While placing comments in code is good, it is not always easy to read. So here is a better rendition.

# Why use Spring

## Dependency Injection(DI)

In the past few years, thanks to the populairity gained by Spring, DI has gained wide acceptance among Java Developers.

Benefits of DI include

### Reduced glue code

The amount of code you have to write to glue all of you objects together is greatly reduce. With DI you Inject an instance of an object into a class that will use it. No need to do new and clean up.

### Simplified application configuration

You use annotations or xml to configure those class that are injectable into other classes.

### The ability to managed common dependencies in a single repository.

Using the traditional approach to dependency management of common service, for example a datasource connection, transaction remote services, etc, you create instance or lookup in some JNDI repository.

When you use DI, all the information about those common dependencies is contained in a single repository making management much similar.

### Improved Testablilty

When you design your classes with DI, you can seamlessly integrate your code with Junit and Mock Testing Frameworks using Annotations to inject Mocks seamlessly into the test framework.

### Fostering good application design

Designing Di provides a lot of benefits for your application.

* Designing against interfaces
* Good programed decisions
* Faster program changes.

### Aspect Oriented Programming

Is one of the programming models of the minute. You can inject behavior into your program with AOP. By defining a dynamic proxy, to the target object and weaving the objects with the configured advice into your program to execute the cross cutting logic.

### Validation Using Spring

The Java Community process JCP developed he Bean Validation API Specification (JSR-303). The Bean validation API provides a standard way for defining bean validation rules. For example when applying the @NotNull annotation to the beans property, it menas the attribute shouldn’t contain a null value before being able to persist into the database.

### Accessing Data In Spring

Spring contains support for JDBC, Hibernate, MyBatis, Java Data Object (JDO), and the Java Persistence API (JPA).

### Object/SML Mapping (OXM in Spring

Spring supports many common xml – java formatting tools.

### Managing Transactions

Spring provides an excellent abstraction layer for transaction management.

### Mail Support

### Job Scheduling Support

Left off here.

# Golf Program Explained

## The Main Program

The first class you will encounter is the MainApp.java. It has just one method the main method. It gets everything started. In order for a class to be an executable it must have a method with the following signature

public static void main(String[] args)

Spring provides this class GenericXmlApplicationContext for free when you add a spring-core dependency to the maven pom.xml

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>3.1.0.RELEASE</version>

</dependency>

This places a jar file in your home directory under a folder .m2/repository/org.springframework/spring-core/3.1.0.RELEASE

The GenericXmlApplicationContext reads an xml file from the classpath : app-contact.xml . The file can be anything you want, you define it in the call to GenericXmlApplicationContext.

This loads up a Spring context. The Spring context is beyond the scope of this document, but it provides a run time environment much like the JVM, but as you will see is more complex. The refresh begins to load the Spring context, and reads the xml file which instructs it to do some things. Number 1 is read in the xsd files that will validate the structure of xml files that can be used.

Secondly we need to define the datasource that will provide the connection parameters to postgres. The username and password are configurable per developer. Postgres always runs on port 5432. This can be changed in postgres, and is configurable in Hibernate as well. It uses JDBC as its underlying mining engine so it is defined as well.

Thirdly with a dataSource defined we now can construct a Session Factory. This is a hibernate session. Notice the dataource is passed in . Additionally the session will read artifacts from a property named packagesToScan. There can be many packages , here we define only one because that is all we have provided Spring components for. This is a Spring file that will be stored in Springs Context and can be retrieved with getBean(“contactDao”) that you will see. Eliminates the need for the new operator and as we grow in our Spring knowledge, you can inject other Spring Beans into other files.

### Spring Annotations

Spring defines 4 annotations. You will see these 4 annotations defined in various classes in this application. @Componet was an annotation defined in Spring 2.0. Spring 2.5 introduced the remaining 3 annotations.

Annotation | Meaning |

+------------+-----------------------------------------------------+

| @Component | generic stereotype for any Spring-managed component |

| @Repository| stereotype for persistence layer |

| @Service | stereotype for service layer |

| @Controller| stereotype for presentation layer (spring-mvc) |

#### @Repository

The golf program has contacts. Contacts define member names, home addresses and phone numbers for the members of the league. It is expected that members will be added to the league, removed from the league, and require editing to change addresses and phone numbers.

In spring you should write all classes from interfaces.

Hence you will see a **ContactDao**, and a **ContactDaoImpl**.

Spring instantiates classes that are annotated with the proper annotations. How does Spring know where to find these classes?

Revisiting the app-context.xml file in /src/main/resources provides the answer. When we bootstrap Spring, it finds base packages in this configuration file and scans through reflection instantatiates the classes.

