Pattern

Different Definitions…

- a fully realized form, original, or model accepted or proposed for imitation…[dictionary]
- 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 [Alexander]
- the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts [Riehle]
- both a thing and the instructions for making the thing [Coplien]
- a literary format for capturing the wisdom and experience of expert designers, and communicating it to novices

A design pattern in architecture and computer science is a formal way of documenting a solution to a design problem in a particular field of expertise. The idea was introduced by the architect Christopher Alexander in the field of architecture and has been adapted for various other disciplines, including computer science. An organized collection of design patterns that relate to a particular field is called a pattern language. --- Christopher Alexander

A pattern is a proven solution to a problem in a context. Christopher Alexander says each pattern is a three-part rule which expresses a relation between a certain context, a problem, and a solution. Design patterns represent solutions to problems that arise when developing software within a particular context. i.e Patterns = problems solution pairs in a context. Patterns, then, represent expert solutions to recurring problems in a context and thus have been captured at many levels of abstraction and in numerous domains. Numerous categories are: Design, Architectural , Analysis , Creational ,Structural , Behavioral . Patterns are generally considered to be an indication of good design and development practices.

The purpose of patterns then can be generalized as providing the following:

- Giving a vocabulary to a problem, context, solution, consequences set
- Cataloging designs that have proved successful in past systems and formalizing their elements, relationships, interactions, etc.
- Provide enough information about trade-offs and consequences to allow an intelligent design decision to be made about applying a given solution

Four Essential elements of Patterns:

- Pattern Name
  - Having a concise, meaningful name for a pattern improves communication among developers
- Problem
  - What is the problem and context where we would use this pattern?
  - What are the conditions that must be met before this pattern should be used?
- Solution
  - A description of the elements that make up the design pattern
  - Emphasizes their relationships, responsibilities and collaborations
  - Not a concrete design or implementation; rather an abstract description
- Consequences
  - The pros and cons of using the pattern
  - Includes impacts on reusability, portability, extensibility

Properties:

- Patterns do provide common vocabulary

- Patterns do provide “shorthand” for effectively communicating complex principles
- Patterns do help document software architecture
- Patterns do capture essential parts of a design in compact form
- Patterns do show more than one solution
- Patterns do describe software abstractions
- Patterns do not provide an exact solution
- Patterns do not solve all design problems

How design patterns solve problem?

- OOP is made up of objects, where an object packages both data and operations that may be performed on that data
- Ideally, the only way to change the data of an object is via a request to that object (normally a method call) – we call this encapsulation; it’s representation is invisible to the outside world
- The hard part of object-oriented design is decomposing a system into objects. Many factors come into play: granularity, encapsulation, dependency, flexibility, performance, reusability, and each affects the nature of the decomposition, often in conflicting ways

A pattern must explain why a particular situation causes problems, and why the proposed solution is considered a good one. Christopher Alexander describes common design problems as arising from "conflicting forces" -- such as the conflict between wanting a room to be sunny and wanting it not to overheat on summer afternoons. A pattern would not tell the designer how many windows to put in the room; instead, it would propose a set of values to guide the designer toward a decision that is best for their particular application. Alexander, for example, suggests that enough windows should be included to direct light all around the room. He considers this a good solution because he believes it increases the enjoyment of the room by its occupants. Other authors might come to different conclusions, if they place higher value on heating costs, or material costs. These values, used by

the pattern's author to determine which solution is "best", must also be documented within the pattern.

A pattern must also explain when it is applicable. Since two houses may be very different from one another, a design pattern for houses must be broad enough to apply to both of them, but not so vague that it doesn't help the designer make decisions. The range of situations in which a pattern can be used is called its context. Some examples might be "all houses", "all two-story houses", or "all places where people spend time". The context must be documented within the pattern.

For instance, in Christopher Alexander's work, bus stops and waiting rooms in a surgery center are both part of the context for the pattern "A PLACE TO WAIT".

