DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING
CONCORDIA UNIVERSITY

Java Design Patterns

Emil Vassev, Joey Paquet : 2006-2012

Outline

- Introduction to Design Patterns
  - Pattern's Elements
  - Types of Design Patterns
- Java Design Patterns
  - The Factory Pattern
  - The Abstract Factory Pattern
  - The Builder Pattern
  - The Prototype Pattern
  - The Singleton Pattern
  - The Adapter Pattern
  - The Bridge Pattern
  - The Composite Pattern
  - Java BluePrints Patterns Catalog

Design Patterns

:: What is a 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." [1]

[Christopher Alexander]

Design patterns capture the *best practices of experienced object-oriented software developers.*

Design patterns are solutions to general software development problems.

In general, a pattern has four essential elements.

- The pattern name
- The problem
- The solution
- The consequences

Design Patterns

:: Pattern's Elements – The Pattern Name

The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.

- Naming a pattern immediately increases the design vocabulary. It lets us design at a higher level of abstraction.
- Having a vocabulary for patterns lets us talk about them.
- It makes it easier to think about designs and to communicate them and their trade-offs to others.

The **problem** describes when to apply the pattern.

- It explains the problem and its context.
- It might describe specific design problems such as how to represent algorithms as objects.
- It might describe class or object structures that are symptomatic of an inflexible design.
- Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations.

- The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations.
- Instead, the pattern 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** are the results and trade-offs of applying the pattern.

- The consequences for software often concern space and time trade-offs.
- They may address language and implementation issues as well.
- Since 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.
- Listing these consequences explicitly helps you understand and evaluate them

Design Patterns

:: Types

Erich Gamma, Richard Helm, Ralph Johnson and John Vlisides in their **Design Patterns** book define 23 design patterns divided into three types:

- *Creational patterns* are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- *Structural patterns* help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- *Behavioral patterns* help you define the communication between objects in your system and how the flow is controlled in a complex program.

Java Design Patterns

:: Why Use Patterns with Java?

- They have been proven. Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work.
- They are reusable. Patterns provide a ready-made solution that can be adapted to different problems as necessary.
- They are expressive. Patterns provide a common vocabulary of solutions that can express large solutions succinctly.
- J2EE provides built in patterns.

- The creational patterns deal with the best way to create instances of objects.
- In Java, the simplest way to create an instance of an object is by using the **new** operator.

```
Fred = new Fred();  //instance of Fred class
```

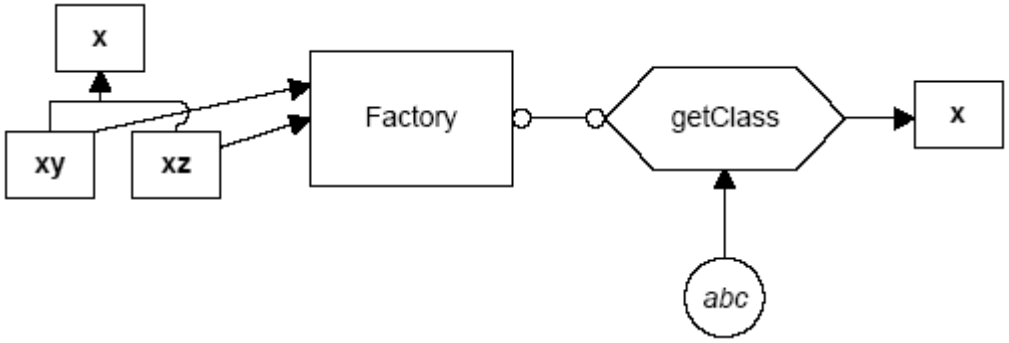
- This amounts to hard coding, depending on how you create the object within your program.
- In many cases, the exact nature of the object that is created could vary with the needs of the program and abstracting the creation process into a special “creator” class can make your program more flexible and general.

- **The Factory Pattern** provides a simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data that are provided.
- **The Abstract Factory Pattern** provides an interface to create and return one of several families of related objects.
- **The Builder Pattern** separates the construction of a complex object from its representation.
- **The Prototype Pattern** starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.
- **The Singleton Pattern** is a class of which there can be no more than one instance. It provides a single global point of access to that instance.

The Factory Pattern

:: How does it Work?

The Factory pattern returns an instance of one of several possible classes depending on the data provided to it.



- Here, **x** is a base class and classes **xy** and **xz** are derived from it.
- The **Factory** is a class that decides which of these subclasses to return depending on the arguments you give it.
- The ***getClass()*** method passes in some value *abc*, and returns some instance of the class **x**. Which one it returns doesn't matter to the programmer since they all have the same methods, but different implementations.

The Factory Pattern

:: The Base Class

- Let's consider a simple case where we could use a Factory class. Suppose we have an entry form and we want to allow the user to enter his name either as *"firstname lastname"* or as *"lastname, firstname"*.
- Let's make the assumption that we will always be able to decide the name order by whether there is a comma between the last and first name.

```
class Namer { //a class to take a string apart into two names
        protected String last;  //store last name here
        protected String first; //store first name here
        public String getFirst() {
                return first;    //return first name
        }
        public String getLast() {
                return last;     //return last name
        }
}
```

The Factory Pattern

:: The First Derived Class

In the *FirstFirst* class, we assume that everything before the last space is part of the first name.

```
class FirstFirst extends Namer {
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" ");   //find separating space
        if (i > 0) {
            first = s.substring(0, i).trim(); //left = first name
            last =s.substring(i+1).trim(); //right = last name
        } else {
            first = "" // put all in last name
            last = s; // if no space
        }
    }
}
```

The Factory Pattern

:: The Second Derived Class

In the *LastFirst* class, we assume that a comma delimits the last name.

```
class LastFirst extends Namer { //split last, first
   public LastFirst(String s) {
      int i = s.indexOf(","); //find comma
         if (i > 0) {
            last = s.substring(0, i).trim(); //left= last name
            first = s.substring(i + 1).trim(); //right= first name
         } else {
            last = s; // put all in last name
            first = ""; // if no comma
         }
     }
}
```

The Factory Pattern

:: Building the Factory

The Factory class is relatively simple. We just test for the existence of a comma and then return an instance of one class or the other.

```
class NameFactory {
    //returns an instance of LastFirst or FirstFirst
    //depending on whether a comma is found
    public Namer getNamer(String entry) {
        int i = entry.indexOf(","); //comma determines name order
        if (i>0)
            return new LastFirst(entry); //return one class
        else
            return new FirstFirst(entry); //or the other
    }
}
```

The Factory Pattern

:: Using the Factory

```
NameFactory nfactory = new NameFactory();
String sFirstName, sLastName;
….
private void computeName() {
  //send the text to the factory and get a class back
  namer = nfactory.getNamer(entryField.getText());
  //compute the first and last names using the returned class
  sFirstName = namer.getFirst();
  sLastName = namer.getLast();
}
```

The Factory Pattern

:: When to Use a Factory Pattern

You should consider using a Factory pattern when:

- A class can't anticipate which kind of class of objects it must create.
- A class uses its subclasses to specify which objects it creates.
- You want to localize the knowledge of which class gets created.

