Hibernate 'til Spring
Benefits of Spring MVC, Hibernate and Apache Tiles to Support Quality Attributes and Usability within Web Applications

Chris O'Brien

April 2, 2014

UNIVERSITY of LIMERICK

OLLSCOIL LUIMNIGH

Abstract

Web development is one of the fastest growing areas in software development, with new tools being developed yearly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Introduction - This whole section needs to be reworks, place holder from interim

This project concerns the development of a web application using a web framework in conjunction with a number of other tools. Throughout development, there is a particular cognisance towards the support of Non-Functional Requirements [NFRs] by both the web framework and the supporting tools throughout the development process.

The main goal of this project is to reflectively analyse a WAF [Web Application Framework], and architecture stack, in the creation of a website. This will be analysed in respect to both functional and non-functional requirements. Two key requirements are extensibility and maintenance. Extensibility refers to the ability of the framework to allow added functionality to the web application without having to modify the core workings of the application. Maintenance refers to the upkeep of the code, and facilitates the modification of the source code after the product is deployed. This may be to correct faults, improve attributes such as performance and security. The creative driver of the project is the development of a website to meet the requirements and needs of Monaleen Tennis Club, for both members of the club and of the committee. These needs will overlap as all committee representatives are all club members, but not all members are on the committee. From this, it was important to identify the precise requirements for each type of user. The main focus of this project was for the club to be able to perform their core functions through the website. This extended to the registration of members, a timetable for the courts, the creation and distribution of tournament schedules, the organisation and timetabling of training sessions, a method to contact all members and a news section to update and advise members of changes and upcoming events .

- Member Management
- Timetable Management
- Tournament Management

1.2 Technology

1.3 Objectives

1.3.1 Primary Objectives

The primary objectives of this project are the support for quality attributes that the use of an architectural stack provides. The architectural stack within this project contains Spring MVC, Hibernate ORM and Apache Tiles.

This project will examine the support provided for specific quality attributes: Security, Productivity, Extensibility and Performance. The provision of these attributed as provided by these frameworks will be examined, with cognisance of usability.

1.3.2 Secondary Objectives

The secondary objectives of this project are:

- To gain knowledge of the development of a web application using frameworks
- To obtain experience working with a architectural stack common to industry
- To use metrics and code visualisations within a project in order to guide and aid development of an application.

1.4 Scope

1.5 Methodology

Chapter 2

Background

2.1 Introduction

There are a number of components needed to build the architecture of a web application. The nature of these components is explored below, and their contribution to the creation of a web application is analysed. A more detailed breakdown on their usage within the application is explored in subsequent chapters.

2.2 Support for Quality Attributes

Within the scope of this application, there were a number of quality attributes that were examined. A quality attribute is "part of an applications non-functional requirements, which capture the many facets of *how* the functional requirements of an application are achieved" (Gorton 2006). They can related to system design, or may be specific to run time or user centric issues.

- Security
  - This is the capability of the system to prevent actions that are outside the designed usage, or actions that are malicious in origin. Within an application, "security boils down to understanding the precise security requirements for an application, and devising mechanisms to support them" (Gorton 2006).
- Performance
  - This is the capability of the system to respond to actions performed on it within a reasonable time frame. It "defines a metric that states the

amount of work an application must perform in a given time" (Gorton 2006).

- Usability
  - This is the capability of the system to cater to the requirements of the user, and the how intuitive the system is to use.
- Extensibility
  - This is the capability of the system to cater for future requirements. It is a "measure of how easy it may be to change an application to cater for new functional and non-functional requirements" (Gorton 2006)
- Productivity
  - This is the influence that the architectural choices of the system affects developer productivity, which has an effect on other quality attributes.

Table 2.1: Quality Attributes

2.3 Technologies

2.3.1 Web Application Framework

The Web Application Framework [WAF] chosen for this project is Spring MVC. A WAF is a defined support structure in which other software applications can be organized and developed (Shan and Hua 2006). Spring MVC is based on the Model-View-Controller [MVC] design pattern. The intention of this pattern is to form a clear division between domain objects and presentation objects. The Model manages the behaviour and data of the application, shown in its classic form in Figure 2.1. The View manages the information obtained from the model and displays it to the user. The Controller manages user input, such as key strokes, mouse movements or a touch display, and can interact and invoke functionality within the Model and/or View.

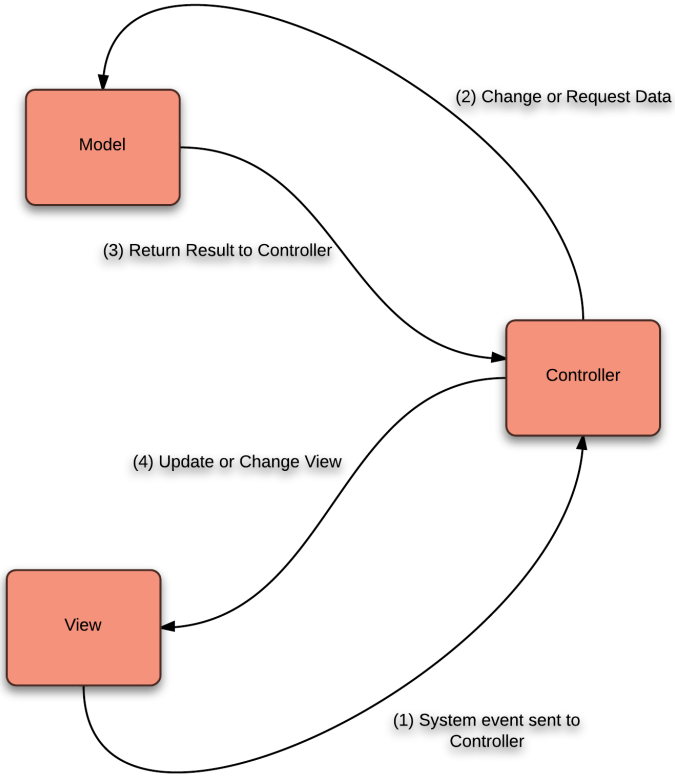

Figure 2.1: Classic MVC

Spring MVC has a *DispatcherServlet*. This is defined within the *web.xml* file, shown in Figure 2.3. This file is located in the WEB-INF folder. Its purpose is to load the application context from a servlet file, defined within this application as *member-servlet.xml*. An Application Context is an interface within the framework. A Servlet is a Java class that is responsible for extending the functionality of a server. This interface provides the configuration for the application.

Spring MVC provides a clear separation of roles. Each role, such as a View, Controller, Mapper, Validator, Model and View Resolver, can be encapsulated within a relevant object. It is a request driven framework designed around the *DispatcherServlet*.