A Java Factory Pattern example

In this example, we'll create a simple "Car factory" that can return a variety of *Car* types, where the "Car" that is returned matches the criteria we specify. For instance, we might tell our factory that we want a small Car, or a large Car, and the Car factory will give us a Car of the type we asked for.

The Car interface

First, we'll create a *Car* interface. Any Car that our factory returns must implement this Java interface, so for the purposes of our example, we'll keep this interface very simple. We'll just specify that any class that calls itself a *Car* must implement a *cost* method that looks like this:

```
interface Car
{
  public void cost ();
}
```

The concrete Car classes

Next, we'll define a few concrete classes that implement our *Car* interface. Keeping with our simple interface, each class implements the *cost* method, but implements it in a slightly

different way that befits each Car type:

```
class Suzuki implements Car
{
  public void cost()
  {
    System.out.println("The Suzuki: $9999");
  }
}

class  Maruti implements Car
{
  public void cost()
  {
    System.out.println("The  Maruti :$8888");
  }
}
```

As you can see from the code, each of these concrete classes (*Suzuki and Maruti*) implements our *Car* interface. This is a key point, and an important part of the Factory Pattern:

> You define a base class type (or in this case an interface), and then have any number of subclasses which implement the contract defined by the base class.

And as you're about to see, the signature of the factory method shows that it will be returning a class which implements our base class, in this case, our *Car* interface.

The Java Factory class

Next we'll define our Java Factory class, which in this case is a CarFactory class. As you can see from the code below, the CarFactory class has a static *getCar* method that returns a "*Car*" depending on the criteria that has been supplied.

```
class CarFactory
{
  public static Car getCar(String criteria)
  {
    if ( criteria.equals("small") )
      return new Suzuki();
    else if ( criteria.equals("big") )
```

```
      return new  Maruti();

    return null;
  }
}
```

As I mentioned, the signature of our Java factory method states that we will be returning a class of type*Car*:

```
public static Car getCar(String criteria)
```

The factory doesn't say it's returning a *Suzuki or Maruti*-- it just says it's returning ***something*** that implements the *Car*interface.

Also, it's very important to note that in this simple example I'm only accepting strings like "small" and "big" as our "criteria". In a more complicated (real world) example, you'll want to tighten down this code much more.

The Java Factory pattern example driver program

Now that we've created our Car factory, our *Car* interface, and all of our Car types, we'll create a "driver" program named *JavaFactoryPatternExample* to test our Car factory. This driver class demonstrates how to get different types of Cars from the factory:

```
/**
 * A "driver" program to demonstrate our "Car factory".
 * @author alvin alexander, devdaily.com
 */
public class JavaFactoryPatternExample
{
  public static void main(String[] args)
  {
    // create a small Car
    Car Car = CarFactory.getCar("small");
    Car.cost();

    // create a big Car
```

```
    Car = CarFactory.getCar("big");
    Car.cost();

  }
}
```

As you can see from our driver class, we create an instance of each type of Car (small, big).

Discussion

My intent here isn't to get too deep into the theory behind the Factory Pattern, but instead to demonstrate the pattern in Java source code. That being said, there are a couple of quick points I'd like to make about the Factory Pattern:

- A simple factory like this returns an instance of any one of several possible classes that have a common parent class.
- The common parent class can be an abstract class, or an interface, as I've shown here.
- The calling program typically has a way of telling the factory what it wants, and the factory makes the decision which subclass should be returned to the calling program. It then creates an instance of that subclass, and then returns it to the calling program.

Dependency Injection & Inversion of Control:

The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code. Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement dependency injection pattern to move the dependency resolution from compile-time to runtime.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

```
public class TextEditor
{
```

```
    private SpellChecker checker;
    public TextEditor()
    {
        checker = new SpellChecker();
    }
}
```

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

```
public class TextEditor
{
    private ISpellChecker checker;
    public TextEditor(ISpellChecker checker)
    {
        this.checker = checker;
    }
}
```

Now, the client creating the TextEditor class has the control over which SpellChecker implementation to use. We're injecting the TextEditor with the dependency.

