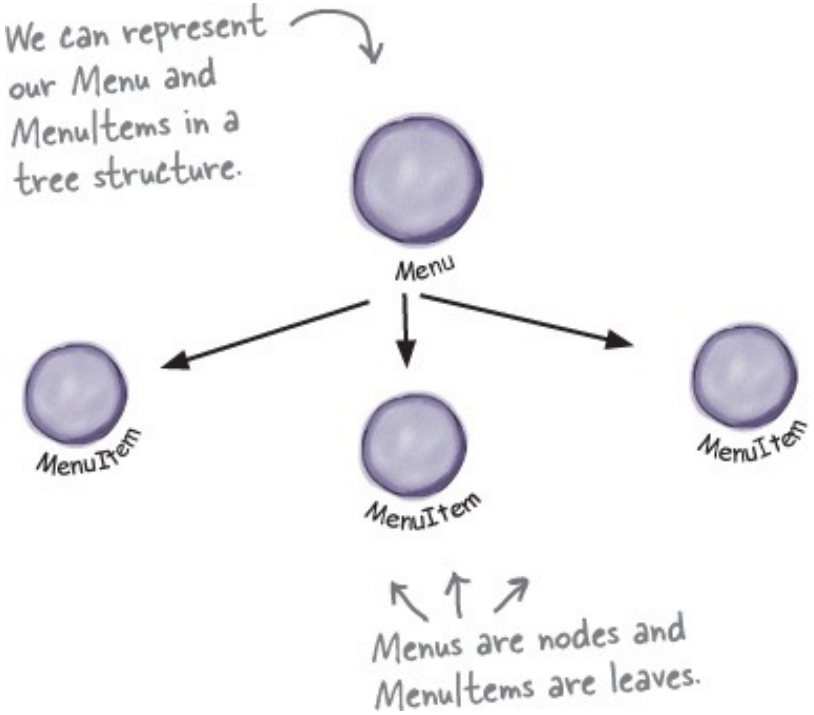
Illustration

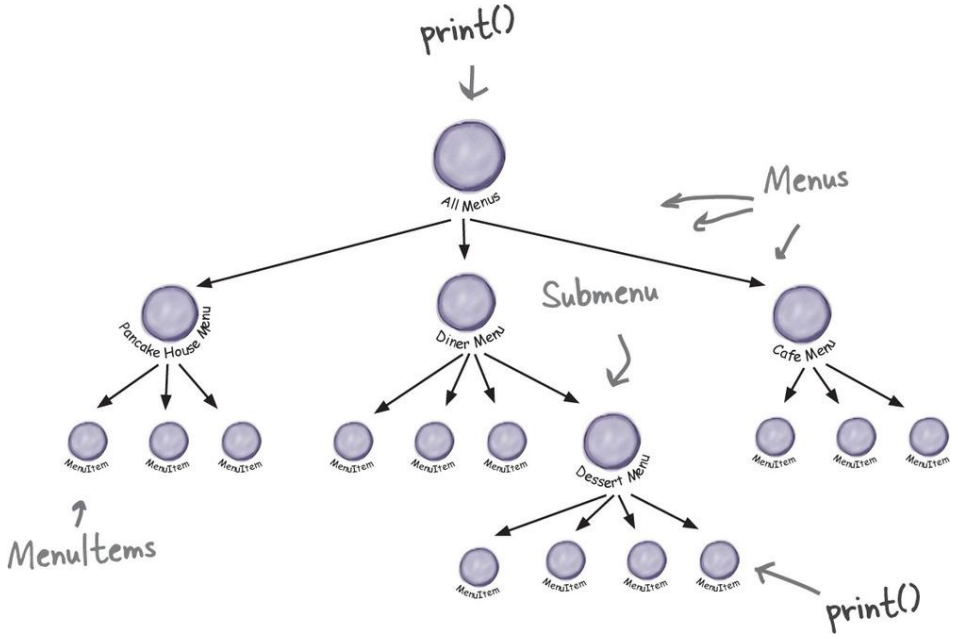
- Let's consider a college organization.
- Assume that there is a principal and two heads of departments – one for computer science (CS) and one for mathematics (Math).
- In the Math department, there are two teachers (or professors/lecturers), and in the CS department, there are three teachers (or professors/lecturers).

```
interface IEmployee {
    void printStructures();
    int getEmployeeCount();
}
class CompositeEmployee implements IEmployee {
    private int employeeCount = 0;
    private String name;
    private String dept;
    private List<IEmployee> controls; //container for child
    public CompositeEmployee(String name, String dept) {
        this.name = name;
        this.dept = dept;
        controls = new ArrayList<IEmployee>();
    }
    public void addEmployee(IEmployee e) {
        controls.add(e);
    }
    public void removeEmployee(IEmployee e) {
        controls.remove(e);
    }
    @Override
    public void printStructures() {
        System.out.println("\t" + this.name + " works in "
                           + this.dept);
        for (IEmployee e: controls) {
            e.printStructures();
        }
    }
    @Override
    public int getEmployeeCount() {
        employeeCount = controls.size();
        for (IEmployee e: controls) {
            employeeCount += e.getEmployeeCount();
        }
        return employeeCount;
    }
}
```

```
class Employee implements IEmployee {
    private int employeeCount = 0;
    private String name;
    private String dept;
    public Employee(String name, String dept) {
        this.name = name;