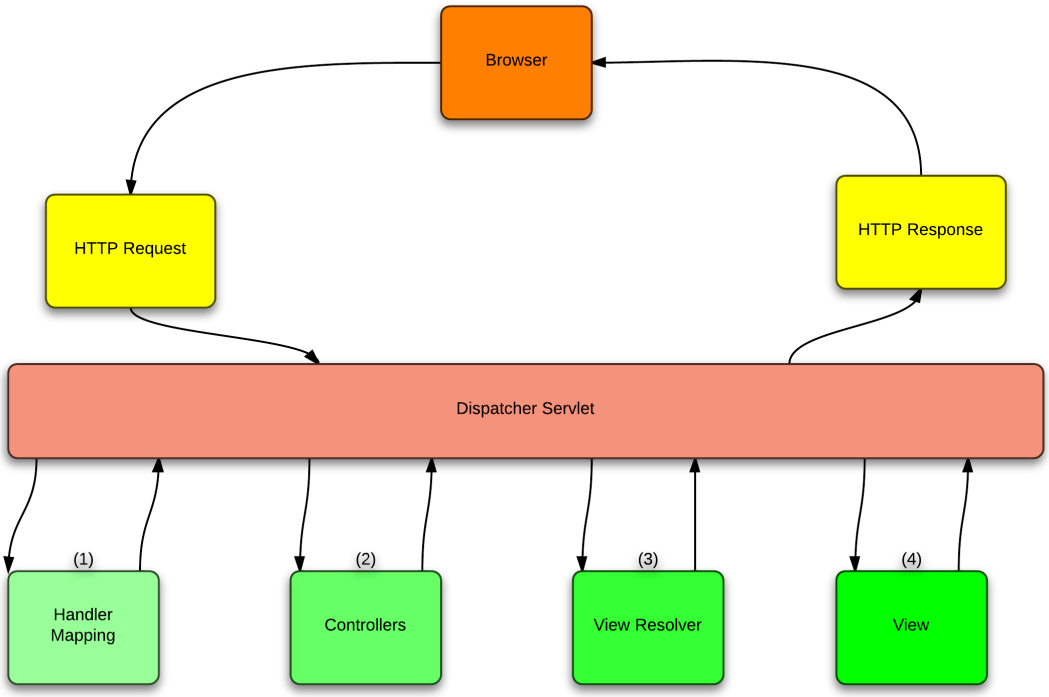

Figure 2.2: DispatcherServlet

The *DispatcherServlet* is responsible for handling HTTP requests from the browser. Once it receives these requests, it consults the HandlerMapper, which calls the appropriate Controller. A HandlerMapper takes a value, such as "/admin", and checks which controller handles this mapping. The Controller will take this request, and call the appropriate method, or methods, and interact with the Service layer of the application, if necessary. The Service layer of the application consists of the group of classes that are responsible for facilitating communication between the controller classes and their respective dao classes. The View name is then returned to the *DispatcherServlet*, which in turn passes the value to the ViewResolver.

A ViewResolver provides a mapping between a *view name* and the *view*, that is to say, the web page requested. A *view name* is represented as a string value that is returned from a controller. Once this view is finalised, the *DispatcherServlet* passes any model created within the Controller to the View, which is then rendered by the browser. This process is shown in Figure 5.2

```
1 <web-app
2 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3 xmlns="http://java.sun.com/xml/ns/javaee"
4 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
```

```
version="2.5">
  <display-name>monaleen-tennis</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <description></description>
    <display-name>members</display-name>
    <servlet-name>members</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>members</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
  <description>Database</description>
  <resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/mtc</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        classpath:beans/dao-context.xml
        classpath:beans/service-context.xml
        classpath:beans/security-context.xml
        </param-value>
  </context-param>
</web-app>
```

Figure 2.3: Spring DispatcherServlet Configuration

Figure 2.3 is the configuration file needed for the *DispatcherServlet*. The file has a number of responsibilities within the application.

- Define DispatcherServlet
  - Line 9: This defines what class the DispatcherServlet implements.

```
1  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
```

- Define ApplicationContext
  - Line 17-19: This specifies the file that defines the application context of the application

```
1  <servlet-mapping>
2  <servlet-name>members</servlet-name>
3  <url-pattern>/</url-pattern>
4   </servlet-mapping>
```

- Define DataSource
  - Lines 22-27: This defines the location of the database, and the DataSource class. A DataSource class creates a connection from the application to the physical data source.

```
1  <resource-ref>
2  <description>DB Connection</description>
3  <res-ref-name>jdbc/mtc</res-ref-name>
4  <res-type>javax.sql.DataSource</res-type>
5  <res-auth>Container</res-auth>
6  </resource-ref>
```

- Define Context Config Location
  - Lines 28-35: This defines the files that contain the configuration for the DAO, Service and Security context files.

```
1  <context-param>
2  <param-name>contextConfigLocation</param-name>
3  <param-value>
4  classpath:beans/dao-context.xml
5  classpath:beans/service-context.xml
6  classpath:beans/security-context.xml
7  </param-value>
8  </context-param>
```

Table 2.2: DispatcherServlet Code

The *Controller*, within the Spring MVC framework, creates a model with data, and selects a view which will represent that data. This is done through the use of a *RequestMapping* annotation, which is discussed in further detail in Section 5 of this report.

The default *ViewResolver* within the Spring MVC is the InteralResourceViewResolver, depicted in Figure 2.4. This is defined with the *members-servlet.xml* file in the application, which is the structure of the *DispatcherServlet*. This class takes the value that is returned by a Controller, and passes a View to the DispatcherServlet. The browser can then render this view. It is important that any views, such as JSP files within the scope of this application, are stored within the *WEB-INF* folder. This is to ensure that the files are treated as an internal resource, and as such, are only accessible by the servlet, or the Controller classes within the application.

```
<bean id="jspViewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
      <property name="prefix" value="/WEB-INF/jsps/"></property>
      <property name="suffix" value=".jsp"></property>
</bean>
```

Figure 2.4: Default ViewResolver Configuration

This *ViewResolver* was not used within this application. Instead, Apache Tiles provides its own *ViewResolver*. This is due to the changes in how JSP pages are constructed and displayed by Apache Tiles, as discussed in the next section.

2.3.2 View Resolver

The framework that provided the *ViewResolver* for this application was Apache Tiles. This framework allows for the composition of a template for a JSP page. Apache Tiles allows the application developer to define page fragments, which are assembled into one page at run time, based on a template. This allows the application to reduce duplication of common page elements, such as headers, footers, link bar and advertising. The defined templates allow for a consistent look and feel across the application, and a change in one place, such as modifying a link in a header, will change across all Views within the application.

In order to use the Apache Tiles *ViewResolver*, it must be defined within the *DispatcherServlet* XML file, see Figure 2.5, in lieu of the default ViewResolver. This ViewResolver is part of the Spring Framework, and allows for interoperability between both the Spring and Apache Tiles frameworks. The reason that *TilesViewResolver* is used instead of *SimpleTilesListener* is for the support of JavaServerPages

Standard Tag Library [JSTL] within the JSP pages, as discussed within the Implementation. JSTL uses tags to perform conditional functions without the use of directives with a JSP page. A directive is instructions that are performed within a JSP page, such as running Java code.

```
<bean id="tilesViewResolver"
      class="org.springframework.web.servlet.view.tiles2.TilesViewResolver">
</bean>
```

Figure 2.5: Default ViewResolver Configuration

2.3.3 Application Server

The application server, or web server, used for this project was Tomcat 7. Tomcat is an open source project by Apache, which is a software implementation of Java Servlet and JavaServer Pages technologies. This application provides an environment in which Java code can run. Tomcat has a servlet-container called Catalina. A servlet container is the part of the web server that interacts with the servlets created by the application. It manages the life-cycle of the servlets, and is responsible for specifying the run time environment for the components within the application, and delivering this content. This includes security, transaction management, deployment and other services.

2.3.4 Project Management Tool

The project management tool used for this project was Maven. Maven was used within the scope of this project to manage the dependencies required by the web application. Dependencies within the project include the JAR files required for Hibernate, Spring MVC, Apache Tiles and the many other frameworks used in the project. A JAR is a archive package that contains Java packages and .class files. Maven came pre-installed and configured within the Spring Tool Suite IDE. Dependency Management can be handled one of two ways. Dependencies can be added using the GUI interface provided by an IDE, in this case, Spring Tool Suite. This GUI links to the repository located at http://mvnrepository.com/, and the user searches for the required files. Otherwise, the *pom.xml* file may be edited to define dependencies manually. Below is an example of the Apache Tiles v3.0.3 dependency.

```
<dependency>
        <groupId>org.apache.tiles</groupId>
        <artifactId>tiles-core</artifactId>
        <version>3.0.3</version>
</dependency>
```

Figure 2.6: Dependency XML Structure for Maven

Maven also provided the archetype, or structure, for the application. It defined the folder structure for both the production and tests environments. It also sets up JUnit within the project to support unit testing throughout the development phase.

2.3.5 Database Model

The database framework used within this project was the open source framework, Hibernate. Hibernate is an Object/Relational Mapping [ORM] solution, and is concerned with relational databases, and more importantly for programmers, objects. Programmers generally "prefer to work with persistent data held (for the moment, anyway) in program objects, rather than use SQL directly for data access" (Bauer and King 2005).

Hibernate implements an Entity Data Model, and "sits between the object world of applications and the underlying database(s)" (Bauer and King 2005). Hibernate is derived from the Java Persistence API (JPA) and can be used in any environment that supports JPA, such as Java EE, Java SE and Enterprise applications.