There are several similar variations on the factory pattern to recognize:

- The base class is abstract and the pattern must return a complete working class.
- The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
- Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

The Abstract Factory Pattern

:: How does it Work?

The Abstract Factory pattern is one level of abstraction higher than the factory pattern. This pattern returns one of several related classes, each of which can return several different objects on request. In other words, **the Abstract Factory is a factory object that returns one of several factories.**

One classic application of the abstract factory is the case where your system needs to support multiple "look-and-feel" user interfaces, such as Windows, Motif or Macintosh:

- You tell the factory that you want your program to look like Windows and it returns a GUI factory which returns Windows-like objects.
- When you request specific objects such as buttons, check boxes and windows, the GUI factory returns Windows instances of these visual interface components.

The Abstract Factory Pattern

:: A Garden Maker Factory?

Suppose you are writing a program to plan the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. However, no matter which kind of garden you are planning, you want to ask the same questions:

- What are good border plants?
- What are good center plants?
- What plants do well in partial shade?

We want a base *Garden* class that can answer these questions:

```
public abstract class Garden {
    public abstract Plant getCenter();
    public abstract Plant getBorder();
    public abstract Plant getShade();
}
```

The Abstract Factory Pattern

:: The Plant Class

The *Plant* class simply contains and returns the plant name:

```
public class Plant {
      String name;
      public Plant(String pname) {
        name = pname; //save name
      }
      public String getName() {
        return name;
      }
}
```

The Abstract Factory Pattern

:: A Garden Class

A Garden class simply returns one kind of each plant. So, for example, for the vegetable garden we simply write:

```
public class VegieGarden extends Garden {
    public Plant getShade() {
        return new Plant("Broccoli");
    }
    public Plant getCenter() {
        return new Plant("Corn");
    }
    public Plant getBorder() {
        return new Plant("Peas");
    }
}
```

The Abstract Factory Pattern

:: A Garden Maker Class – The Abstract Factory

We create a series of *Garden* classes - ***VegieGarden***, ***PerennialGarden, and AnnualGarden,*** each of which returns one of several Plant objects. Next, we construct our **abstract factory** to return an object instantiated from one of these *Garden* classes and based on the string it is given as an argument:

```
class GardenMaker {
    //Abstract Factory which returns one of three gardens
   private Garden gd;
   public Garden getGarden(String gtype) {
       gd = new VegieGarden(); //default
       if(gtype.equals("Perennial"))
              gd = new PerennialGarden();
       if(gtype.equals("Annual"))
              gd = new AnnualGarden();
       return gd;
   }
}
```

The Abstract Factory Pattern

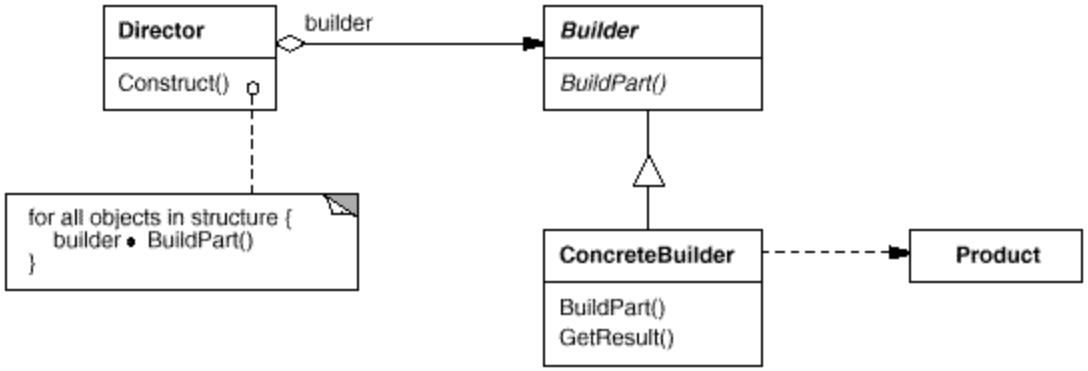
:: Consequences of Abstract Factory

- One of the main purposes of the **Abstract Factory** is that it isolates the concrete classes that are generated.
- The actual class names of these classes are hidden in the factory and need not be known at the client level at all.
- Because of the isolation of classes, you can change or interchange these product class families freely.
- Since you generate only one kind of concrete class, this system keeps you for inadvertently using classes from different families of products.
- While all of the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some derived classes from having additional methods that differ from the methods of other classes.

The Builder Pattern

:: How does it Work? - I

The Builder Pattern separates the construction of a complex object from its representation so that the same construction process can create different representations.



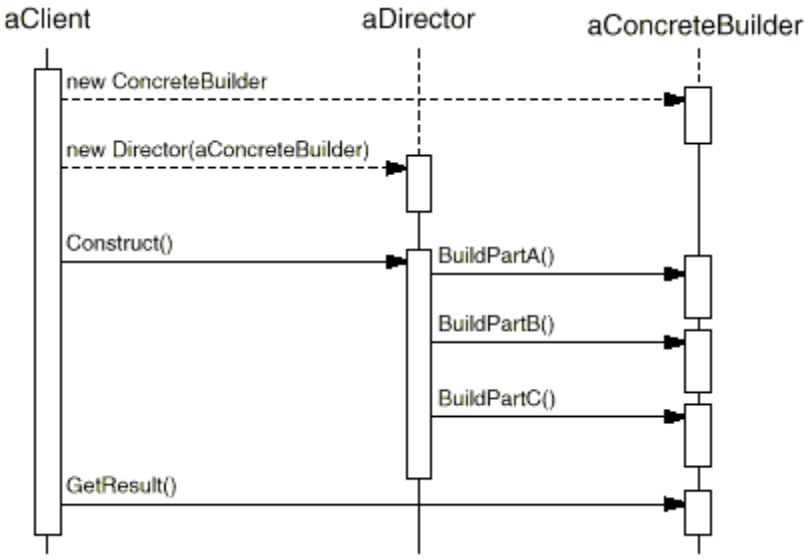
- **Builder** - specifies an abstract interface for creating parts of a **Product** object.
- **ConcreteBuilder** - constructs and assembles parts of the product by implementing the Builder interface. Also, it defines and keeps track of the representation it creates and provides an interface for retrieving the product .
- **Director** - constructs an object using the Builder interface.
- **Product** - represents the complex object under construction.

The Builder Pattern

:: How does it Work? - II

- The client creates the *Director* object and configures it with the desired *Builder* object.
- *Director* notifies the builder whenever a part of the product should be built.
- *Builder* handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

The following interaction diagram illustrates how *Builder* and *Director* cooperate with a client.