        this.dept = dept;
    }
    @Override
    public void printStructures() {
        System.out.println("\t\t" + this.name + " works in " + this.dept);
    }
    @Override
    public int getEmployeeCount() {
        return employeeCount;
    }}
class CompositePatternExample { //the client
    public static void main(String[] args) {
        Employee mathTeacher1 = new Employee("Math Teacher-1","Math"); //leaf
        Employee mathTeacher2 = new Employee("Math Teacher-2","Math");
        Employee csTeacher1 = new Employee("CS Teacher-1", "CS");
        ...
        CompositeEmployee hodMath =  //composite node
              new CompositeEmployee("Mrs. S. Das(HOD-Math)","Math");
        CompositeEmployee hodCS =
              new CompositeEmployee( "Mr. V. Sarcar(HOD-CS)", "CS");
        CompositeEmployee principal =
             new CompositeEmployee("Dr. S.Som(Principal)",
                                   "Planning-Supervising-Managing");
        hodMath.addEmployee(mathTeacher1);
        hodMath.addEmployee(mathTeacher2);
        hodCS.addEmployee(cseTeacher1);
        ...
        principal.addEmployee(hodMath);
        ...
        principal.printStructures(); //Prints the complete structure
              hodCS.printStructures(); //Prints the details of CS department
        mathTeacher1.printStructures(); //Prints the leaf node
        ...
    }}
```

Composite Pattern – Another Example





Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

These patterns characterize complex control flow that's difficult to follow at run-time.

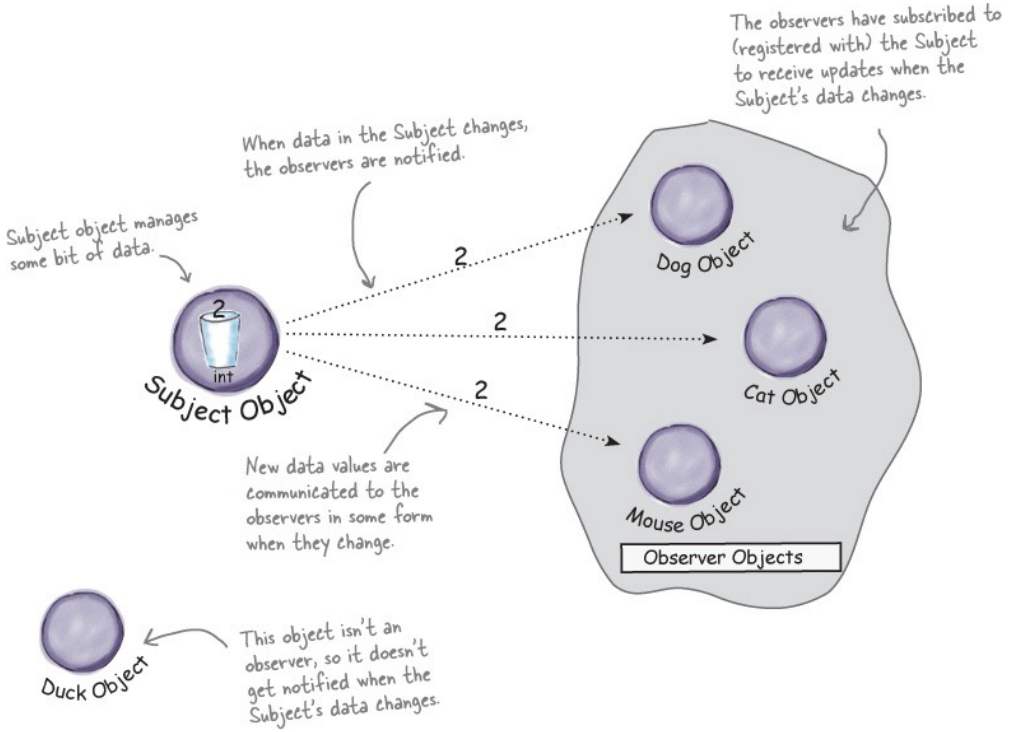
They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