It manages objects that need to be persisted, known as Entity Classes [EC], using annotations, which are detailed in subsequent sections. Each EC has a unique identifier "whose value is not important to the application apart from its use as an identifier" (Bauer and King 2005).

A rudimentary examination of Hibernate with Java Database Connectivity [JDBC] will be completed with regards towards the CRUD operations of each ORM database. This is examined within the Evaluation section of this report. CRUD is an acronym that relates to the four main operations that are performed on a database: Create, Read, Update and Delete.

2.3.6 Source Control

Source Control is the management of changes to the source code of an application. In this day and age, it is not unusual for a program to be worked on by a number of different people. In fact, it is more likely to be a globally dispersed team of programmers, so management of changes to the code base is very important. Essentially, it is

a system that "provides facilities for storing, updating and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change" (Rochkind 1975).

The source control system used within this project was GitHub, a free open source solution located at www.github.com. It provides integration with STS through the use of the *egit* plugin. egit is an open source plugin for Eclipse that allows for Git repositories to be imported in the IDE. It also allows actions, such as committing changes, to be performed within the IDE. GUI and Shell user interfaces also exist for a variety of operating systems. It was also used to manage the different versions of the report you are reading now. Within the scope of GitHub, each project is called a repository.

GitHub also provides graphs and statistics about each repository, such as which days are the busiest for commits as shown in Figure 2.7, the growth of the code base over time and many more.

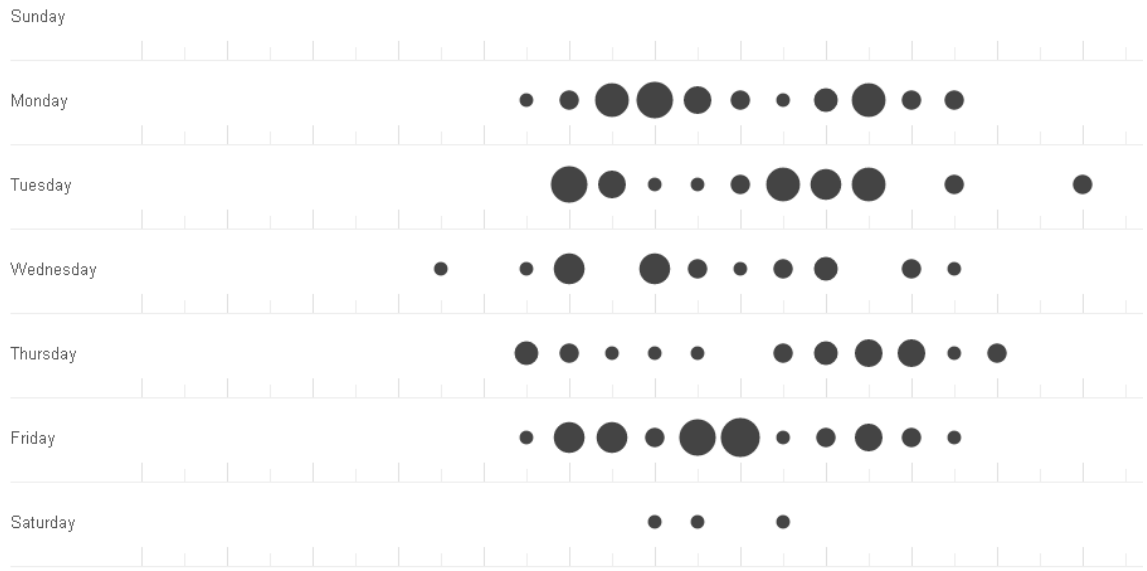

Figure 2.7: GitHub Visualisation of Commits/Day

2.3.7 Integrated Development Environment

The IDE used for this project was Spring Tool Suite [STS], a modified version of the open source IDE, Eclipse. The advantages of using STS over Eclipse are the pre configured services within the application. Tomcat, Maven, egit, and the core Spring dependencies themselves come pre-packaged within the application. Java EE and web application support are also present. One clear advantage of using STS over

Eclipse is that if a organisation were using Eclipse as as IDE, they could be running a variety of different versions of application servers or plug ins. A pre-packaged solution like STS reduces the risks of bugs being introduced, or not being able to reproduce bugs, on different development environments.

2.3.8 Logging

Logging was used within the application to check both the flow of the application, as well as to pin point where certain lines of code were being called. The logging implementation used was log4j, an open source logging solution created by Apache. Log4j is designed to enable or disable logging at run time. This is important in an application may have thousands of logging instances within it. This would be difficult to remove from production code, and would increase the risk of introducing bugs were it to be attempted.

Logging can also be used to analyse usability and a "log will contain statistics about the frequency with which each user has used each feature in the program and the frequency with which various events of interest (such as error messages) have occurred." (Holzinger 2005)

2.4 Usability Studies

The criteria that govern usability were looked at within this project. In doing this, two other websites were examined: the current website for the club, and a site that was recommended to me by a club member, Tralee Tennis club.

2.4.1 Method of Evaluation

The criteria for evaluating the websites in the case studies were adapted from Smith 2001. Smith breaks down the criteria into two groups: *Information Content Criteria* and *Ease-of-use Criteria*. While accuracy of content is a factor in usability, it is outside the remit of a final year project. *Ease-of-use Criteria* will be the sole group used for evaluation herein.

2.4.2 Case Study: Monaleen GAA Tennis Club

The first site examined was the existing site for the club. This site is located at *http://www.monaleengaatennisclub.com/*. The current site is a basic HTML site with a CSS style sheet, that occasionally uses a PHP script to facilitate users to register for tournaments. It uses a basic architecture stack consisting of these HTML pages coupled with an Apache HTTP Server. The reasons for this choice that the

club just needed a presence on the internet, there was no need for much functionality at the time, and simplicity.

This architectural choice places a number of constraints on the potential growth of the site however. The definition of roles within the system is not possible. An example would be that committee members cannot add news stories themselves, but must contain the web-master to perform this action on their behalf. There is also no scope to add secure features to the site, such as a members only feature. Any introduction of these features would require a considerable overhaul of the existing architecture.

The only NFR looked at during the development of the site was availability, which is fulfilled by the choice of architecture and lightweight implementation.

2.4.3 Case Study: Tralee Tennis Club

The second site chosen as a comparison was the site for the Tralee Tennis Club. This site was mentioned during requirements elicitation, as members from Monaleen had travelled there for a tournament. While there was no opportunity to speak with the site administrator, the site has the appearance of a professional design. The site is developed with PHP and JavaScript. It facilitates the booking of available timetable slots through a user login system. This system is available through the website, and a physical kiosk on location. The type of web architecture, and the information pertaining to why these choices were made, were not available.

The site is available at *http://www.traleetennisclub.ie/*.

2.5 Usability Results

These criteria are based on a Yes-No answer. A Yes results in 1 point, where a No results in 0. Any exceptions are noted within the relevant tables.

Criteria	**Monaleen Tennis Club**	**Tralee Tennis Club**
Links are updated?	No	Yes
Short-cuts for frequent users?	No	No
Warning if link leads to large file?	n/a	n/a
Indication of restricted access for link	n/a	Yes
Link text indicates nature of target	No	Yes
Total	0/5	3/5

Table 2.3: Table 1 - Links

Criteria	**Monaleen Tennis Club**	**Tralee Tennis Club**
Contact Details for website maintainer	Yes	Yes
Link to page maintainer on each page	Yes	Yes
Forms provided for users to enter details	Yes*	No
FAQ provided	No	Yes
Feedback mechanisms are fully operational	No	No
Total	2.5/5	3/5

Table 2.4: Table 2 - Feedback Mechanisms

**Only when tournament sign ups are open, .5 mark as it is not a regular part of the site*

Criteria	**Monaleen Tennis Club**	**Tralee Tennis Club**
Speed of site adequate	Yes	Yes
Site Availability High?*	Yes	Yes
Website made known through search tools**	.5/3	2.5/3
Link back to parent entity	Yes	Yes
Name of entity reflected in URL	Yes	Yes
URL is not over complex,likely to be mistyped	No	No
Total	4.5/8	6.5/8