The Builder Pattern

:: Applicability of Builder Pattern

Use the Builder pattern when:

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- The construction process must allow different representations for the object that is constructed.

The Builder Pattern

:: Consequences of Builder Pattern

- A **Builder** lets you vary the internal representation of the product it builds. It also hides the details of how the product is assembled.
- Each specific builder is independent of the others and of the rest of the program. This improves modularity and makes the addition of other builders relatively simple.
- Because each builder constructs the final product step-by-step, depending on the data, you have more control over each final product that a Builder constructs.
- A **Builder pattern is somewhat like an Abstract Factory** pattern in that both return classes made up of a number of methods and objects. The main difference is that while the Abstract Factory returns a family of related classes, the Builder constructs a complex object step by step depending on the data presented to it.

The Builder Pattern

:: Pizza Builder

```
/** "Product" */
class Pizza {
  private String dough = "";
  private String sauce = "";
  private String topping = "";
  public void setDough(String dough) {
    this.dough = dough; }
  public void setSauce(String sauce) {
    this.sauce = sauce; }
  public void setTopping(String topping) {
    this.topping = topping; }
}
```

The Builder Pattern

:: Pizza Builder

```
/** "Abstract Builder" */
 abstract class PizzaBuilder {
    protected Pizza pizza;
    public Pizza getPizza() {
      return pizza; }
    public void createNewPizzaProduct() {
      pizza = new Pizza(); }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
 }
```

The Builder Pattern

:: Pizza Builder

```
/** "ConcreteBuilder" */
 class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough()   {
      pizza.setDough("cross"); }
    public void buildSauce()   {
      pizza.setSauce("mild"); }
    public void buildTopping() {
      pizza.setTopping("ham+pineapple"); }
 }
 /** "ConcreteBuilder" */
 class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough()   {
      pizza.setDough("pan baked"); }
    public void buildSauce()   {
      pizza.setSauce("hot"); }
    public void buildTopping() {
      pizza.setTopping("pepperoni+salami"); }
 }
```

The Builder Pattern

:: Pizza Builder

```
/** "Director" */
class Waiter {
   private PizzaBuilder pizzaBuilder;
   public void setPizzaBuilder(PizzaBuilder pb) {
     pizzaBuilder = pb; }
   public Pizza getPizza() {
     return pizzaBuilder.getPizza(); }
   public void constructPizza() {
      pizzaBuilder.createNewPizzaProduct();
      pizzaBuilder.buildDough();
      pizzaBuilder.buildSauce();
      pizzaBuilder.buildTopping();
   }
}
```

The Builder Pattern

:: Pizza Builder

```
 /** A customer ordering a pizza. */
class BuilderExample {
   public static void main(String[] args) {
        Waiter waiter = new Waiter();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

        waiter.setPizzaBuilder( hawaiianPizzaBuilder );
        waiter.constructPizza();

        Pizza pizza = waiter.getPizza();
   }
}
```

The Prototype Pattern

:: Definition & Applicability

- The **Prototype** pattern specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- A **Protoype** pattern is used when creating an instance of a class is very time-consuming or complex in some way. Then, rather than creating more instances, you make copies of the original instance and modify them as appropriate.
- Prototypes can also be used whenever you need classes that differ only in the type of processing they offer, for example in parsing of strings representing numbers in different radixes. In this sense, the prototype is nearly the same as the **Examplar** pattern described by Coplien [4].

Example:

- Let's consider the case of an extensive database where you need to make a number of queries to construct an answer. Once you have this answer as a table or ResultSet, you might like to manipulate it to produce other answers without issuing additional queries.

The Prototype Pattern

:: Cloning in Java - I

You can make a copy of any Java object using the **clone** method.

```
Jobj j1 = (Jobj)j0.clone();
```

The clone method always returns an object of type Object. You must cast it to the actual type of the object you are cloning. There are three other significant restrictions on the clone method:

- It is a protected method and can only be called from within the same class or the module that contains that class.
- You can only clone objects which are declared to implement the *Cloneable* interface.
- Objects that cannot be cloned throw the *CloneNotSupported* Exception.

The Prototype Pattern

:: Cloning in Java - II

This suggests packaging the actual clone method inside the class where it can access the real clone method:

```
public class SwimData implements Cloneable {
        public Object clone()
        {
                try{
                        return super.clone();
                }
                catch(Exception e) {
                        System.out.println(e.getMessage());
                        return null;
                }
        }
}
```

The Prototype Pattern

:: Cloning in Java - III

- This implementation has the advantage of encapsulating the try-catch block inside the public clone method.
- Note that if you declare this public method to have the same name "clone," it must be of type Object, since the internal protected method has that signature. We could, however, change the name and do the typecasting within the method instead of forcing it onto the user:

```
public SwimData cloneMe() {
        try{
                return (SwimData)super.clone();
        }
        catch(Exception e) {
                System.out.println(e.getMessage());
                return null;
        }
}
```

The Prototype Pattern

:: Using the Prototype - I

Let's write a simple program that reads data from a database and then clones the resulting object. In our example program, *SwimInfo*, we just read these data from a file, but the original data were derived from a large database as we discussed above.

We create a class called *Swimmer* that holds one name, club name, sex and time:

```
class Swimmer {
        String name;
        int age;
        String club;
        float time;
        boolean female;
        public String getName () {return name};
        public int getAge () {return age};
        public float getTime () {return time};
}
```

The Prototype Pattern

:: Using the Prototype - II

We create a class called *SwimData* that maintains a vector of the *Swimmers* we read in from the database.

```
public class SwimData implements Cloneable {
        Vector<Swimmer> swimmers;
        public Swimmer getSwimmer(int i) {
               return swimmers.get(i);};
        public SwimData(String filename) {
               String s = "";
               swimmers = new Vector();
               InputFile f = new InputFile(filename); //open file
           s= f.readLine(); //read in and parse each line
               while(s != null) {
                  swimmers.addElement(new Swimmer(s));
                  s= f.readLine();
               }
               f.close();
}
```

The Prototype Pattern

:: Using the Prototype - III

We clone this class and sort the data differently in the new class. Again, we clone the data because creating a new class instance would be much slower, and we want to keep the data in both forms.

In the original class, the records are sorted by name, while in the cloned class, they are sorted by time.

```
SwimData sdata = new SwimData();
sdata.sortByName(); //sort by name
…
sxdata = (SwimData)sdata.clone();
sxdata.sortByTime(); //re-sort by time

for(int i=0; i< sxdata.size(); i++)
//display sorted values from clone
{
        Swimmer sw = sxdata.getSwimmer(i);
        System.out.println(sw.getName()+" "+sw.getTime());
}
```

The Prototype Pattern

:: Consequences of the Prototype Pattern - I

Using the Prototype pattern:

- You can add and remove classes at run time by cloning them as needed.
- You can revise the internal data representation of a class at run time based on program conditions.
- You can also specify new objects at run time without creating a proliferation of classes and inheritance structures.