Observer – a.k.a Dependents, Publish-Subscribe

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- Observer pattern intent to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The Observer pattern defines and maintains a dependency between objects.
- The classic example of Observer is **Model/View/Controller**, where all views of the model are notified whenever the model's state changes.

Observer Pattern

- In this pattern, there are many observers (objects) that are observing a particular subject (also an object).
- Observers register themselves to a subject to get a notification when there is a change made inside that subject.
- When they lose interest of the subject, they simply unregister from the subject. It is also referred to as the publish-subscribe pattern.



Observer Pattern – applicability

- Use the Observer pattern in any of the following situations
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them 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 who these objects are. In other words, you don't want these objects tightly coupled.

Real-World Example

- Think about a celebrity who has many followers on social media.
- Each of these followers wants all the latest updates from their favorite celebrity. So, they follow the celebrity until their interest wanes.
- When they lose interest, they simply do not follow that celebrity any longer.
- You can think each of these fans or followers as an observer and the celebrity as a subject.

Computer-World Example

In the world of computer science, consider a simple UI-based example.

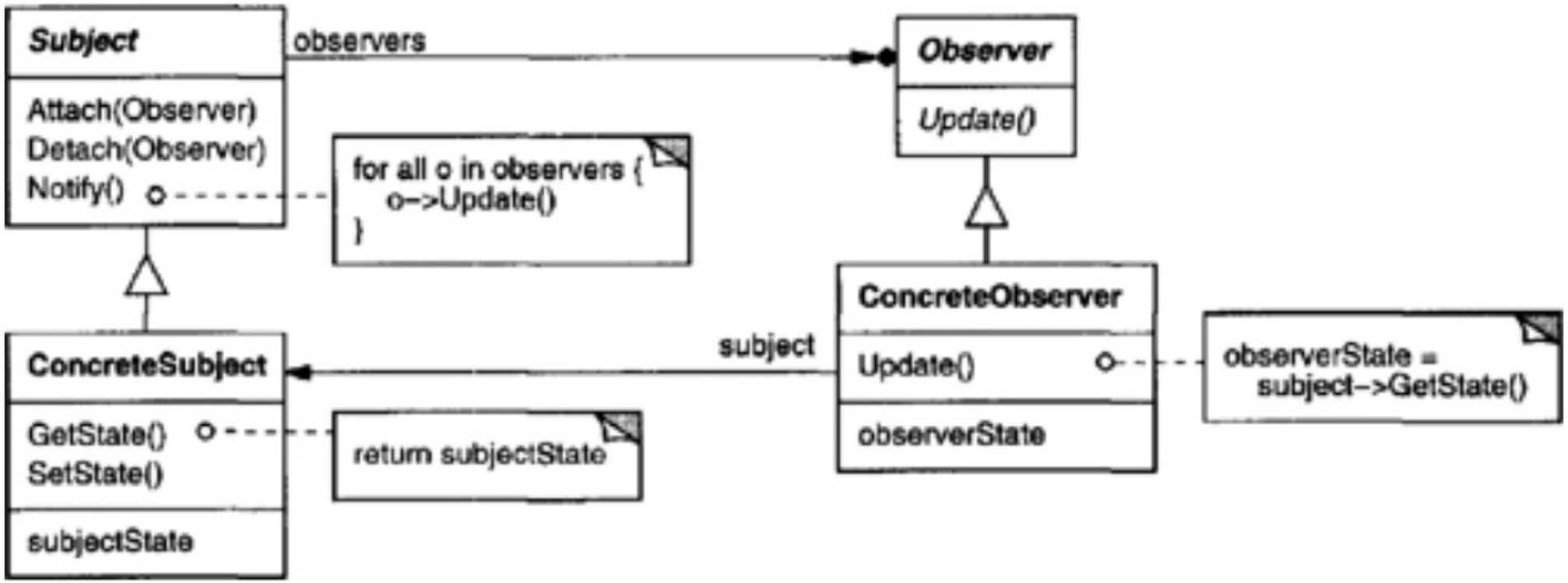
Let's assume that this UI is connected to a database.