Table 2.5: Table 3 - Accessability

**While not possible to test extensively over a period of time, this write was never denied access at any time*
***This was tested with 3 search engines: Google, Bing and Yahoo! Marked out of 3 - 1pt for first result, .5pt for first page, 0 for page 2+. Search terms were 'Limerick Tennis' and 'Kerry Tennis'*

Criteria	**Monaleen Tennis Club**	**Tralee Tennis Club**
Format,Graphic Design Appropriate	Yes	Yes
Pages appropriate length, clear, uncluttered	No	Yes
Consistent format through website	No	Yes
Site compatible with multiple browsers	Yes	Yes
Site can be used without graphics	No	No
Total	2/5	4/5

Table 2.6: Table 4 - Design

Criteria	**Monaleen Tennis Club**	**Tralee Tennis Club**
Website organised by anticipated user need	No	Yes
Navigation options are distinct	Yes	Yes
Conventional navigation models*	Yes	Yes
Navigation links are provided on all pages	Yes	Yes
Browsing facilitated by menu or site map	Yes	Yes
Reach any point in appropriate number of clicks**	Yes	Yes
Search Engine Provided	No	No
Total	5/7	6/7

Table 2.7: Table 5 - Navigability

**eg menu on top or left of screen*
***defined as 3,* Smith 2001

Site	Total
Monaleen Tennis Club	14/30
Tralee Tennis Club	22.5/30

Table 2.8: Table 6 - Results

Tralee Tennis Club is a clear winner with a site that adheres to the usability criteria defined by Smith. This site provides a benchmark in which to compare the application developed for this project.

Chapter 3

Requirements

3.1 Introduction

Requirements Elicitation is an important step in the development of a software application. There are a number of techniques possible, such as "interviewing, protocol analysis, repertory grid, work groups" (Davis et al. 2006). Structured interviews "appear to be one of the most effective elicitation techniques in a wide range of domains and situations" (Davis et al. 2006).

In eliciting possible requirements for the site, there was a discussion with three club members with varying backgrounds and experience within the club.

Name	Age Bracket	Club Role	Club Membership	Work Background
S1	35 - 45	Committee Member	5 years	Senior Software Engineer
S2	18 - 25	New Member	1 year	Graduate Software Engineer
S3	55+	Senior Member	10+ years	Retired Public Servant

Table 3.1: Stakeholders for Requirements Elicitation

3.2 Methodology - Grounded Theory

A methodology for analysing requirements that was examined in this project was the concept of Grounded Theory [GT]. Grounded Theory refers to theories that are "grounded in data systematically gathered and analyzed" (Strauss and Corbin 1994). There are a number of steps required for a GT process.

1. Data Collection

- Data collection is "collecting requirements ideas" (Berry et al. 2013). In this project, interviews were used as the primary method of collecting data for possible requirements.

2. Coding
    - Coding refers to the "classifying requirements into functional or non-functional" (Berry et al. 2013). It also involves ranking of requirements by such criteria as necessity, feasibility, cost etc.
3. Sampling
    - Sampling is asking stakeholders follow up questions about the "codings of the requirements ideas" (Berry et al. 2013).
4. Memoing
    - Memoing is the how the requirements are represented. These could be examples such as user stories, scenarios, a specification such as Volere.
5. Sorting
    - This ensures that the requirements are sorted according to a category. This can be application specific, such as Timetable, Tournaments, or a higher level like Functional and Non Functional requirements.
6. Writing up
    - This is the preparation of the final document that contains the final requirements specification. They type of documents will depend on the software life cycle process. An example of a template used to create a Requirements Specification would be Volere or ISO 9126.

Due to the time constraints present within a FYP, it was not possible to follow this process empirically. An abbreviated version of the process was followed with regard towards data collection, coding, sampling and sorting. The representation and creation of a requirements document were outside the scope of this project.

3.3 Methods for Requirements Elicitation

3.3.1 Storyboarding

At an early stage of the application, rough storyboards were prepared for the FYP presentation. These storyboards were used to demonstrate how a page, such as the timetable shown in Figure 3.1, would be displayed by the application.

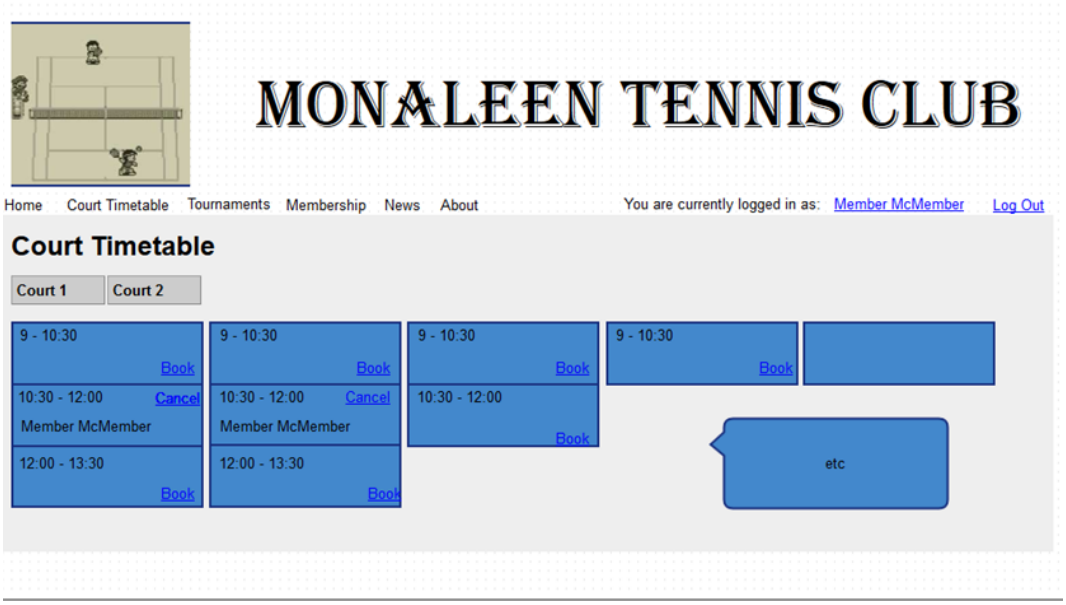

Figure 3.1: Timetable Storyboard, October 2013

The storyboarding visualised aspects of the site, and gave a rough idea of functionality that would be needed within the application. These were shown to interviewees in order to help them visualise requirements.

3.3.2 Interviews

In order to elicit requirements, a number of interviews were held with stakeholders. These questions are listed in A.3, Requirements Elicitation Questions.

During the elicitation process a number of areas were highlight as desired features.

1. Online Timetable
    - Allow members to view courts and make/view bookings
2. Tournaments
    - Allow user to register for a tournament, and view tournaments ongoing.
3. Contact Members
    - Easy way to contact all members
4. Member Directory
    - A list of all members, contact details, roles

5. News Section
    - Create new items to display for members
6. Members Area
    - A secure area that only members could access
7. Member Application
    - Automated registration, replace old paper form
8. Club Map
    - Directions to the club for new members and non-local visitors
9. Contact Details
    - Information on how to contact within the club for specific needs
10. Statistics
    - Such as games played, Win/Loss ratio

Table 3.2: Requested Features

Table 3.3 refers to each numbered requirement, and whether it was brought up by a stakeholder during the elicitation process.

Name	1	2	3	4	5	6	7	8	9	10
S1	N	Y	Y	Y	Y	Y	Y	Y	Y	N
S2	Y	Y	Y	N	N	N	N	N	N	Y
S3	Y	N	N	N	N	N	Y	Y	N	N
Total	2	2	2	1	1	1	2	2	1	1

Table 3.3: Requested Feature Breakdown

3.4 Application

This features were broken down into four categories: *Timetable*, *Tournaments*, *Members*, and *News*.

3.5 Functional Requirements

3.5.1 Timetable

The timetable is the core aspect of the application, and one that would be most likely to be used by all members, not just those involved competitively. While the regular member would only be concerned with the booking of slots, there are a number of requirements defined for use by the administrator in order to configure a relevant timetable not the club. The timetable needs to be flexible to allow the administrator full control at all stages.