<context:component-scan base-package="com.myapps.golfprogram.golfprogram.dataaccess" />

@Repository("contactDao")

@Transactional

public class ContactDaoImpl implements ContactDao{

The main method goes on to pull the class from the Spring Repository. Each Annotated class that is instantiated by the Spring context, is stored in a Spring Repository as a bean.

final ContactDao contactDao = ctx.getBean("contactDao", ContactDao.class);

The contact bean instantiates the contact class to retrieve all contacts.

final List<Contact> contacts = contactDao.findAll();

The golf Toll is not constructed with the contact list.

GolfTool golfTool = new GolfTool(contacts, contactDao);

Finally the GolfTool is asked to construct itself, passing in the ContactDao that was retrieved from the Spring Context, and passes in the contacts that were retrieved from the ContactDao. Which leads to a couple of TODO things we should get to soon.

The First TODO is why pass in the ContactDao, and then pass in what we got from the Dao. . Why not just pass in the DAO, and inside the tool use the DAO to get the list of contacts.

The second TODO is the look in the GolfTool Constructor and make some good programming decisions about what to do with the GolfView and GolfModel. A constructor of any class is designed to accept arguments and to create and populate class members The Golf Tool instantiates a GolfModel and GolfView. Both of these are references to objects. References to objects of this type should not be mutable. Once The GolfView is created, we would not want to ever change it. We may update is publically accessible members, but the reference points to a memory location. Always make references of this sort final. What should we have done? Declare the objects final like

Instead of

private GolfModel golfModel;

private GolfView golfView;

private final GoflModel golfView;

private final GolfModel golfModel;

In the Golf Tool Constructor creates these objects.

golfModel = new GolfModel(contacts);

golfView = new GolfView(this,golfModel);

Once they are created, any attempt to change them or create a new object with these members will be a compilation error. Try if for yourself. Create a private final String fooString = “foo”; Somewhere in your program try to set fooString to a different value. It will simply not compile. It will complain about not being able to modify a final variable.

Should do the same thing with ContactDao. We do not want a different reference to that object either.

Once the Golf Tool is constructed, the following 3 lines show the Contacts Frame where most of the interaction with Contact will take place. The Main App should not be a controller for anything in the Golf MVC Framework This work should take place in the Golf Tool and Golf View. There should be a method in the Golf View that sets its size, and sets its own visibility. . The Tool will call this method to show bring the Golf View up.

With these changes in place setVisible to True brings the Golf View to life and you will see the contact panel.

## Model View Controller

Lets look at our first example of this.

Ideally the Controller is the traffic cop of the program. The controller handles business logic like going to the Persistence Layer to get contacts, scores, and contact details. The controller also handles communication between the Model and The View. When a user selects a Button for example to save contacts, we do not want to mix business logic with View , so the View notifies the tool of this save , and passes the data to be persisted along.

The Ideal behind the view is it can be swapped out with little impact to the program. A view may be represented by Swing, HTML, Servletts, or Portals. Every attempt to isolate the Tool from any view implementation specifics that will be tied to one of the above must be avoided.

The Model is designed to back the view. If the model and view are properly designed, the tool will not need to visit the view to make changes to one of its UI components. WE update the Model and the Model will update the view.

Lets look at an example of this in the Golf program. A JCombo Box is a Swing component that displays a list of in our case Strings, that represent names. Note that the name Roser, Brett or Roser, Pat is not stored in the Database as such. These are stored in the DB as last\_name, and first\_name. Programmatically we build the String that is represented by taking these two db items, and concatenating them together as in contact.getLastName() + “, “ + contact.getFirstName();

While the building of the Strings is not important, how they arrive in the JComboBox is . Every Swing Component is backed by a Model. JComboBox is backed by a DefaultComboBoxModel. JTextField is backed by a PlainDocumentModel. When you interact with the model, you are not required to perform any action on the JComboBox. Removing an item from the DefaultComboBoxModel automatically updates the view.

In the View you will see these two lines which create a Model, and a JComboBox

private DefaultComboBoxModel comboModel = new DefaultComboBoxModel();

private JComboBox memberCombo = new JComboBox(comboModel);

When we save a contact the following lines come into play

//The Tool saves the data in the DB.

contactDao.save(data);

// WE ask the View for the DefaultComboBoxModel, and add an element to it. Notice we do nothing

// with the JComboBox itself. Updating the model updates the view and the JCombo Box doesn’t know //about it.

golfView.getComboModel().addElement(data.getLastName() + ", " + data.getFirstName());

That is really all there is to the MVC architecture.