A user can execute a query through that UI, and after searching the database, the result is returned in the UI.

Here you segregate the UI from the database in such a way that if a change occurs in the database, the UI is notified, and it updates its display according to the change.

In general, you see the presence of this pattern in event-driven software.

Observer Pattern – Structure



Illustration

Let's consider a subject maintains a list for all its registered users.

Our observers want to receive notification when a flag value changes in the subject.

The observers are getting the notifications when flag values are changed to 5, 50, and 100, respectively.

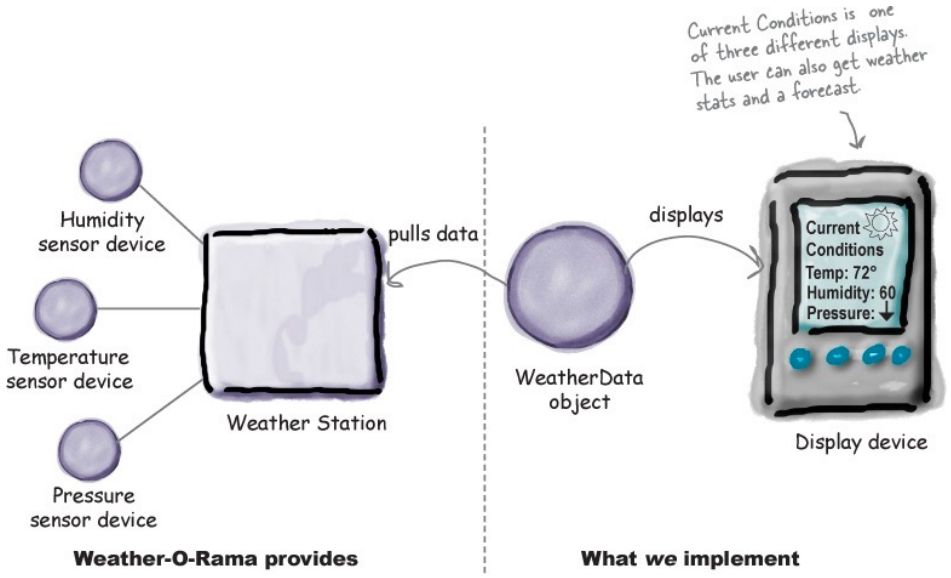
```
interface Observer {
    void update(int updatedValue);
}

class ObserverType1 implements Observer {
    String nameOfObserver;
    public ObserverType1(String name) {
        this.nameOfObserver = name;
    }
    @Override
    public void update(int updatedValue) {
        System.out.println(nameOfObserver +
           " has received an alert: Updated
                 myValue in Subject is: "+ updatedValue);
    }
}
class ObserverType2 implements Observer {
    String nameOfObserver;
    public ObserverType2(String name) {
        this.nameOfObserver = name;
    }
    @Override
    public void update(int updatedValue) {
        System.out.println(nameOfObserver + " has received
               an alert: The current value of myValue
               in Subject is: "+ updatedValue);
    }
}
```

```
interface SubjectInterface {
    void register(Observer anObserver);
    void unregister(Observer anObserver);
    void notifyRegisteredUsers(int notifiedValue);
}
class Subject implements SubjectInterface {
    private int flag;
    List<Observer> observerList = new ArrayList<Observer>();
    public int getFlag() {
        return flag;
    }
    public void setFlag(int flag) {
        this.flag = flag;
        notifyRegisteredUsers(flag); //flag value changed
    }
   @Override
    public void register(Observer anObserver) {
        observerList.add(anObserver);
    }
    @Override
    public void unregister(Observer anObserver) {
        observerList.remove(anObserver);
    }
    @Override
    public void notifyRegisteredUsers(int updatedValue) {
        for (Observer observer : observerList)
            observer.update(updatedValue);
    }
}
```