1. Flexible
2. Edit individual slots
3. Define a template for a timetable
4. Reset timetable
5. Define look ahead for timetable (how many weeks in advance a user can see)
6. Delete timetable
7. Enable and disable timetable
8. Timetable analysis (No slots free, booked etc)

3.5.2 Tournament

3.5.3 News

3.5.4 Members

3.6 Use Cases

Use Case 1	**View All Members**
Scope:	System-wide
Level:	User can view a list of all registered, and approved, members of the club
Primary Actor:	All Registered and Authenticated Users

Stakeholders and Interests:	• All Users: contact information for club members
Preconditions:	User is registered and approved
Postconditions:	User must be authenticated by the framework

Main Success Scenario:

1. User logs in
2. User clicks on View Members
3. System displays member information

Extensions:

1.a Invalid login data:

1. System shows failure message
2. User returns to step 1

1.a User not approved:

1. System shows failure message
2. Admin is emailed about attempted access by unapproved member

Frequency of Occurrence:	High

3.7 Non Functional Requirements

3.7.1 Security

Since user details would be stored in the application, security of this data was highlighted as a concern. While no payment information would be held by the application, there would be names, addresses and phones numbers held within the application. The security of this information would need to be ensured.

3.7.2 Extensibility

Extensibility of the application was discussed with particular attention of the ability of the application to deal with changed to the club structure. Currently, the club has two courts in which users can book time for games. It is likely that the club will be expanding in the near future. This will result in the creation of four next courts in the near future. The application should be able to scale with this possibility without issue.

Tournaments were also mentioned as an area where future requirements may stem from. Currently, the club operates three kinds of tournaments: Singles, Doubles and Mixed Doubles. These are broken down into Ladder style tournaments and Bracket style tournaments. The club regularly holds tournaments with other national tennis clubs. The ability to create tournaments that suit these events could be necessary.

3.7.3 Usability

The age of the club members range from 8 to 80, so ease of use of any software solution is important. If a system is going to replace the existing system that all can use, a similar level of usability is required. The most common function used by members is the reservation of time slot in one of the courts. A software alternative needs to be intuitive for all members.

3.7.4 Performance

Chapter 4

Design

4.1 Introduction

4.2 Framework Design

Spring incorporates a number of design patterns within its own classes.

- Singleton
  - Beans defined within the Spring MVC framework and singletons by default.
- Factory
  - The Factory pattern is used for loading Beans through the *BeanFactory* class, and the *Application Context*.
- Inversion of Control
  - This pattern is central to the dependency injection facilities of the Spring framework

4.3 Controller-Service-DAO Design

In order to achieve modularity across the application, the core of the application is separated into four key areas.

- Entities
  - A Java object that is persisted to a table in a relational database.
- Controllers
  - A controller will handled the user requests the application must deal with
- Service
  - The Service layer facilitates communication between the controller and the DAO layer.
- Data Access Objects [DAO]
  - This layer is responsible for persisting an entity or any changes to an entity.

The flow between these objects is shown in Figure 4.1. An example flow would be the creation of a user objects. (Do I need specific code samples here, or is defining the flow enough (or necessary)?) The current work done is an example, not finished.

1. Controller present user with 'Home' view.
2. User clicks on 'Register' which sends a request to the 'Members' controller
   - ```
     1 @RequestMapping("/creatememberss")
     2 public String createMembers(Model model) {
     3 model.addAttribute("member", new User());
     4 return "creatememberss";
     5 }
     ```
3. Controller returns 'Create Members' View (Line 4)
   - The Service layer facilitates communication between the controller and the DAO layer.
4. Data Access Objects [DAO]
   - This layer is responsible for persisting an entity or any changes to an entity.
     ```

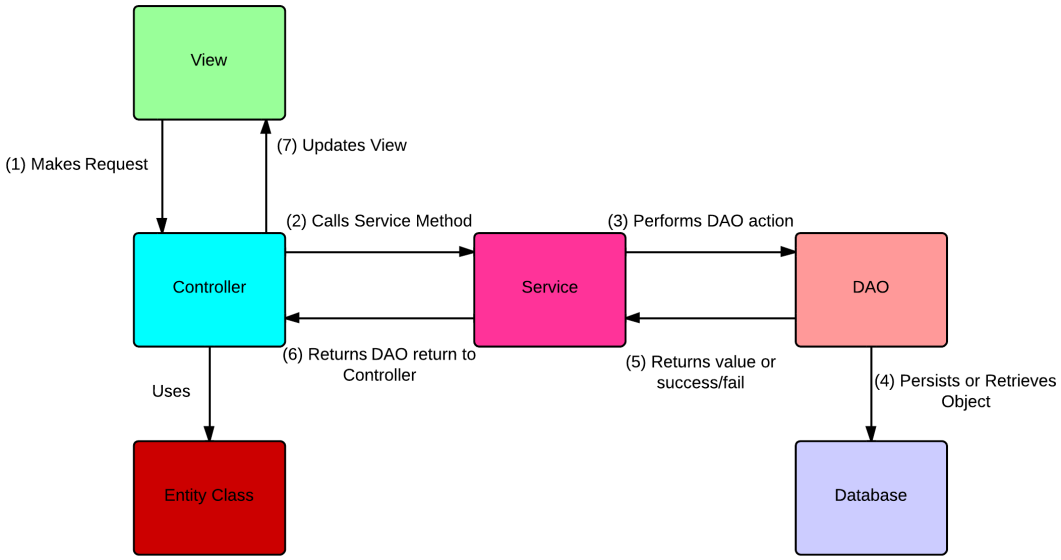

Figure 4.1: Controller-Service-DAO Flow

4.3.1 User Roles

The User class was designed with the existing application form of Monaleen Tennis Club as a foundation. Spring controls access within the application. Firstly, it uses an *authority* hierarchy to separate different levels of users. For this web application, there were three main levels of authority, with one level containing three different branches.

Roles

- ROLE ADMIN
  - This refers to the main administration group. The group retains full rights across the web application
- ROLE COMMITTEE
  - This refers to the committee, as defined by the club themselves. This group with have the ability to perform some administrator privileges, but only those directly related to club activities, not site activities.
- ROLE MEMBER
  - The default user state. This group can perform actions such as booking slots in a timetable, registering for a tournament, and will have access to parts of the site unavailable to non-registered users.

- ROLE WARNING
  - A restriction placed upon a member. For example, a member who books time slots, but does not attend. In this application, it reduces the number of allowed bookings on a court per week from 3 to 2.
- ROLE SUSPEND
  - A further restriction placed upon a member. Number of bookings on a court per week reduced to 1.

Chapter 5

Implementation and Testing

5.1 Introduction

This chapters deals with the implementation of the application, with the focus on the application entities, and how they were configured.

5.2 Application Entities

5.2.1 Users - Persistence

The *User* class represents every user account within the application. (Link to appendices showing class attributes). This section will focus on a regular user, how it is configured within the application in terms of bean definition and persistence.

Firstly, as shown in line one of Figure 5.1, the class needs to be configured as a *Component* for the application. This ensures that the Spring framework considers the User class as one for auto-detection, through the use of class path scanning and annotations prevalent within this application. The framework instantiates this bean, or object, automatically, without the developer having to use the *new* keyword.