Dependency injection is basically providing the objects that an object needs (its dependencies) instead of having it construct them itself. It's a very useful technique for testing, since it allows dependencies to be mocked or stubbed out.

Dependencies can be injected into objects by many means. One can even use specialized dependency injection frameworks to do that, but they certainly aren't required. You don't need those frameworks to have dependency injection. Instantiating and passing objects (dependencies) explicitly to is just as good an injection as injection by a framework.

Instead of having your objects creating a dependency or asking a factory object to make one for them, you pass the needed dependencies in to the constructor or via property setters, and you make it somebody else's problem (an object further up the dependency graph, or a dependency injector that builds the dependency graph). A dependency as I'm using it here is any

other object the current object needs to hold a reference to.

One of the major advantages of dependency injection is that it can make testing lots easier. Suppose you have an object which in its constructor does something like:

```
public SomeClass() {
    myObject = Factory.getObject();
}
```

This can be troublesome when all you want to do is run some unit tests on SomeClass, especially if myObject is something that does complex disk or network access. So now you're looking at mocking myObject but also somehow intercepting the factory call. Hard. Instead, pass the object in as an argument to the constructor. Now you've moved the problem elsewhere, but testing can become lots easier. Just make a dummy myObject and pass that in. The constructor would now look a bit like:

```
public SomeClass (MyClass myObject) {
    this.myObject = myObject;
}
```

Any application is composed of many objects that collaborate with each other to perform some useful stuff. Traditionally each object is responsible for obtaining its own references to the dependent objects (dependencies) it collaborate with. This leads to highly coupled classes and hard-to-test code.

For example, consider a `Car` object. A `Car` depends on Wheels, Engine, Fuel, Battery, etc to run. Traditionally we define the brand of such dependent objects along with the definition of the `Car` object.

```
class Car{
  private Wheel wh= new NepaliRubberWheel();
  private Battery bt= new ExcideBattery();
  //rest
}
```

Here, the `Car` object is responsible for creating the dependent objects.

What if we want to change the type of its dependent object - say `Wheel` - after the initial `NepaliRubberWheel()` punctures? We need to recreate the Car object with its new dependency say `ChineseRubberWheel()`, but only the `Car` manufacturer can do

that.
Then what the `Dependency Injection` do us for ...
When using Dependency Injection, objects are given their dependencies at run time rather than compile time (car manufacturing time). So that we can now change the `Wheel` whenever we want. Here, the `Dependency` (`Wheel`) can be injected into `Car` at run time.

Benefits of DI

- Key benefit is loose coupling between dependent objects.If an object operates on their dependencies by their interface not by implementation then compile time dependency can be swapped out with Dependency Injection.
- Very useful when we have objects (Wheel) whose implementations change often (if the type punctures often we can change it easily) ?
- Very useful for large projects where there is issue of maintainability, simplicity and many others ty...

Disadvantages of Dependency Injection

- If overused, it can lead to maintenance issues because effect of changes are known at runtime.
- Dependency injection hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

Real-world Example:

```
/*
Even if this program compiles and executes without problem, Dependency Injection not handled: Since  the email service instance creation is done in MyApplication class
*/
class EmailService {

    public void sendEmail(String message, String receiver){
```

```
        //logic to send email
        System.out.println("Email sent to "+receiver+ " with
Message="+message);
    }
}

 class MyApplication {

    private EmailService email = new EmailService();

    public void processMessages(String msg, String rec){
        //do some msg validation, manipulation logic etc
        this.email.sendEmail(msg, rec);
    }
}
public class DependencyInjectionUnhandledDemo {
    public static void main(String[] args) {
        MyApplication app = new MyApplication();
        app.processMessages("Hi Bidur", "bidur@abc.com");
    }
}
```

The above code logic has certain limitations due to unhandled dependency injection:

- `MyApplication` class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even more harder.