The Prototype Pattern

:: Consequences of the Prototype Pattern - II

Difficulties:

- One difficulty in implementing the Prototype pattern in Java is that if the classes already exist, you may not be able to change them to add the required clone or ***deep Clone*** methods. The *deep Clone* method can be particularly difficult if all of the class objects contained in a class cannot be declared to implement the *Serializable* interface.
- Classes that have circular references to other classes cannot really be cloned.
- The idea of having prototype classes to copy implies that you have sufficient access to the data or methods in these classes to change them after cloning. This may require adding data access methods to these prototype classes so that you can modify the data once you have cloned the class.

The Singleton Pattern

:: Definition & Applicability - I

Sometimes it is appropriate to have exactly one instance of a class:

- window managers,
- print spoolers,
- filesystems.

Typically, those types of objects known as singletons, are accessed by disparate objects throughout a software system, and therefore require a global point of access.

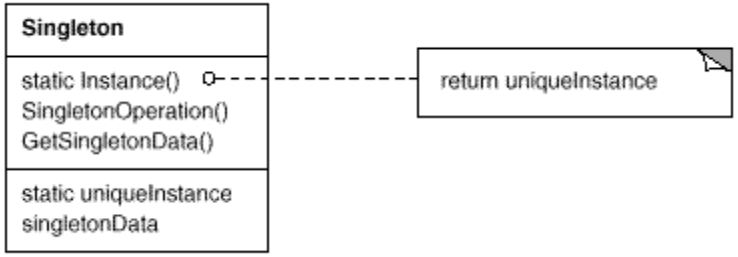
The Singleton pattern addresses all the concerns above. With the Singleton design pattern you can:

- Ensure that only one instance of a class is created.
- Provide a global point of access to the object.
- Allow multiple instances in the future without affecting a singleton class' clients.

The Singleton Pattern

:: Definition & Applicability - II

- The Singleton pattern ensures a class has only one instance, and provides a global point of access to it.
- The class itself is 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.
- Singletons maintain a static reference to the **sole singleton instance** and return a reference to that instance from a static *instance()* method.



The Singleton Pattern

:: The Classic Singleton - I

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

      protected ClassicSingleton() {
         // exists only to defeat instantiation.
      }
      public static ClassicSingleton getInstance() {
            if(instance == null) {
              instance = new ClassicSingleton();
            }
            return instance;
      }
}
```

The *ClassicSingleton* class maintains a static reference to the lone singleton instance and returns that reference from the static getInstance() method.

The Singleton Pattern

:: The Classic Singleton - II

- The ClassicSingleton class employs a technique known as lazy instantiation to create the singleton; as a result, the singleton instance is not created until the getInstance() method is called for the first time. This technique ensures that singleton instances are created only when needed.
- The ClassicSingleton class implements a protected constructor so clients cannot instantiate ClassicSingleton instances; however, the following code is perfectly legal:

```
public class SingletonInstantiator {
    public SingletonInstantiator() {
      ClassicSingleton instance =
  ClassicSingleton.getInstance();

      ClassicSingleton anotherInstance = new
  ClassicSingleton();
          ...
    }
}
```

The Singleton Pattern

:: Problems - I

How can a class that does not extend *ClassicSingleton* create a *ClassicSingleton* instance if the *ClassicSingleton* constructor is protected?

- Protected constructors can be called by subclasses and *by other classes in the same package*. Hence, because *ClassicSingleton* and *SingletonInstantiator* are in the same package (the default package), *SingletonInstantiator()* methods can create *ClassicSingleton* instances.

Solutions:

1. We can make the ***ClassicSingleton* constructor private** so that only ClassicSingleton's methods call it; however, that means *ClassicSingleton* cannot be subclassed. Also, it's a good idea to declare the singleton class **final**, which makes that intention explicit and allows the compiler to apply performance optimizations.
2. We can put your singleton class in an explicit package, so classes in other packages (including the default package) cannot instantiate singleton instances.

The Singleton Pattern

:: Problems - II

The ClassicSingleton class is not thread-safe.

If two threads – we will call them Thread 1 and Thread 2, call *ClassicSingleton.getInstance()* at the same time, two *ClassicSingleton* instances can be created if Thread 1 is preempted just after it enters the if block and control is subsequently given to Thread 2.

Solution: **Synchronization**

```
public class ClassicSingleton {
   private static ClassicSingleton instance = null;
   private static Object syncObject; // to synchronize a block
   protected ClassicSingleton() {
      /*exists only to defeat instantiation*/ };
   public static ClassicSingleton getInstance() {
      synchronized(syncObject) {
      if (instance == null) instance = new ClassicSingleton();}
      return instance;}
}
```

The Singleton Pattern

:: Consequences of the Singleton Pattern

- It can be difficult to subclass a Singleton, since this can only work if the base Singleton class has not yet been instantiated.
- We can easily change a Singleton to allow a small number of instances where this is allowable and meaningful.
- We can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
- The Singleton pattern permits refinement of operations and representation. The Singleton class may be subclassed, and it is easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.

- Structural patterns describe how classes and objects can be combined to form larger structures.
- The difference between *class* patterns and *object* patterns is that class patterns describe how inheritance can be used to provide more useful program interfaces.
- Object patterns, on the other hand, describe how objects can be composed into larger structures using object composition, or the inclusion of objects within other objects.
- The Structural patterns are:
  - Adapter
  - Composite
  - Proxy
  - Flyweight
  - Façade
  - Bridge
  - Decorator

- The Adapter pattern can be used to make one class interface match another to make programming easier.

- The Composite pattern is a composition of objects, each of which may be either simple or itself a composite object.

- The Proxy pattern is frequently a simple object that takes the place of a more complex object that may be invoked later, for example when the program runs in a network environment.

- The Flyweight pattern is a pattern for sharing objects, where each instance does not contain its own state, but stores it externally. This allows efficient sharing of objects to save space, when there are many instances, but only a few different types.

- The Façade pattern is used to make a single class represent an entire subsystem.

- The Bridge pattern separates an object's interface from its implementation, so you can vary them separately.

- The Decorator pattern, which can be used to add responsibilities to objects dynamically.

The Adapter Pattern

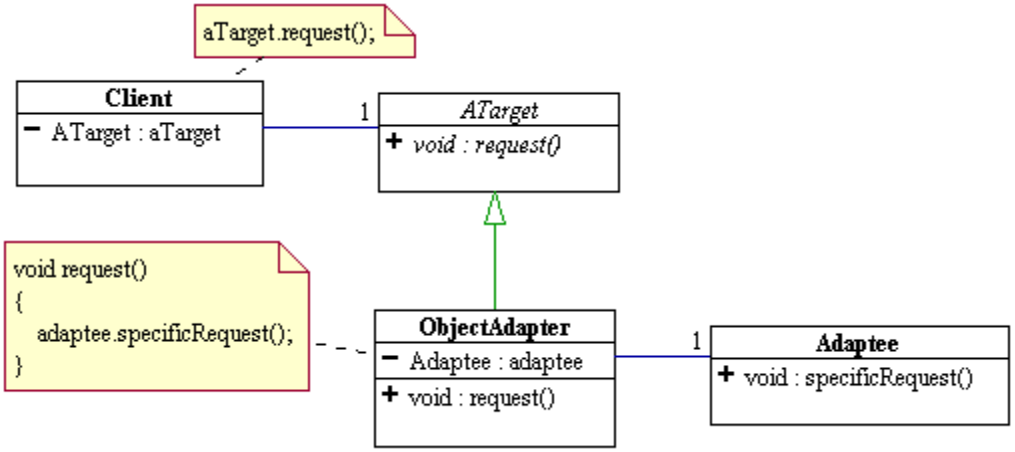
:: Definition & Applicability

- Adapters are used to enable objects with different interfaces to communicate with each other.
- The Adapter pattern is used to convert the programming interface of one class into that of another. We use adapters whenever we want unrelated classes to work together in a single program.
- Adapters come in two flavors, **object adapters and class adapters**.
- The concept of an adapter is thus pretty simple; we write a class that has the desired interface and then make it communicate with the class that has a different interface.
- Adapters in Java can be implemented in two ways: by inheritance, and by object composition.

The Adapter Pattern

:: Object Adapters

- Object adapters use a compositional technique to adapt one interface to another.
- The adapter inherits the target interface that the client expects to see, while it holds an instance of the adaptee.
- When the client calls the *request()* method on its target object (the adapter), the request is translated into the corresponding specific request on the adaptee.
- Object adapters enable the client and the adaptee to be completely decoupled from each other. Only the adapter knows about both of them.



The Adapter Pattern

:: Example - I