```
1  @Component
2  @Entity
3  @Table(name="users")
4  public class User {
5          @Id
6          @GeneratedValue
7          int id;
8
9          @NotNull(groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
10         @Pattern(regexp=".+\\@.+\\..+", message="This does not appear to be
               a valid email address",
               groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
11         @Column(name="username")
12         String username;
13
14         @Size(min=5, max=45, message="Named must be between 5 and 45
               characters",groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
15         @Column(name="name")
16         String name;
17
18         @Column(name="password")
19         @Size(min=5, max=15, message="Password must be between 5 and 15
               characters", groups=FormValidationGroup.class)
20         String password;
21
22         @Column(name="gender")
23         String gender;
24
25         @Pattern(regexp="08[35679]([0-9]{7})", message="Number must be in
               the format 083, 085, 086, 087, 089 and 7 additional numbers eg
               0851234567", groups={PersistenceValidationGroup.class,
               FormValidationGroup.class})
26         @Column(name="contact_num")
27         String contact_num;
28         //Class truncated. Some repetitive attributes omitted
29         //Getters and Setters below here.
```

Figure 5.1: User Class Definition and Configuration

The *Entity* and *Table* annotations of lines 2 and 3 respectively belong to the javax.persistance package. These annotations are used by Hibernate in order to manage and persist the class. The *@Table* annotation has a 'name' attribute that

refers to the schema table the class maps to. There are two ways that an attribute can be assigned to a table column by Hibernate. Both methods are shown in Figure 5.1. An annotation may be placed on the attribute in order to specify a column name. Line 12 in Figure 5.1 shows the username attribute being mapped to the username column within the User database schema. The other way of specifying where an attribute should be persisted is to ensure that the attribute name matches the column name within the table. This implicitly allows Hibernate to map a class, without having to explicitly define the mapping for the persistence framework.

The User class has a number of attribute constraints placed upon it. There are two types of constraints within this application: *FormValidationGroup* and *Persistence-ValidationGroup*. These are interface classes with no attributes that serve as identifiers. As shown in lines 10, 14, 19 and 25 of Figure 5.1, an attribute may be constrained by one or more groups. An annotation, from the javax.validation.constraints, is applied to the attribute. The annotations used within this application were as detailed in Table 5.1.

Constraint Name	Description
NotNull [Line 9]	Ensures the value within the attribute does not have a null value
Pattern [Line 10]	Ensures the value within the attribute conforms to a regular expression
@Size - Min [Line 14]	Ensures the value within the attribute has a minimum length
@Size - Max [Line 14]	Ensures the value within the attribute has a maximum length

Table 5.1: Class Constraints

These validation package interfaces provide a *groups* attribute, which is an array of objects. The *FormValidationGroup* and *PersistenceValidationGroup* are passed to this attribute. These groups allow attributes to have different constraints at different stages in the application. When using this attribute within the application, such as the creation of a user within a form, the Controller classes apply the validation to the user input and persisted data. The reason for having two groups of validation within this application is due to security. In every application, it is advisable to perform encryption on sensitive data, such as passwords. Within the scope of this application, user passwords were defined as being between 5 and 15 characters long, with no restriction to the content of the password. This application flow on taking in user input and persisting it is demonstrated below in Figure **??**.

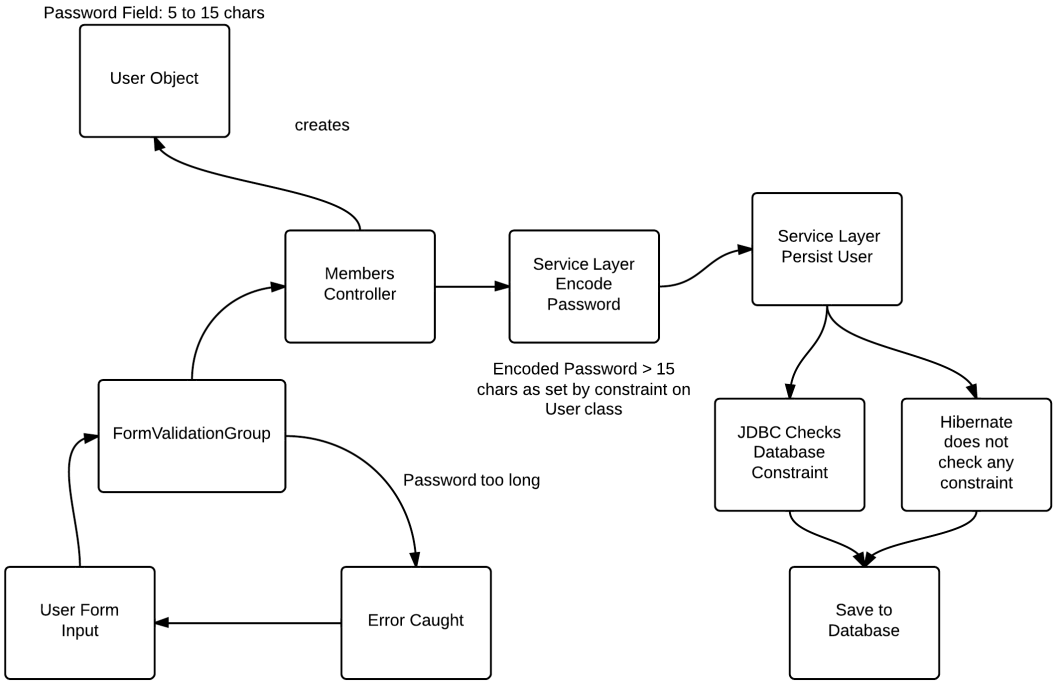

Figure 5.2: DispatcherServlet

For example, a user password with 8 characters would pass form validation with no issues. When the PasswordEncoder bean is applied to the string prior to persistence, it will result in a value like *'acb172137243c0b931321d7645dc31c2efb8346385ae0547d11b1d8de333215b'*, a value much longer than 15 characters. This will cause a failure with Hibernate persistence. This is because Hibernate works at a class level, and does take note of the constraints placed upon the class, while JDBC does not. The constraints that JDBC takes note of are those taken directly from the database itself. In this application, passing form validation is sufficient, as there are security annotations placed on the Service classes that manage data persistence. An example of how the controller handles validation is shown later in Figure 5.6.

Alongside class configuration, it is also necessary to configure Hibernate to scan the packages that contain entities, as detailed in Figure 5.3. This is done through the creation of a sessionFactory bean, which uses the AnnotationSessionFactoryBean class. This bean is responsible for the creation of session instances within the application, though each application usually only has one session. It is an immutable object, and cannot be changed once it is created, so proper configuration of classes to facilitate object-relational mapping is important. The Session object created by

the factory is responsible for creating the connection between the application and the database.

```
1  <bean id="sessionFactory"
2  class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
3  <property name="dataSource" ref="dataSource"></property>
4  <property name="hibernateProperties">
5          <props>
6                  <prop
                       key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
7          </props>
8  </property>
9  <property name="packagesToScan">
10         <list>
11                 <value>users</value>
12                 <!-- user refers to a package that contains user related
                       classes --!>
13         </list>
14 </property>
15 </bean>
```

Figure 5.3: Hibernate SessionFactory Configuration

The HibernateProperties property defined in Line 4 refers to the SQL dialect used by Hibernate. This dialect provides the functionality to use human readable expressions to define query statements. Hibernate does not use SQL to structure these statement. It uses a variant called Hibernate Query Language [HQL]. HQL syntax is based has its basis in object oriented programming. The class itself defines the table, not the syntax. The different in implementation is shown in Figure 5.4.

```
1  SQL [JDBC]: jdbc.query("select * from users", new UserRowMapper());
2  // UserRowMapper is a class that maps rows to objects
3
4  HQL [Hibernate]: session().createQuery("from User").list;
```

Figure 5.4: HQL-SQL Comparison

The Form Validation is provided by the Spring Security file, which will be examined in detail in the Administration Implementation section on page 52. As discussed previously, there are a number of validation constraints placed on the *User* class. Spring provides a facility to ensure these constraints are enforced, and to also provide a positive user experience. It does this through the use of a BindingResult object.

This object holds a record of any errors from the form that the user populates. The controller that deals with the form will check the BindingResult object for errors, and can respond appropriately. In order for this to work, both the Controller and the form, as depicted in Figures 5.5 and 5.6, need to be defined clearly. The form needs to be created using the Spring Framework form tag library, and errors needs to be specified for each input within the form.

```
1 <!-- Excerpt from the User registration form. Formatting removed for
      clarity --!>
2 <sf:form id="details" method="post"
      action="${pageContext.request.contextPath}/register"
      commandName="member">
3 Name <sf:input name = "name" path="name" type="text"/>
4 <sf:errors path="name" cssClass="error"></sf:errors>
5 Password <sf:input id="password" name = "password" path="password"
      type="password"/>
6 <sf:errors path="password" cssClass="error"></sf:errors>
7 </sf:form>
```

Figure 5.5: User Registration Form

The form structure, illustrated partially in Figure 5.5, contains a number of important attributes. The form used is an extended version of the HTML ¡form¿ object. The Spring MVC version adds extra functionality such as mapping to objects within the controller classes and error validation checking. The *commandName* is a variable within the form that contains the information within the form. This temporarily persists data between the form and the controller. This benefits the user as the application repopulates valid fields in the form should an error be made. The *sf:error* tags allow the controller to identify specific errors within the form. These also reference attributes within the class the form is linked with. In this example, the *User* class is being created with this form.

Figure **??** Line 4 shows the declaration of a BindingResult object. This object holds the results of the form validation. Line 5 declares that if the result object has any errors, the controller returns to the form, where the *sf:error* tags will highlight the error message relation to any attributes that broke their constraints. Lines 9 through 13 show home the application ensures that the primary key remains unique.