- If we want to extend our application to provide additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.
- Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

Now let's see how we can apply dependency injection pattern to solve all the problems with above implementation. Dependency Injection pattern requires at least following:

1. Service components should be designed with base class or interface. It's better to prefer interfaces or abstract classes that would define contract for the services.
2. Consumer classes should be written in terms of service interface.
3. Injector classes that will initialize the services and then the consumer classes.

Singleton pattern

Singleton pattern is a design solution where an application wants to have one and only one instance of any class, in all possible scenarios without any exceptional condition.

How the Singleton pattern works?

Following is the source of simple class that follows singleton pattern.

```
public class SimpleSingleton {

   private static SimpleSingleton INSTANCE = new SimpleSingleton();

   //Marking default constructor private to avoid direct instantiation.
```

```
    private SimpleSingleton() {
    }
    //Get instance for class SimpleSingleton
    public static SimpleSingleton getInstance() {
        return INSTANCE;
    }
}
```

In above code snippet, we have declared a static object reference to SimpleSingleton class called INSTANCE and is returned every time getInstance() is called. In this design, the object will be instantiate when the class will be loaded and before the method getInstance() is being called. Demand loading (lazy loading) also called initialization on demand holder idiom is not seen in this implementation.
We can change this code to add lazy init (Instantiate only when first time getInstance() is called) into this. Following is the code snippet for that.

```
public class SimpleSingleton {
    private SimpleSingleton singleInstance = null;
    //Marking default constructor private to avoid direct instantiation.
    private SimpleSingleton() {
    }
    //Get instance for class SimpleSingleton
    public static SimpleSingleton getInstance() {
            if(null == singleInstance) {
```

```
            singleInstance = new SimpleSingleton();
        }
            return singleInstance;
    }
}
```

Eager initialization

This is a design pattern where an instance of a class is created much before it is actually required. Mostly it is done on system start up. In singleton pattern, it refers to create the singleton instance irrespective of whether any other class actually asked for its instance or not.

```
public class EagerSingleton {
    private static volatile EagerSingleton instance = new EagerSingleton();
    // private constructor
    private EagerSingleton() {
    }
    public static EagerSingleton getInstance() {
        return instance;
    }
}
```

Above method works fine, but has one drawback. Instance is created irrespective of it is required in runtime or not. If this instance is not big object and you can live with it being unused, this is best approach.

This problem can be handled with lazy initialization.

Lazy initialization

In computer programming, lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. In singleton pattern, it restricts the creation of instance until requested first time. Lets see in code:

```
public final class LazySingleton {

    private static volatile LazySingleton instance = null;

    // private constructor

    private LazySingleton() {

    }

    public static LazySingleton getInstance() {

        if (instance == null) {

            synchronized (LazySingleton.class) {

                instance = new LazySingleton();

            }

        }

        return instance;

    }

}
```

On first invocation, above method will check if instance is already created using instance variable. If there is no instance i.e. instance is null, it will create an instance and will return its reference. If instance is already created, it will simply return the reference of instance.

But, this method also has its own drawbacks. Lets see how. Suppose there are two threads T1 and T2. Both comes to create

instance and execute “instance==null”, now both threads have identified instance variable to null thus assume they must create an instance. They sequentially goes to synchronized block and create the instances. At the end, we have two instances in our application.

This error can be solved using double-checked locking. This principle tells us to recheck the instance variable again in synchronized block in given below way:

```
public class EagerSingleton {

    private static volatile EagerSingleton instance = null;

    // private constructor

    private EagerSingleton() {

    }

    public static EagerSingleton getInstance() {

        if (instance == null) {

            synchronized (EagerSingleton.class) {

                // Double check

                if (instance == null) {

                    instance = new EagerSingleton();

                }

            }

        }

        return instance;

    }

}
```

Above code is the correct implementation of singleton pattern.

Do you need Singleton pattern?

Singletons are useful only when you need one instance of a class and it is undesirable to have more than one instance of a class. When designing a system, you usually want to control how an object is used and prevent from making copies of it or creating new instances. For example, you can use it to create a connection pool. It’s not wise to create a new connection every time a program needs to write something to a database; instead, a connection or a set of connections that are already a pool can be instantiated using the Singleton pattern.