```
/**
* The SquarePeg class.
* This is the Target class.
*/
public class SquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);}
}

/**
* The RoundPeg class.
* This is the Adaptee class.
*/
public class RoundPeg {
  public void insertIntoHole(String msg) {
   System.out.println("RoundPeg insertIntoHole(): " + msg);}
}
```

If a client only understands the **SquarePeg** interface for inserting pegs using the **insert()** method, how can it insert round pegs, which are pegs, but that are inserted differently, using the **insertIntoHole()** method?

The Adapter Pattern

:: Example - II

Solution:

Design a **RoundToSquarePeg** adapter that enables to **insertIntoHole()** a **RoundPeg** object connected to the adapter to be inserted as a **SquarePeg**, using **insert()**.

```
/**
* The RoundToSquarePegAdapter class.
* This is the Adapter class.
* It adapts a RoundPeg to a SquarePeg.
* Its interface is that of a SquarePeg.
*/
public class RoundToSquarePegAdapter extends SquarePeg {
  private RoundPeg roundPeg;
  public RoundToSquarePegAdapter(RoundPeg peg) {
    //the roundPeg is plugged into the adapter
    this.roundPeg = peg;}
  public void insert(String str) {
    //the roundPeg can now be inserted in the same manner as a squarePeg!
    roundPeg.insertIntoHole(str);}
}
```

The Adapter Pattern

:: Example - III

Example:

```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[]) {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");

    // Now we'd like to do an insert using the round peg.
    // But this client only understands the insert()
    // method of pegs, not a insertIntoHole() method.
    // The solution: create an adapter that adapts
    // a square peg to a round peg!

    RoundToSquarePegAdapter adapter = new RoundToSquarePegAdapter(roundPeg);
    adapter.insert("Inserting round peg...");}
}
```

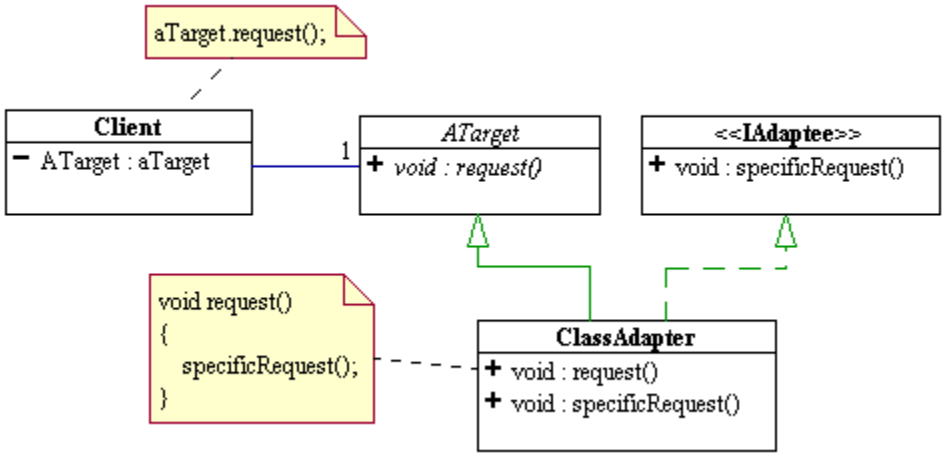
Execution trace:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
```

The Adapter Pattern

:: Class Adapters

- Class adapters use multiple inheritance to achieve their goals.
- As in the object adapter, the class adapter inherits the interface of the client's target. However, it also inherits the interface of the adaptee as well.
- Since Java does not support true multiple inheritance, this means that **one of the interfaces must be inherited from a Java Interface type.**
- Both of the target or adaptee interfaces could be Java Interfaces.
- The request to the target is simply rerouted to the specific request that was inherited from the adaptee interface.



The Adapter Pattern

:: Example I

Here are the interfaces for round and square pegs:

```
/**
*The IRoundPeg interface.
*/
public interface IRoundPeg {
public void insertIntoHole(String msg);
}

/**
*The ISquarePeg interface.
*/
public interface ISquarePeg {
public void insert(String str);
}
```

The Adapter Pattern

:: Example II

Here are the new **RoundPeg** and **SquarePeg** classes. These are essentially the same as before except they now implement the appropriate interface.

```
// The RoundPeg class.
public class RoundPeg implements IRoundPeg {
  public void insertIntoHole(String msg) {
    System.out.println("RoundPeg insertIntoHole(): " + msg);}
}

// The SquarePeg class.
public class SquarePeg implements ISquarePeg {
  public void insert(String str) {
    System.out.println("SquarePeg insert(): " + str);}
}
```