Observer Pattern

```
//Sample code segment of a client program
Observer myObserver1 = new ObserverType1("Roy");
Observer myObserver2 = new ObserverType1("Kevin");
Observer myObserver3 = new ObserverType2("Bose");
Subject subject = new Subject();
subject.register(myObserver1);
subject.register(myObserver2);
subject.register(myObserver3);
//generate a notification to the observers
subject.setFlag(5);
subject.unregister(myObserver1);
```

Another Example – Weather Monitor System




```
currentConditionsDisplay.update(temp, humidity, pressure);
statisticsDisplay.update(temp, humidity, pressure);
forecastDisplay.update(temp, humidity, pressure);
```

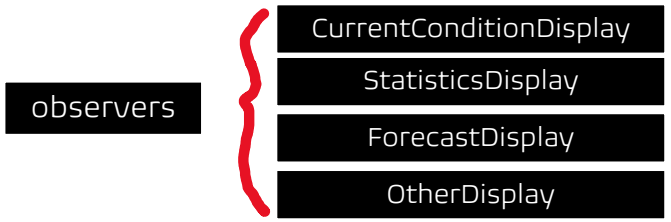
Weather Monitor System

```
public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }
```

WeatherData now implements the Subject interface.

We've added an ArrayList to hold the Observers, and we create it in the constructor.

```
//all observers implement update()
public void notifyObservers() {
    for (Observer observer: observers)
          observer.update(temperature, humidity, pressure);
}
```



The Power of Loose Coupling

- When two objects are loosely coupled, they can interact, but have very little knowledge of each other.
- The Observer Pattern provides an object design where subjects and observers are loosely coupled – the only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer
  - We can add new observers at any time.
  - We never need to modify the subject to add new types of observers.
  - We can reuse subjects or observers independently of each other.
  - Changes to either the subject or an observer will not affect the other.

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method	Description
void	**addListener** (**InvalidationListener** listener)	Adds an `InvalidationListener` which will be notified whenever the `Observ` invalid.
void	**removeListener**	Removes the given listener from the list of listeners, that are notified whenever t `Observable` becomes invalid.

Another Example – Javafx Observable Interface

- ObservableList Interface is the sub-Interface of Observable Interface

Android – ObservableList and ObserableArrayList

Public methods	
`abstract void`	`addOnListChangedCallback(OnListChangedCallback<? extends ObservableList<T>> callback)` Adds a callback to be notified when changes to the list occur.
`abstract void`	`removeOnListChangedCallback(OnListChangedCallback<? extends ObservableList<T>> callback)` Removes a callback previously added.

ObservableArrayList

```
public class ObservableArrayList
    extends ArrayList<T> implements ObservableList<T>
```

java.lang.Object
↳ java.util.AbstractCollection<T>
↳ java.util.AbstractList<T>
↳ java.util.ArrayList<T>
↳ android.databinding.ObservableArrayList<T>

An ObservableList implementation using ArrayList as an implementation

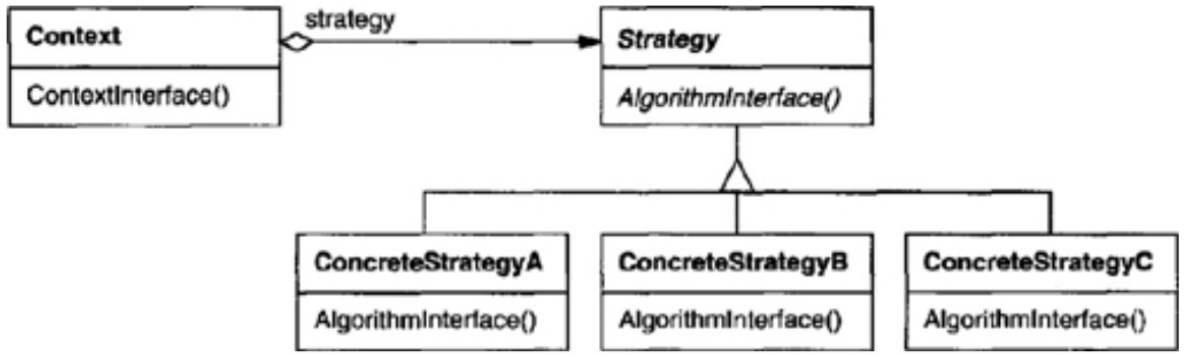
adapter.notifyDataSetChanged()

Strategy Pattern

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Suppose there is an application where you have multiple algorithms and each of these algorithms can perform a specific task.
- A client can dynamically pick any of these algorithms to serve its current need.
- The strategy pattern suggests that you implement these algorithms in separate classes.
- When you encapsulate an algorithm in a separate class, you call it a strategy.