The Singleton pattern is often used in conjunction with the factory method pattern to create a systemwide resource whose specific type is not known to the code that uses it. An example of using these two patterns together is the Abstract Windowing Toolkit (AWT). In GUI applications, you often need only one instance of a graphical element per application instance, like the Print dialog box or the OK button.

Lazy Initialisation

Lazy Initialization is a technique that initializes a variable (in OO contexts usually a field of a class) on it's first access. It canonical form is something like this:

```
public FooClass Foo {
  get {
    if (_foo = null) _foo = calculateFoo();
    return _foo;
  }
}
```

Lazy initialization is useful when calculating the value of the field is time consuming and you don't want to do it until you actually need the value. So it's often useful in situations where the field isn't needed in many contexts or we want to get the object

initialized quickly and prefer any delay to be later on.

Remember that this is an optimization technique to help responsiveness in situations where a client doesn't need the lazy initialized value. As with any optimization you shouldn't use this technique unless you have a real performance problem to solve.

lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This is typically accomplished by maintaining a flag indicating whether the process has taken place. Each time the desired object is summoned, the flag is tested. If it is ready, it is returned. If not, it is initialized on the spot.

Lazy initialization is a performance optimization. It's used when data is deemed to be 'expensive' for some reason. For example:

- if the `hashCode` value for an object might not actually be needed by its caller, always calculating the `hashCode` for all instances of the object may be felt to be unnecessary.
- since accessing a file system or network is relatively slow, such operations should be put off until they are absolutely required.

Lazy initialization has two objectives:

- *delay* an expensive operation until it's absolutely necessary
- *store* the result of that expensive operation, such that you won't need to repeat it again

As usual, the size of any performance gain, if any, is highly dependent on the problem, and in many cases may not be significant. As with any optimization, this technique should be used only if there is a clear and significant benefit.

A real-world example

Let's now examine a more realistic example, where lazy instantiation can play a key role in reducing the amount of resources used by a program.

Assume that we have been asked by a client to write a system

that will let users catalog images on a filesystem and provide the facility to view either thumbnails or complete images. Our first attempt might be to write a class that loads the image in its constructor.

```
public class ImageFile
{
  private String filename_;
  private Image image_;
  public ImageFile(String filename)
  {
    filename_=filename;
    //load the image
  }
  public String getName(){ return filename_;}
  public Image getImage()
  {
    return image_;
  }
}
```

In the example above, `ImageFile` implements an overeager approach to instantiating the `Image` object. In its favor, this design

guarantees that an image will be available immediately at the time of a call to `getImage()`. However, not only could this be painfully slow (in the case of a directory containing many images), but this design could exhaust the available memory. To avoid these potential problems, we can trade the performance benefits of instantaneous access for reduced memory usage. As you may have guessed, we can achieve this by using lazy instantiation.

Here's the updated `ImageFile` class using the same approach as class `MyFrame` did with its `MessageBox` instance variable:

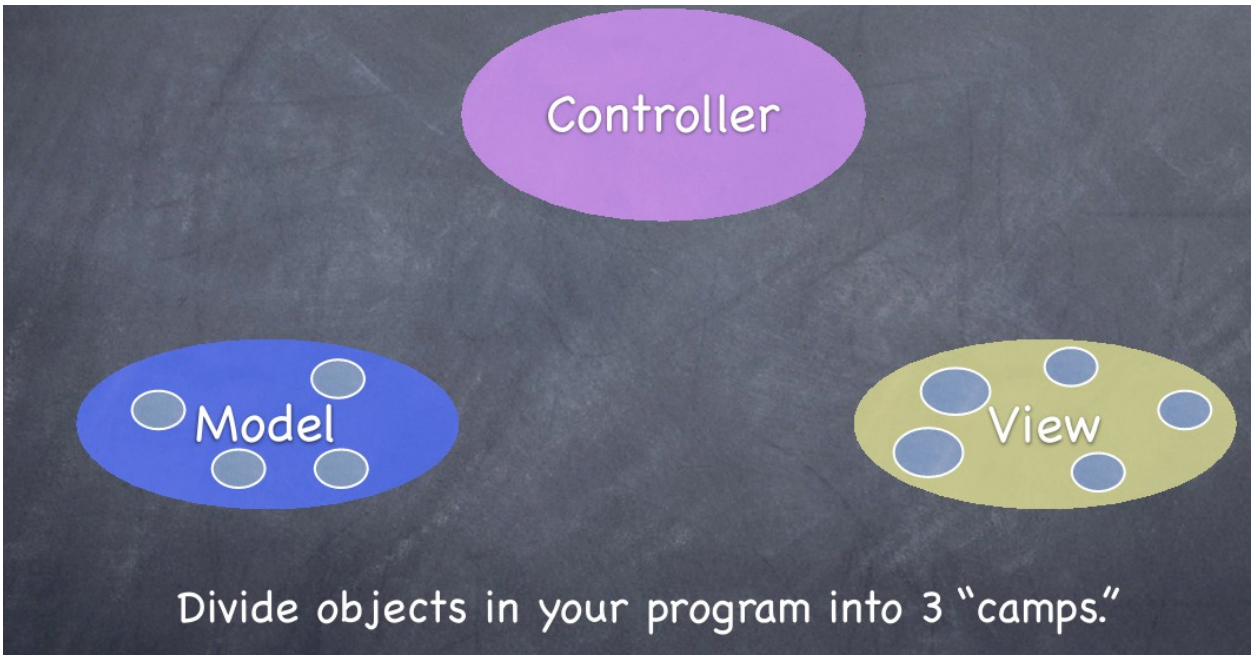
```
public class ImageFile

{

  private String filename_;

  private Image image_; //=null, implicit

  public ImageFile(String filename)

  {

    //only store the filename

    filename_=filename;

  }

  public String getName(){ return
filename_;}

  public Image getImage()

  {

    if(image_==null)

    {
```

```
    //first call to getImage()

    //load the image...

   }

   return image_;

  }

}
```

In this version, the actual image is loaded only on the first call to ***`getImage()`***. So to recap, the trade-off here is that to reduce the overall memory usage and startup times, we pay the price for loading the image the first time it is requested.

Lazy instantiation is dangerous in multithreaded applications, so use singleton pattern.

Model View Controller (MVC)