The Adapter Pattern

:: Example III

And here is the new **PegAdapter** class:

```
/**
* The PegAdapter class.
* This is the two-way adapter class.
*/
public class PegAdapter implements ISquarePeg, IRoundPeg {
  private RoundPeg roundPeg;
  private SquarePeg squarePeg;

  public PegAdapter(RoundPeg peg) {
    this.roundPeg = peg;}
  public PegAdapter(SquarePeg peg) {
    this.squarePeg = peg;}

  public void insert(String str) {
    roundPeg.insertIntoHole(str);}
  public void insertIntoHole(String msg){
    squarePeg.insert(msg);}
}
```

The Adapter Pattern

:: Example IV

A client that uses the two-way adapter:

```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[]) {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");

    // Create a two-way adapter and do an insert with it.
    ISquarePeg roundToSquare = new PegAdapter(roundPeg);
    roundToSquare.insert("Inserting round peg...");

    // Do an insert using the round peg.
    roundPeg.insertIntoHole("Inserting round peg...");

    // Create a two-way adapter and do an insert with it.
    IRoundPeg squareToRound = new PegAdapter(squarePeg);
    squareToRound.insertIntoHole("Inserting square peg...");}
}
```

The Adapter Pattern

:: Example V

Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
RoundPeg insertIntoHole(): Inserting round peg...
SquarePeg insert(): Inserting square peg...
```

The Adapter Pattern

:: Consequences of the Adapter Pattern

Class and object adapters have different trade-offs.

A class adapter:

- **adapts Adaptee to Target by committing to a concrete Adapter class**;
- **lets Adapter override some of Adaptee's behavior**, since Adapter is a subclass of Adaptee;
- **introduces only one object, and no additional indirection is needed to get to the adaptee.**

An object adapter

- **lets a single Adapter work with many Adaptees** - that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- **makes it harder to override Adaptee behavior**. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

The Bridge Pattern

:: Definition & Applicability

The **Bridge** pattern is used to separate the interface of class from its implementation, so that either can be varied separately.

- At first sight, the bridge pattern looks much like the **Adapter** pattern, in that a class is used to convert one kind of interface to another. However, the intent of the Adapter pattern is to make one or more classes' interfaces look the same as that of a particular class.
- The **Bridge** pattern is designed to separate a class' interface from its implementation, so that you can vary or replace the implementation without changing the client code.

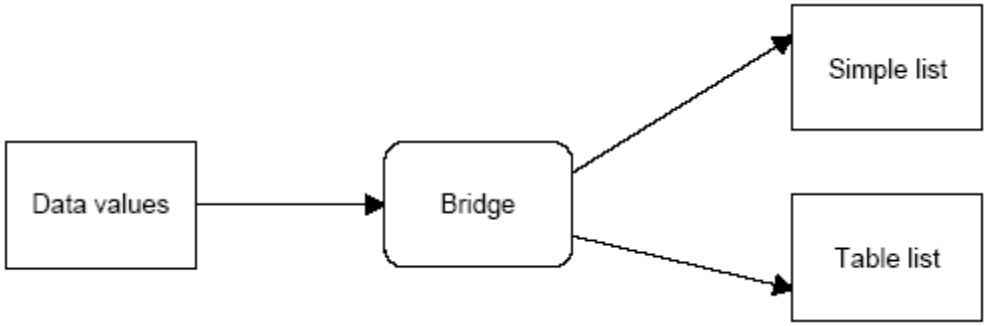
Example:

Suppose that we have a program that displays a list of products in a window. The simplest interface for that display is a simple *JList* box. But, once a significant number of products have been sold, we may want to display the products in a table along with their sales figures.

The Bridge Pattern

:: Building the Bridge Pattern - I

Following the example from the previous slide, suppose that we need to make some changes in the way these lists display data. So, rather than deriving new classes whenever we need to change these displays further, let's build a single ***bridge*** that does this work for us.



The Bridge Pattern

:: Building the Bridge Pattern - II

We want the bridge class to return an appropriate visual component, so we will extend the Java *JSrollPane* class:

```
public class ListBridge extends JScrollPane
{…}
```

When we design a bridge class, we have to decide how the bridge will determine which of the several classes it will instantiate. This could be decided based on the values or quantity of data to be displayed, or based on some simple constants. Here we define the two constants inside the *ListBridge* class:

```
static public final int TABLE = 1, LIST = 2;
```

The Bridge Pattern

:: Building the Bridge Pattern - III

The constructor of the *ListBridge* class:

```
public ListBridge(Vector v, int table_type)
{
     Vector sort = sortVector(v); //sort the vector
     if (table_type == LIST)
       getViewport().add(makeList(sort)); //make table

     if (table_type == TABLE)
       getViewport().add(makeTable(sort)); //make list
}
```

The Bridge Pattern

:: Building the Bridge Pattern - IV

We can use the *JTable* and *JList* classes directly without modification and thus can put any adapting interface computations in the data models that construct the data for the list and table.

```
private JList makeList(Vector v) {
        return new JList(new ListModel(v));
}
//---------------------------------
private JTable makeTable(Vector v) {
        return new JTable(new TableModel(v));
}
```

Where *ListModel* and *TableModel* are Java API classes.

The Bridge Pattern

:: The Bridge Pattern Class