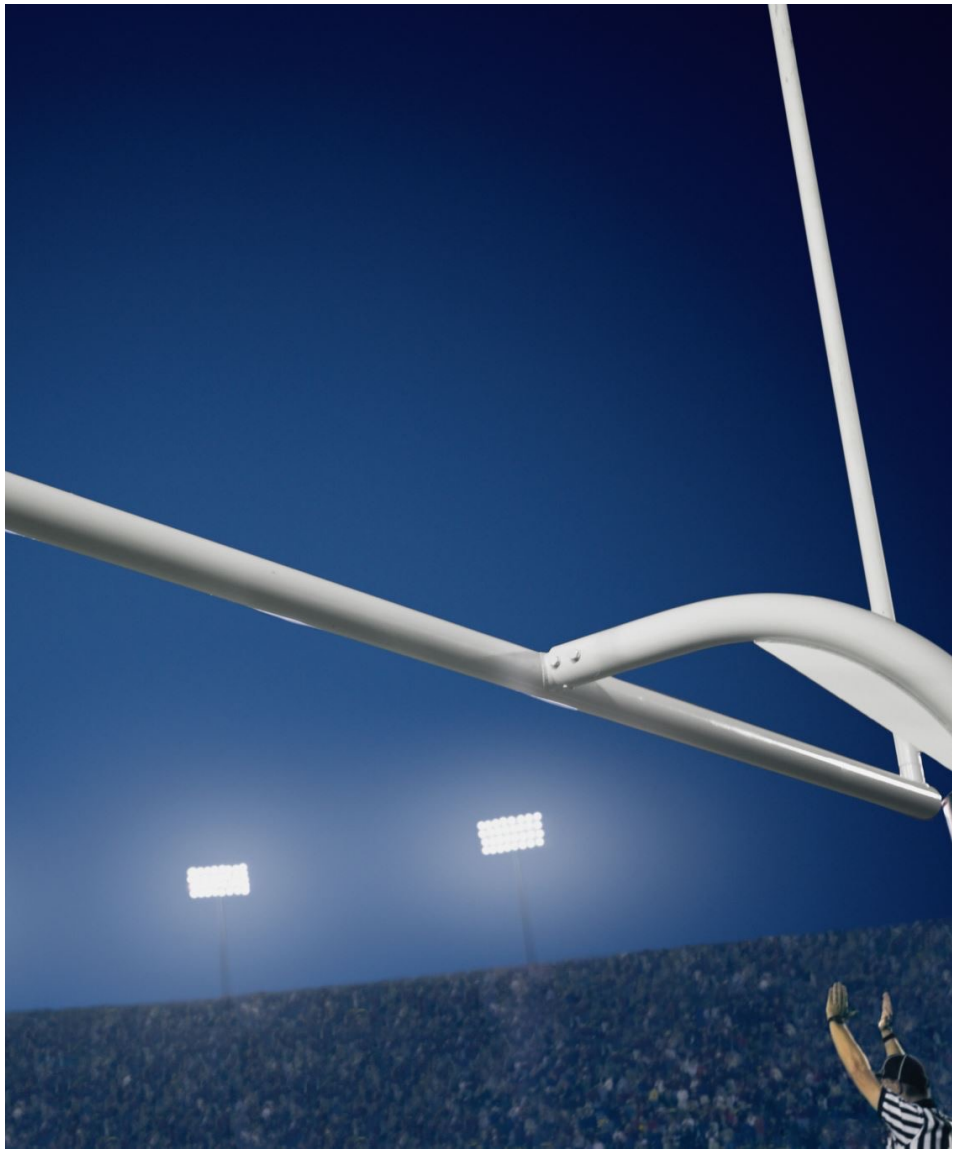
Strategy Pattern - structure

- An object that uses the strategy object is often referred to as a context object.
- These "algorithms" are also called behaviors in some applications.