```
1  //Method from the MembersController class
2  //This method is responsible for validating the form that users complete
       to register.
3  @RequestMapping(value = "/register", method = RequestMethod.POST)
4  public String doRegister(Model model,
5  @Validated(FormValidationGroup.class) @ModelAttribute("member") User
       member, BindingResult result) {
6  if (result.hasErrors()) {
7          return "createmembers"; // if the result has errors, go back to
               create page
8  }
9  if (userService.exists(member.getUsername())) {
10         result.rejectValue("username", "Duplicate Key",
11         "This email address has already been used");
12         return "createmembers";
13         //if the email address already exists, return with this message.
14 }
15         else {
16                try {
17                       member.setAuthority("ROLE_MEMBER");
18                       userService.create(member);
19                       return "registerSuccess";
20                       //successful creation of member
21                       } catch (Exception e) {
22                              return "error";
23                       }
24                }
25 }
```

Figure 5.6: User Registration Controller

Line 13 in Figure 5.6 shows the use of a *ModelAttribute*. A *ModelAttriute* is a property of the model that is supplied by Spring MVC to the controller. This object is created using the data mapped from the preceding form. This object is being validated using the *FormValidationGroup* as specified using the *Validated* annotation.

Within this application, the Service layer is responsible for the Controller communicating with the DAO layer to persist objects like the User class. Since Hibernate is configured at a class level, in relation to attributes and column names, there is no need for any INSERT or UPDATE statements. The current session, see Figure 5.7 is returned to the DAO object via the configured bean, and the necessary methods, such as save() and delete(), are called upon it. An object must be passed to the *save()* method of the current sessionFactory object, detailed in 5.3.

```
1 public Session session(){
2         logger.info("Session Factory returning current session.....");
3         return sessionFactory.getCurrentSession();
4 }
```

Figure 5.7: UserDAO getSession()

In the case of the User object, the password needs to be encoded prior to the object being persisted by the session(). In order to encode password, a bean responsible for the encoding must be defined within the application context, illustrated in Figure 5.8. Spring provides a class that allows passwords to be encoded, and the bean for this class is defined within a security-context file.

```
1 <bean id="passwordEncoder"
2 class="org.springframework.security.crypto.password.StandardPasswordEncoder">
3 </bean>
```

Figure 5.8: Password Encoder Definition

This Spring defined class provides an implementation for encoding data using SHA-256 hashing with 1024 iterations, with a random 8 byte salt value. This object then calls *encode()* on the value passed from the form filled by the user. As a result, the actual password is never stored in the database, just an encrypted form of it, as shown in Figure 5.9. Once the password is encoded (Line 4, 5.9), the session() object can save the object. Due to the class configuration, there is no need to specify any database schema information within the DAO classes.

```
1 @Transactional
2 public void createUser(User user) {
3         user.setPassword(passwordEncoder.encode(user.getPassword()));
4         session().save(user);
5 }
```

Figure 5.9: Persisting User object with encoded password

5.2.2 Timetable - Persistence Changes

One of the most difficult features to implement within the application was the Timetable. While the goal was to create a timetable suitable for Monaleen Tennis Club, it was desirable that the timetable retain the portability and not rely on hard

coded attributes. One issue this raised was how to handle a varying number of slots in each day. If a user could define 10 slots a day, how would be store this in such a way that a user could also define a timetable with 20 slots a day? Hibernate was able to facilitate this design with considerably less input from a developer.

The solution implemented in this application was to use List objects to store the values for each day. There would be seven lists in the Timetable class, one for each day, as per Figure 5.10.

```
1  @Entity
2  @Component
3  @Table(name = "timetable")
4  public class MonaleenTTV1 implements Timetable {
5          @Id
6          @GeneratedValue
7          private int id;
8          /**
9          * Other Attributes Here
10         **/
11
12         @ElementCollection
13         @CollectionTable (name = "monday",
               joinColumns=@JoinColumn(name="id"))
14         private List<String> monday;
15
16         @ElementCollection
17         @CollectionTable (name = "tuesday",
               joinColumns=@JoinColumn(name="id"))
18         private List<String> tuesday;
19
20         @ElementCollection
21         @CollectionTable (name = "wednesday",
               joinColumns=@JoinColumn(name="id"))
22         private List<String> wednesday;
23
24         @ElementCollection
25         @CollectionTable (name = "thursday",
               joinColumns=@JoinColumn(name="id"))
26         private List<String> thursday;
27
28         @ElementCollection
29         @CollectionTable (name = "friday",
               joinColumns=@JoinColumn(name="id"))
30         private List<String> friday;
31
32         @ElementCollection
```

```
33          @CollectionTable (name = "saturday",
                joinColumns=@JoinColumn(name="id"))
34          private List<String> saturday;
35
36          @ElementCollection
37          @CollectionTable (name = "sunday",
                joinColumns=@JoinColumn(name="id"))
38          private List<String> sunday;
39
40          //getters and setters
41  }
```

Figure 5.10: Timetable Class List Configuration

This results in the Timetable object being made up of 8 database tables. The core of the class is stored in the 'Timetable' database table. This table contains the primary keys and all other attributes, such as name, number of slots, timetable series. The seven other tables each represent a collection within the Timetable class. Each of these tables has a non unique, foreign key that ties it back to the central table. This is illustrated in Figure 5.10 with use of the *CollectionTable* annotation. This annotation defines the table that each collection belongs to. It also specifies a column for the HQL and SQL JOIN statement in the *JoinColumn* annotation. For example, if we were to create a Timetable with 10 slots, with the primary key being 1, each of the collection tables would have 10 entries. Each of these entries would have an id of 1, to match the primary key in the core table, plus a value, such as 'Free Court'. The position in the table corresponds to the position within the collection class.

The following example, in Figure 5.2, shows a timetable configured with two slots. Upon creation, a series is created for each timetable. Each series contains 52 timetables, one for each week. The currently displayed timetable is determined by its position in the series in relation to the current week of the year, as shown by the Java Calendar class. The 'preview' attribute determines how far ahead the application will allow the user to browse in a timetable series. By default, the application displays the current weeks timetable, and does not allow a user to go backwards. If the current week, as defined by the Java Calendar class, is 13, the user will be able to see weeks 14, 15 and 16, owing to the preview with a value of 3. In Table 5.2, the next three weeks of timetables are available for viewing and booking. This allows the administrator to dynamically restrict how far in advance users can book slots.

pid	monday
1	Free Court
1	Free Court
2	Tournament
2	Free Court
3	User Booking
3	Free Court
4	Training
4	Free Court

Table 5.3: Timetable Storage Table for Monday Collection

pid	name	slots	startTime	endTime	enabled	preview	series	total
1	Court One Week 1	2	8	22	1	3	1	52
2	Court One Week 2	2	8	22	0	3	1	52
3	Court One Week 3	2	8	22	0	3	1	52
4	Court One Week 4	2	8	22	0	3	1	52

Table 5.2: Timetable Core Database Table

While the table structure of the *User* and *Timetable* differ considerably, since Hibernate deals with objects, that have table structures defined within their classes, there is no change to the way the Timetable is saved and updated, as per Figure **??**. Using JDBC, there would have been 8 SQL queries to save to each of the 8 table, increasing the risk of bugs within the application, or invalid data being saved or retrieved.