```
public class ListBridge extends JScrollPane{
        static public final int TABLE = 1, LIST = 2;
        private JList makeList(Vector v) {
                return new JList(new ListModel(v));
        }
        private JTable makeTable(Vector v) {
                return new JTable(new TableModel(v));
        }
        public ListBridge(Vector v, int table_type) { //constructor
                Vector sort = sortVector(v); //sort the vector
                if (table_type == LIST)
                        getViewport().add(makeList(sort)); //make table
                if (table_type == TABLE)
                        getViewport().add(makeTable(sort)); //make list
        }
}
```

The Bridge Pattern

:: Consequences of the Bridge Pattern

- The Bridge pattern is intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use.

- This can prevent you from recompiling a complicated set of user interface modules, and only require that you recompile the bridge itself and the actual end display class.

- You can extend the implementation class and the bridge class separately, and usually without much interaction with each other.

The Composite Pattern

:: Definition & Applicability

The **Composite** Design pattern allows a client object to treat both single components and collections of components identically.

Composite patterns are often used to represent **recursive** data structures. The recursive nature of the Composite structure naturally gives way to recursive code to process that structure.

Use the Composite pattern when:

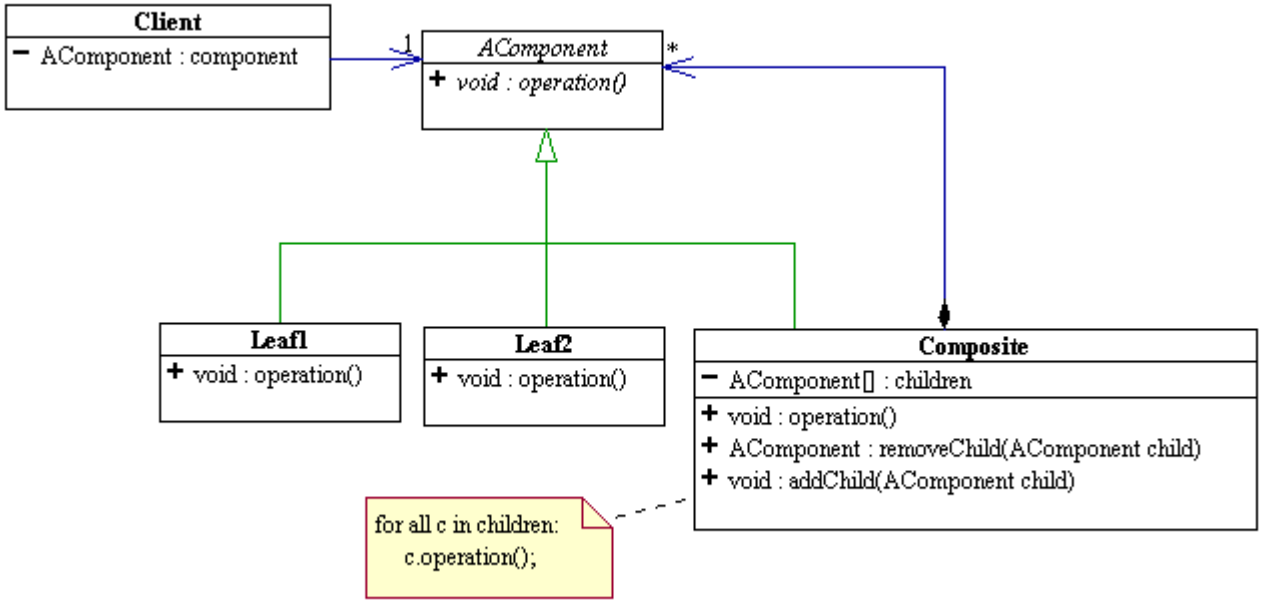
- You want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

The Composite Pattern

:: Example

In the UML class diagram below:

- The Client uses an abstract component, *AComponent*, for some abstract task, *operation*().
- At run-time, the Client holds a reference to a concrete component such as Leaf1 or Leaf2.
- When the operation task is requested by the Client, the specific concrete behavior with the particular concrete component will be performed.



The Composite Pattern

:: Consequences of the Composite Pattern

- The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program.
- Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.
- The Composite pattern also makes it easy for you to add new kinds of components to your collection, as long as they support a similar programming interface.
- The composite is essentially a singly-linked tree, in which any of the objects may themselves be additional composites.

The Observer Pattern

:: Definition & Applicability - I

Motivation

The cases when certain objects need to be informed about the changes occured in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.

Intent

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

This pattern is a cornerstone of the Model-View-Controller architectural design, where the Model implements the mechanics of the program, and the Views are implemented as Observers that are as much uncoupled as possible to the Model components.

The Observer Pattern

:: Definition & Applicability - III

The participants classes in the Observer pattern are:

Observable - interface or abstract class defining the operations for attaching and de-attaching observers to the client. In the GOF book this class/interface is known as **Subject**.

ConcreteObservable - concrete Observable class. It maintain the state of the observed object and when a change in its state occurs it notifies the attached **Observers**.

Observer - interface or abstract class defining the operations to be used to notify the Observer object.

ConcreteObserverA, ConcreteObserverB - concrete **Observer** implementations.

The Observer Pattern

:: Definition & Applicability - IV

Behavior

- The client class instantiates the ConcreteObservable object.
- Then it instantiate and attaches the concrete observers to it using the methods defined in the Observable interface.
- Each time the state of the subject it's changing it notifies all the attached Observers using the methods defined in the Observer interface.
- When a new Observer is added to the application, all we need to do is to instantiate it in the client class and to add attach it to the Observable object.
- The classes already created will remain unchanged.

The Observer Pattern

Java Observable class API

```
Observable()
        Construct an Observable with zero Observers.
void addObserver(Observer o)
        Adds an observer to the set of observers for this object, provided
        that it is not the same as some observer already in the set.
protected void clearChanged()
        Indicates that this object has no longer changed, or that it has
        already notified all of its observers of its most recent change, so
        that the hasChanged method will now return false.
int countObservers()
        Returns the number of observers of this Observable object.
void deleteObserver(Observer o)
        Deletes an observer from the set of observers of this object.
void deleteObservers()
        Clears the observer list so that this object no longer has any
        observers.
```

The Observer Pattern

Java Observable class API

```
boolean hasChanged()
        Tests if this object has changed.
void notifyObservers()
        If this object has changed, as indicated by the hasChanged method,
        then notify all of its observers and then call the clearChanged
        method to indicate that this object has no longer changed.
void notifyObservers(Object arg)
        If this object has changed, as indicated by the hasChanged method,
        then notify all of its observers and then call the clearChanged
        method to indicate that this object has no longer changed.
protected void setChanged()
        Marks this Observable object as having been changed; the hasChanged
        method will now return true.
```

The Observer Pattern

:: Example - I

```
// A Sub-class of Observable: a Clock Timer
//

import java.util.Observable;

class ClockTimerModel extends Observable {
  public:
    ClockTimer();
    int GetHour(){return hour};
    int GetMinute(){return minute};
    int GetSecond(){return second};
    void tick(){
     // update internal time-keeping state
     // ...
     // The Observable object notifies all its registered observers
     setChanged();
     notifyObservers();};
  private:
    int hour;
    int minute;
    int second;
};
```

In green are the changes to be applied to the class to be made an observable class.

The Observer Pattern

Java Observer interface API

```
public void update(Observable o, Object arg)
        This method is called whenever the observed object is changed. An
        application calls an Observable object's notifyObservers method to
        have all the object's observers notified of the change.
    Parameters:
        o   - the observable object.
        arg - an argument passed to the notifyObservers method.
```

The Observer Pattern

:: Example - II

```
// A specific Observer to observe ClockTimerModel: DigitalClockView
//

import java.util.Observer;

class DigitalClockView implements Observer {

    public void update(Observable obs, Object x) {
     //redraw my clock's reading
     draw();};

    void draw(){
     int hour   = obs.GetHour();
     int minute = obs.GetMinute();
     int second = obs.GetSecond();
     // draw operation};
};
```

The Observer Pattern

:: Example - III