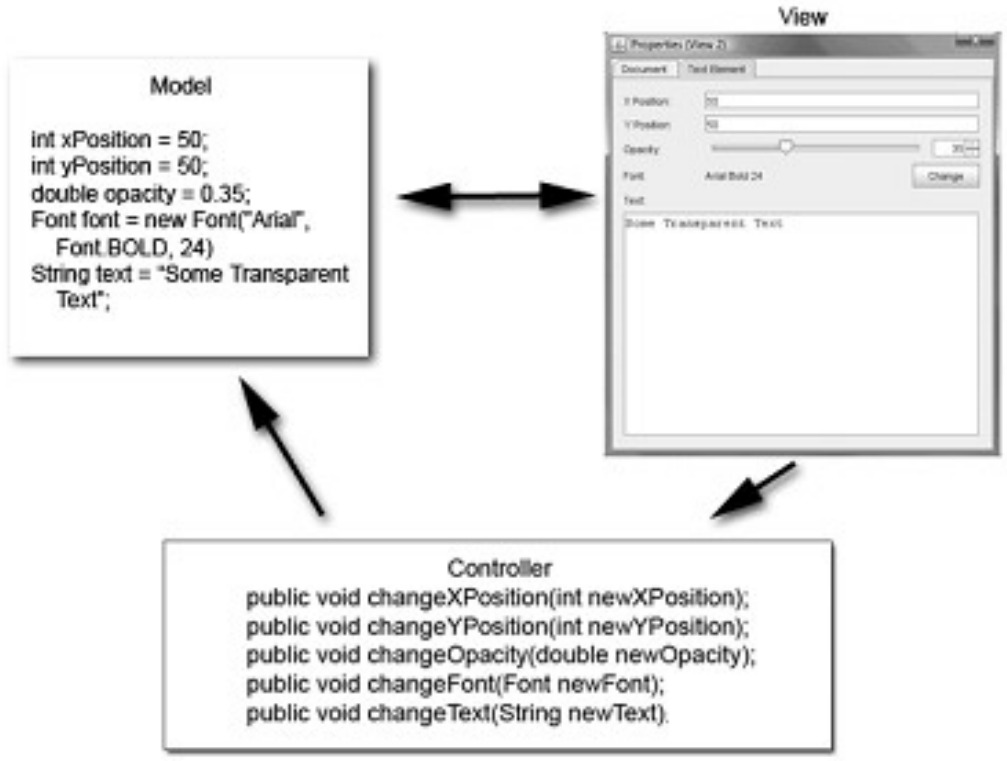


- It's all about managing communication between camps
- Model = What your application is (but not how it is displayed)
- Controller = How your Model is presented to the user (UI logic)
- View = Controller's minions
- **Name** (essence of the pattern)
  - Model View Controller MVC
- **Context** (where does this problem occur)
  - MVC is an architectural pattern that is used when developing interactive application such as a shopping cart on the Internet.
- **Problem** (definition of the reoccurring difficulty)
  - User interfaces change often, especially on the internet where look-and-feel is a competitive issue. Also, the

same information is presented in different ways. The core business logic and data is stable.

- **Solution** (how do you solve the problem)
  - Use the software engineering principle of “separation of concerns” to divide the application into three areas:
    - **Model** encapsulates the core data and functionality
    - **View** encapsulates the presentation of the data there can be many views of the common data
    - **Controller** accepts input from the user and makes request from the model for the data to produce a new view.

MVC Example



- The **Model** represents the structure of the data in the application, as well as application-specific operations on those data.
- A **View** (of which there may be many) presents data in some form to a user, in the context of some application function.
- A **Controller** translates user actions (mouse motions, keystrokes, words spoken, etc.) and user input into application function calls on the model, and selects the appropriate View based on user preferences and Model state.

MVC Flow

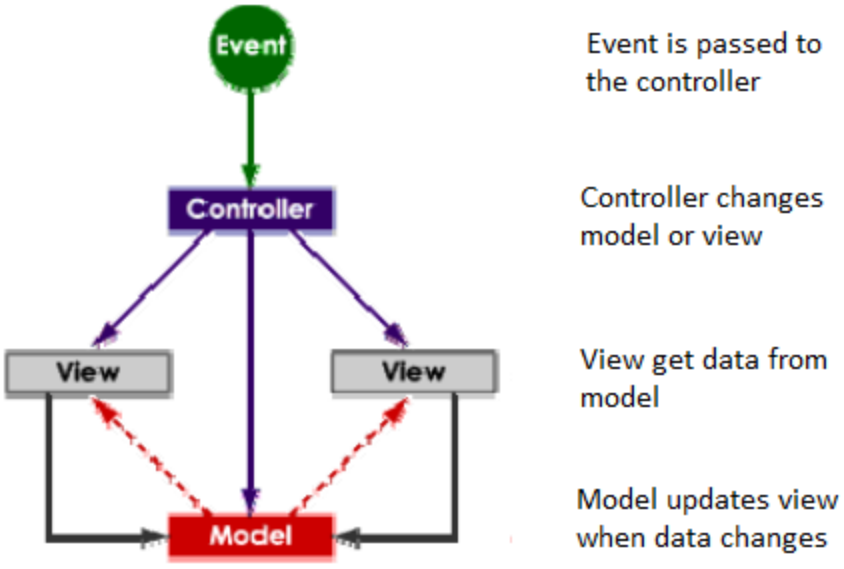
The user manipulates a view and, as a result, an event is generated. Events typically cause a controller to change a model, or view, or both. Whenever a controller changes a model's data or properties, all dependent views are automatically updated. Similarly, whenever a controller changes a view, for example, by revealing areas that were previously hidden, the view gets data from the underlying model to refresh itself.