Real-World Example

- Generally, at the end of a soccer match, if team A is leading 1–0 over team B, instead of attacking they become defensive to maintain the lead.
- On the other hand, team B goes for an all-out attack to score the equalizer.



Computer-World Example

- Suppose that you have a list of integers, and you want to sort them.
- You do this by using various algorithms; for example, Bubble Sort, Merge Sort, Quick Sort, Insertion Sort, and so forth. So, you can have a sorting algorithm with many different variations (implementations)
- Now you can implement each of these variations (algorithms) in separate classes and pass the objects of these classes in client code to sort your integer list.

Illustration

- The strategy pattern encourages you to use object composition instead of subclassing. So, it suggests you do not override parent class behaviors in different subclasses. Instead, you put these behaviors in separate classes (called a strategy) that share a common interface.
- The client class only decides which algorithm to use; the context class does not decide that.
- In the following example, Vehicle class is an abstract class that plays the role of a context. Boat and Aeroplane are two concrete implementations of the Vehicle class in which they are associated with different behaviors: one travels through water and the other one travels through air.
- These behaviors are placed in two concrete classes: AirTransport and WaterTransport. These classes share a common interface, TransportMedium.

```
public abstract class Vehicle {
    TransportMedium transportMedium;
    public void showTransportMedium() {
        transportMedium.transport();
    }
    public void commonJob() {
        System.out.println("We all can be used to transport");
    }
    public abstract void showMe();
}

public class Boat extends Vehicle {
    public Boat() {
        transportMedium = new WaterTransport();
    }
    @Override
    public void showMe() {
        System.out.println("I am a boat.");
    }
}

public class Aeroplane extends Vehicle {
    public Aeroplane() {
        transportMedium = new AirTransport();
    }
    @Override
    public void showMe() {
        System.out.println("I am an aeroplane.");
    }
}

public interface TransportMedium {
    public void transport();
}
```

```
public class WaterTransport implements TransportMedium {
    @Override
    public void transport() {
        System.out.println("I am transporting in water.");
    }
}

public class AirTransport implements TransportMedium {
    @Override
    public void transport() {
        System.out.println("I am transporting in air.");
    }
}
```

```
//Client code
public class StrategyPatternExample {
    public static void main(String[] args) {
        Vehicle vehicleContext = new Boat();
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();
        System.out.println("________");
        vehicleContext = new Aeroplane();
        vehicleContext.showMe();
        vehicleContext.showTransportMedium();
} }
```

MVC Pattern

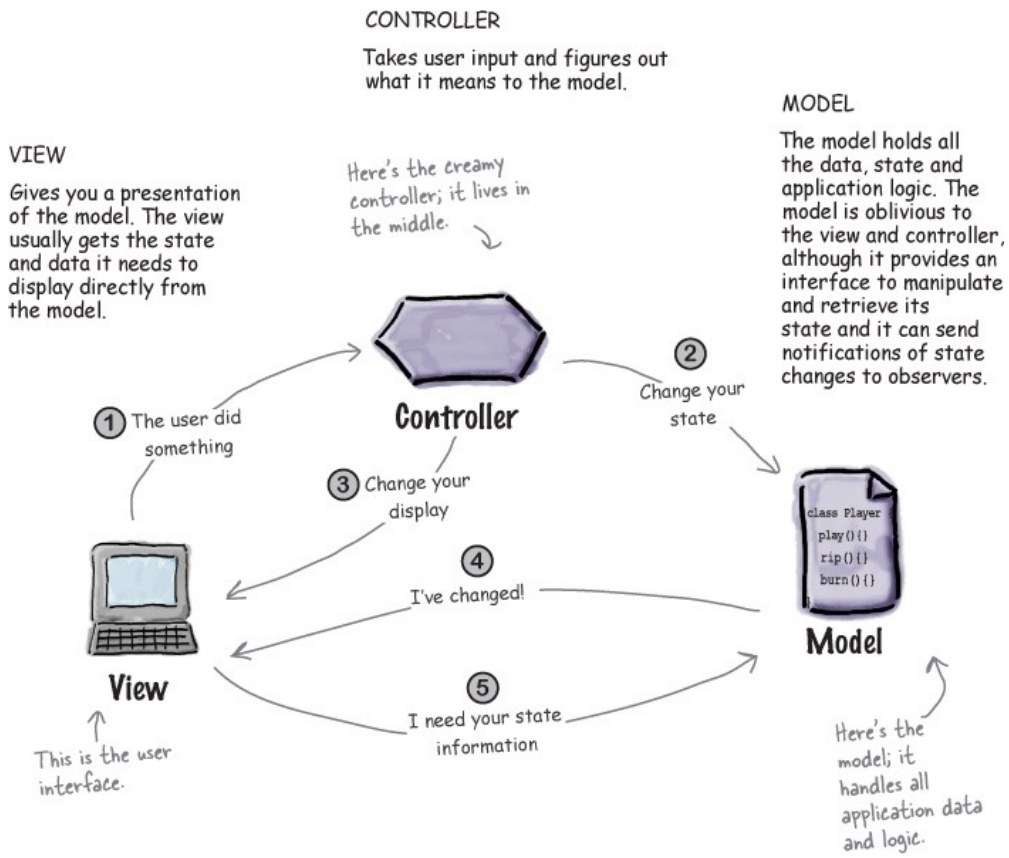
- Model-View-Controller (MVC) is an architectural pattern.
- The use of this pattern is commonly seen in web applications or when we develop powerful user interfaces.
- It is important to note that some developers believe that it is not a true design pattern, instead, they prefer to call it "MVC architecture."
- Here you separate the user interface logic from the business logic and decouple the major components in such a way that they can be reused efficiently. This approach also promotes parallel development.