```
public class ObserverDemo extends Object {
  DigitalClockView clockView;
  ClockTimerModel clockModel;

  public ObservDemo() {
    clockView = new DigitalClockView();
    clockModel = new ClockTimerModel();
    clockModel.addObserver(clockView);
  }

  public static void main(String[] av) {
    ObserverDemo me = new ObserverDemo();
    me.demo();
  }

  public void demo() {
    clockModel.Tick();
  }
```

Java Design Patterns

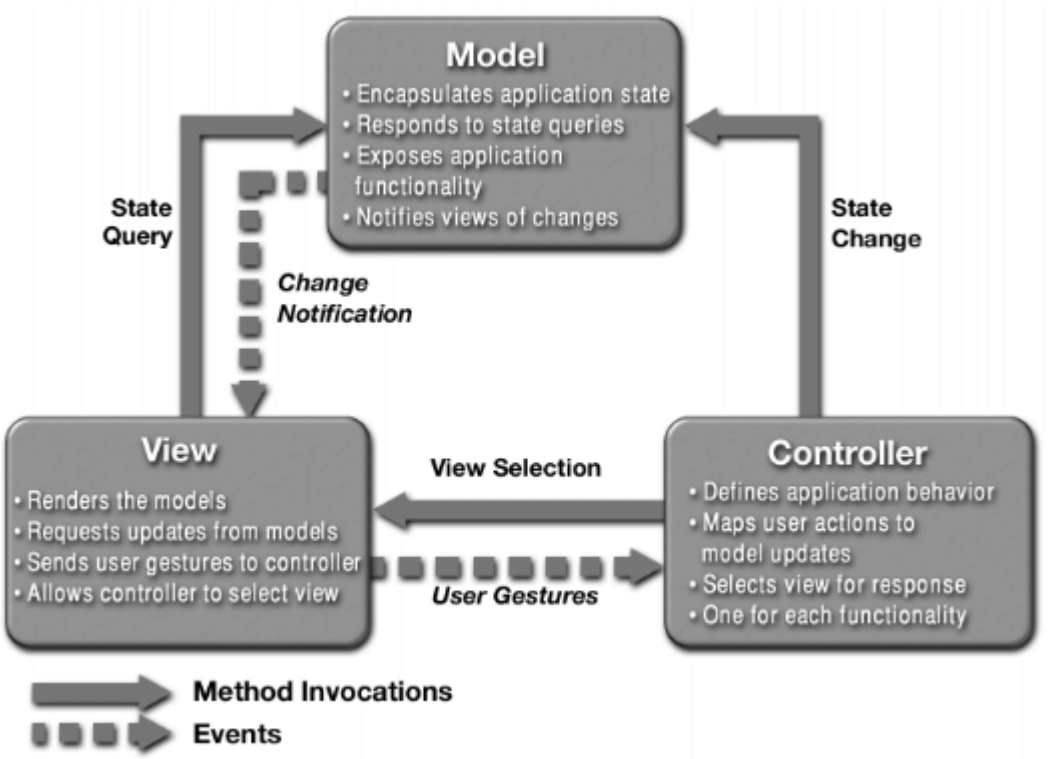
:: Java BluePrints Patterns Catalog

- **Business Delegate** - Reduce coupling between Web and Enterprise JavaBeansTM tiers
- **Composite Entity** - Model a network of related business entities
- **Composite View** - Separately manage layout and content of multiple composed views
- **Data Access Object (DAO)** - Abstract and encapsulate data access mechanisms
- **Fast Lane Reader** - Improve read performance of tabular data
- **Front Controller** - Centralize application request processing
- **Intercepting Filter** - Pre- and post-process application requests
- **Model-View-Controller** - Decouple data representation, application behavior, and presentation
- **Service Locator** - Simplify client access to enterprise business services
- **Session Facade** - Coordinate operations between multiple business objects in a workflow
- **Transfer Object** - Transfer business data between tiers
- **Value List Handler** - Efficiently iterate a virtual list
- **View Helper** - Simplify access to model state and data access logic

The MVC Architectural Pattern

:: Introduction

- MVC was first introduced by Trygve Reenskaug at the Xerox Palo Alto Research Center in 1979.
- Part of the basic of the Smalltalk programming environment.
- Widely used for many object-oriented designs involving user interaction.
- A three-tier architectural model:



The MVC Architectural Pattern

:: Model

- manages the **behavior** and **data** of the application domain,
- responds to requests for information about its state (usually from the view),
- responds to instructions to change state (usually from the controller).
- In event-driven systems, the model notifies observers (usually views) when the information changes so that they can react. (see observer pattern)
- In enterprise software, a model often serves as a software approximation of a real-world process.
- In a game, the model is represented by the classes defining the game entities, which are embedding their own state and actions.

The MVC Architectural Pattern

:: View

- Renders the model into a form suitable for interaction, typically a user interface element.
- Multiple views can exist for a single model for different purposes.
- The view renders the contents of a portion of the model's data.
- If the model data changes, the view must update its presentation as needed. This can be achieved by using:
  - a *push model*, in which the view registers itself with the model for change notifications (see the observer pattern)
  - a *pull model*, in which the view is responsible for calling the model when it needs to retrieve the most current data.

The MVC Architectural Pattern

:: Controller

- Receives user input and initiates a response by making calls on appropriate model objects.
- Accepts input from the user and instructs the model to perform actions based on that input.
- The controller translates the user's interactions with the view it is associated with, into actions that the model will perform.
- A controller may also spawn new views upon user demand.

The MVC Architectural Pattern

:: Interactions between Model, View and Controller

Upon creation of a Model-View-Controller triad:

1. The view registers as an observer on the model. Any changes to the underlying data of the model immediately result in a broadcast change notification, which all associated views receives (in the *push back* model). Note that the model is not aware of the view or the controller -- it simply broadcasts change notifications to all interested observers.
2. The controller is bound to the view and can react to any user interaction provided by this view. This means that any user actions that are performed on the view will invoke a method in the controller class.
3. The controller is given a reference to the underlying model.

The MVC Architectural Pattern

:: Interactions between Model, View and Controller

Once a user interacts with the view, the following actions occur:

1. The view recognizes that a GUI action -- for example, pushing a button or dragging a scroll bar -- has occurred, e.g using a listener method that is registered to be called when such an action occurs. The mechanism varies depending on the technology/library used.
2. In the listener method, the view calls the appropriate method on the controller.
3. The controller translates this signal into an appropriate action in the model, which will in turn possibly be updated in a way appropriate to the user's action.
4. If the model has been altered, it notifies interested observers, such as the view, of the change. In some architectures, the controller may also be responsible for updating the view. Again, technical details may vary according to technology/library used.

Resources

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Adisson-Wesley, 1995.

[3] James W. Cooper, *The Design Patterns – Java Companion Elements of Reusable Object-Oriented Software*, Adisson-Wesley, 1998.

[4] James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, Reading, MA., 1992.

[5] David Geary, *Simply Singleton*, Java Design Patterns at JavaWorld, April 2003, http://www.javaworld.com/columns/jw-java-design-patterns-index.shtml

[6] Design Patterns, http://www.exciton.cs.rice.edu/JavaResources/DesignPatterns/

[7] Robert Eckstein, *Java SE Application Design With MVC*, Oracle Technology Network, March 2007. http://www.oracle.com/technetwork/articles/javase/mvc-136693.html