It can be observed that:

• Both the view and the controller depend on the model.

• The model depends on neither the view nor the controller.

• The separation between views and controller is secondary in many rich-client applications, and, in fact, many user interface frameworks implement the roles as one object.

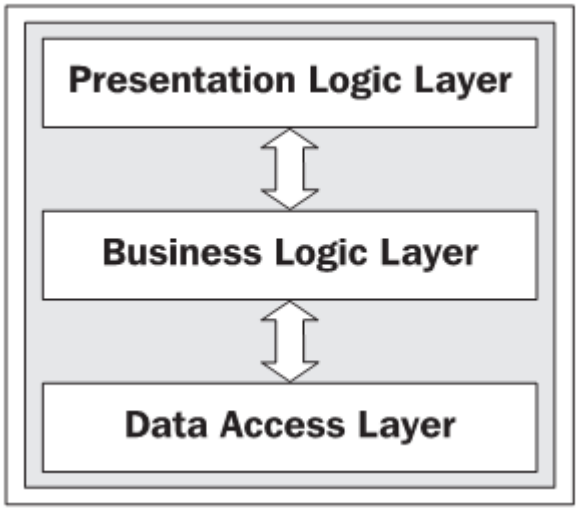


Layers and Separation of Concerns

Multi-tier software architecture should do the job. The most commonly used architecture is one with three tiers or layers. These are:

- Presentation Logic Layer
- Business Logic Layer
- Data Access Layer

Separating these layers in project development cycles allows achieving rapid application development with project maintainability in mind. In the three-tier architecture, the interaction between layers is shown in the following diagram:



At the Data Access Layer, we find both the data and the ways to extract the data that we want to show to the user. The Data Access Layer may contain:

- A database (MySQL, PostgreSQL, MSSQL, and so on) and the SQL language used to extract data from the database
- Files that store data and functions (e.g. PHP functions or other languages) that parse the files

- Data acquisition software (for example, a thermometer on the parallel port)

Now that we've extracted the data, we need to manipulate it in order to get the results we need to display. Data manipulation and validation is done at the Business Logic Layer, which may contain:

- Data validation based on the business plan (for example, list only in-stock items)
- Data manipulation according to the business plan (for example, discounts, stock liquidations, and so on)
- Functions and formulas to calculate things like shipping expenses, and so on.

The Presentation Logic Layer is where the web page layout—how data from the Business Logic Layer is arranged in the web page—is defined. This is done using:

- Web page templates
- Text/CSS
- Images, banners, and menu styles

Without a templating engine (like PHP Smarty) at the Presentation Logic Layer, we find HTML and PHP creating the layout with pure HTML and generating HTML code from PHP. In this case, we cannot divide the Presentation Logic and Business Logic layers into two separate layers, making the work of designers and programmers very difficult for complex software projects. That's where Smarty comes in.

Reference:

http://martinfowler.com/bliki/LazyInitialization.html

msdn.microsoft.com/en-us/library/dd997286(v=vs.110).aspx
http://en.wikipedia.org/wiki/Lazy_initialization

http://www.javaworld.com/article/2077568/learn-java/java-tip-67--lazy-instantiation.html

http://www.journaldev.com/2394/dependency-injection-design-pattern-in-java-example-tutorial

http://howtodoinjava.com/2012/10/22/singleton-design-pattern-in-java/ http://tutorials.jenkov.com/dependency-injection/index.html

http://martinfowler.com/articles/injection.html

http://en.wikipedia.org/wiki/Factory_method_pattern

http://www.oodesign.com/factory-pattern.html

http://alvinalexander.com/java/java-factory-pattern-example

http://www.tutorialspoint.com/design_pattern/factory_pattern.htm