If this were about UI development, all of our JTextFields would never get a setText(String content) call. We would ask each JTextField for its PlainDocumentModel, and calls its insertSting method(….);

We could then create an interface that is not Swing Specific and add a method updateFirstName(String firstName). Once we define an interface for this, Each implementation class that implements this interface would handle the details of what type of UI components they are , and properly address the setting of its backing models.

## Golf Tool Golf Model Golf View

Here are the classes that represent the Initial activity of Starting the golf program.



So first thing I notice when I step through this initial startup is some things thing that must change. Why is the Golf Tool creating a JFrame. That breaks the Model View Controller pattern. The Golf Tool should also not be creating a second UI class called Contact Panel. Not shown is also a button Panel being constructed. All of these should be moved to their own package ui.contacts. We won’t change any code at this point, we are just creating new classes that will be more MVC compatible.



To fix The Constant “Add Contact Panel” I am expecting a Constants file that will hold all of our UI Strings. This constants file will also hold items for other UI components as well. Since we have a package com.myapps.golfprogram.golfprogram.ui, This looks like a great place for a constants file.

Create a java class file here ( Name it whatever you like as long as it is self-describing) and add a Constant to it with some name like CONTACT\_ADD\_PANEL\_NAME; It is typical in Java to capitalize all Constants member names and have an underscore separating full words. Each constants should be final static.

No on to placing the JFrame construction , and ContactPanel , and the ButtonPanel in a more suitable place. What we should have is a Class that extends JFrame , and is responsible for creating the add contacts frame. I would place this file in package com.myapps.golfprogram.golfprogram.ui.contacts. Also create a class Called ContactPanel here. A Third for the ButtonPanel. Since the AddContactsFrame, extends JFrame you could call **super** in the constructor and pass in the Constant you created above in it.

What does super do? When we extend a class like JFrame, JFrame becomes the super class. So the super(..) call calls the super class constructor and creates a JFrame with the argument passed in, In this case your constant. Super(..) must always be the first line in your constructor of any class. Failure to observe this will result in a compilation error.

So you should be able to complete these new classes without changing any code that may call them. A good rule of software engineering, is baby steps. Let’s get this far, code review, checkin or in get speak push to origin/master, and drink beer. Then move on .

## Refactoring the Add Contacts.

Program interactions before the Refactoring effort. The MainButtonPanel class that contained the buttons that allowed a user to interact with the program, is shown below.



The actionPerformed is a required method since we elected to implement the ActionListener Interface in the MainButtonPanel. The Source of the button selected is checked with a conditional if statement.

When reading a class diagram above, the very first two items you see: JPanel and actionListener shown in italics are the interfaces this class implements. The name of the class and its first package scope follow. Package scope is ui, and the class Name is MainButtonPanel. There is a section omitted. If this class had a constructor, it would be shown next. The next section of items are the members of the class The “-“ in front of the members is a scope of private. Public is “+”, and Protected is “#”. The last section of the document is the methods, or operations allowable on this class. Only public methods are accessible outside the class, private methods can only be executed inside the class.

The actionPerformed method does all of the heavy lifting here. It responds to any button press and calls some “delegate”, a word used in programming to indicate a generic handler of some action. The delegate in this case is the frame class. The frame class calls the GolfTool to perform some actions.

@Override

public void actionPerformed(ActionEvent e) {

if (e.getSource() == closeButton) {

frame.notifyCloseSelected();

} else if (e.getSource() == addButton) {

frame.notifyAddSelected();

} else if (e.getSource() == deleteButton) {

frame.notifyDeleteSelected();

} else if (e.getSource() == editButton) {

frame.notifyEditSelected();

} else if (e.getSource() == addScoreButton) {

frame.notifyAddScoreSelected();

}

}

Referring to the class diagram above, notice the MainButtonPanel takes the Golf View as an argument and hangs on to a reference to it. . So when button selections are made, the MainButtonPanel should not contain any business logic. So it hands off the Logic to the GolfView to implement any actions. Since any View should not handle business logic it in turn simply notifies the Golf Tool to handle business Logic.

This method in the Golf Tool handles the business logic that will bring up the AddContact Panel you have just created, and must have a way to notify the Controller of the Add Contact implementation to call our contactDaoImpl class when we need to save a Contact.

public void notifyAddSelected(){…}

This creates an instance of the old AddContactPanel, and passes a reference to itself (Golf Tool) to it. The AddContactPanel has its own buttons save, and cancel. The save button on the Add Contact Panel, will call the GolfTools notify save selected to save the new contact into the Postgres DB. It does this by using the Spring Managed ContactDaoImp.



I think we can simplify this model now. WE still need the golf view, but by passing in the GolfTool, we can cut out the middle man out of his equation. The middle man is going to the view.

Try this