Real-World Example

- Let's revisit our template method pattern's real-life example. But this time you interpret it differently.
- In a restaurant, based on customer input, a chef can vary the taste and make the final products. The customers do not place their orders directly to the chef.
- The customers see the menu card (view), may consult with the waiter/waitress, and place their order.
- The waiter/waitress passes the order slip to the chef, who gathers the required materials from the restaurant's kitchen (similar to storehouses/computer databases).
- Once prepared, the waiter/waitress carries the plate to the customer's table. So, you can consider the role of the waiter/waitress as the controller, and the chef with their kitchen as the model (and the food preparation materials as data).

Model View Controller (MVC)

- MVC includes a set of patterns working together
- Observer,
- Composite,
- Strategy

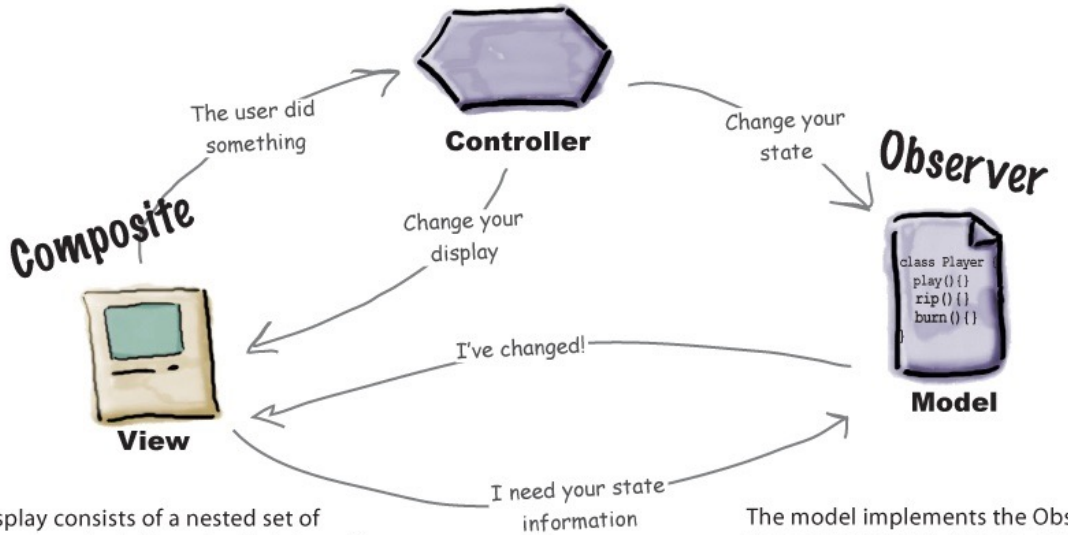


MVC Pattern

- Observer
- Composite
- Strategy

Strategy

The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.