```
1 @Transactional
2 public void createTimetable(Timetable t) {
3         session().save(t);
4 }
```

Figure 5.11: Hibernate Save Timetable

The timetable is displayed in the application as shown in Figure 5.12.

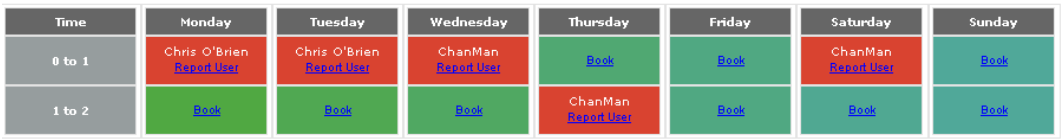

Figure 5.12: Timetable display within application

5.2.3 Tournaments and Timetable Events

The third major area of the site were the *Tournament* and *Event* classes. The structure and configuration of these classes are not different from the *User* or *Timetable* classes. In the context of this report and application, an Event is any object that can populate a field within a Timetable class.

Events

Within the scope of this application, an *Event* is any item that can be scheduled within the Timetable object. As with the *Tournament* class, the *Event* class is configured much like the *User* and *Timetable* classes. One event that is required by all versions of the Timetable is the 'Free Court' event. This event is created by default (see Figure 5.13) either the first time an administrator attempts to create an Event, or to create a Timetable. An Event has two attributes: a name and an author. In the case of a system event, the name will be the login of the person who booked the timetable slot, while the author is defined as *BOOKING SYSTEM*. For an administrator event, the name is defined by that administrator, while the author is listed as the administrator who created the event. Example events for Monaleen Tennis would be:

- Training Sessions
- Club Social Events
- Ladies Tennis
- Recurring Tournaments
- General Club Activities

```
1 @RequestMapping(value = "/saveEvent", method = RequestMethod.POST)
2 public String saveEvent(Model model, @ModelAttribute("event") Event e,
3         BindingResult result) {
4         if (eventService.getEventById(1).equals(null)){
```

```
5                   e.setName("Free Court");
6                   e.setAuthor(userService.emailToName(SecurityContextHolder
7                   .getContext().getAuthentication().getName()));
8                   e.setEnabled(true);
9                   eventService.createEvent(e); // see Line 29-32 for creation
                        of Event.
10          }
11          logger.info("Event Save Method....");
12          if (result.hasErrors()) {
13                  return "createEvent";
14          }
15          if (eventService.exists(e.getName())) {
16                  result.rejectValue("name", "Duplicate Key",
17                  "This Event Name has already been used.");
18                  return "createEvent";
19          } else {
20                  e.setAuthor(userService.emailToName(SecurityContextHolder
21                  .getContext().getAuthentication().getName()));
22                  e.setEnabled(false);
23                  eventService.createEvent(e);
24                  return viewEvent(model);
25          }
26  }
27
28  //Event DAO
29  public void createNewEvent(I_Event e){
30          logger.info("Creating new event....");
31          session().save(e);
32  }
```

Figure 5.13: Event Creation

Lines 4 through 10 are necessary to ensure no NullPointerExceptions are introduced when creating a timetable. If no events exist within the system either the first time a timetable is created or, as illustrated above in Figure 5.13, an event is created, the system will create a *Free Court* event which is needed for the timetable to function correctly.

In order for an event to show up as an option when creating, or editing, a timetable, it must be enabled, as shown in Figure 5.14. This implementation allows the administrator to pre-create events for later use without overcrowding the timetable.

```
1  @RequestMapping("/changeEventStatus")
2  public String changeEventStatus(Model model, HttpServletRequest request) {
```

```
        Event e = (Event) eventService.getEventById(Integer.valueOf(request
        .getParameter("eventID")));
        if (e.isEnabled()) {
                e.setEnabled(false);
                eventService.updateEvent(e);
                return viewEvent(model);
        } else {
                e.setEnabled(true);
                eventService.updateEvent(e);
                return viewEvent(model);
        }
}
```

Figure 5.14: Change Event Status

Tournament

The *Tournament* class is configured similar to the *Timetable* class, in that it has a secondary table which contains a list of members who have registered for the tournament. The key area for tournament implementation was for the application to differentiate between the various states of the tournament, such as registration being enabled, yet the tournament not having started yet. It was also important that the tournament could link up with the Timetable system, in order to book slots to play the tournament. In this regards, a tournament was required to manage an Event object, see Figure 5.15, and to ensure that no ambiguity was present when dealing with multiple tournaments and possible multiple timetables. It was also necessary for a Tournament object to manage its own event in order to minimise both the risk of ambiguity and administration involvement.

```
@RequestMapping(value = "/registerTournament", method = RequestMethod.POST)
public String doCreateTournament(
        Model model, BindingResult result,
        @Validated(FormValidationGroup.class) @ModelAttribute("tournament")
            Tournament t) {

        if (result.hasErrors()) {
                return "createTournament";
        }
        if (tournamentService.exists(t.getTournamentName())){
                if (eventService.exists(t.getTournamentName())){
                result.rejectValue("tournamentName", "Duplicate Key",
                "An event of this name already exists");
                return "createTournament";
```

```
        }
        result.rejectValue("tournamentName", "Duplicate Key",
                "A tournament of this name already exists");
                return "createTournament";
                }
        else {
                try {
                        tournamentService.create(t);
                        eventCreation(t);
                        logger.info("Tournament Created");
                        return "tournamentSuccess";
                } catch (Exception e) {
                return "error";
        }
}
public void eventCreation(Tournament t){
        Event e = new Event();
        e.setName(t.getTournamentName());
        e.setAuthor(userService.emailToName(SecurityContextHolder.getContext()
        .getAuthentication().getName()));
        eventService.createEvent(e);
}
```

Figure 5.15: Tournament Creation

Deleting a tournament needed similar logic, as displayed in Figure 5.16. Similar to the *User* class, the Tournament name, much like the username, is used as a primary key, and can be recycled once a tournament is deleted.

```
        @RequestMapping("/confirmDelete")
        public String deleteTournament(Model model, HttpServletRequest
            request){
                Tournament t = tournamentService.getTournamentById(request
                                .getParameter("tournamentID"));
                tournamentService.deleteTournament(t);
                eventService.deleteEvent(eventService.getEventIdByName(t.getTournamentName()));
                model.addAttribute("tour",
                    tournamentService.getAllTournaments());
                return "deleteTournament";
        }
```

Figure 5.16: Delete Tournament

Due to constraints on time within the final year project, it wasn't possible to fully flesh out this area of the site, though it was implemented in such a way to allow for extensibility such as an interface for the sorting of registered members into teams. This interface would allow the specification of business logic to allow the system to, for example, sort tournament teams by experience and ranking.

5.2.4 Administration - Security and Session Management

The *Administration* section of the application deals with the implementation, and configuration, of the security and session management aspects of the application. An administrator has the same structure as a *User* and is defined by the *Role* it has, as seen previously in Figure 4.3.1. These roles are configured through Spring Security, in which a specific database schema must be adhered to. By default, there should be two tables: *users* and *authorities*, with a foreign key constraint. In this application, as shown in Line 3, Figure 5.17, this was modified to keep the user data within the same table.

```
1 <security:authentication-provider>
2 <security:jdbc-user-service data-source-ref="dataSource"
3 id="jdbcUserService" authorities-by-username-query="select username,
      authority from users where binary username = ?" />
4 <security:password-encoder
      ref="passwordEncoder"></security:password-encoder>
5 </security:authentication-provider>
```

Figure 5.17: Spring Role Configuration

The security within the application is controlled by the *security-context.xml* file, which used Expression Based Access Control [EBAC] in order to restrict site access to relevant roles. EBAC allows complicated boolean logic to be encapsulated within a single expression, such as whether a user is authenticated or not. The base class used within this application is *SecurityExpressionRoot*. These expressions are contained within the *security-context.xml* file, as seen in Figure 5.18. This configuration file is partly responsible for controller access to the mappings within the application.

```
1 <security:http use-expressions="true">
2 <security:intercept-url pattern="/static/**" access="permitAll" />
3 <security:intercept-url pattern="/images/**" access="permitAll" />
4 <security:intercept-url pattern="/createmembers" access="permitAll" />
5 <security:intercept-url pattern="/tournamentSuccess"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_COMMITTEE')" />
```

```
6 <security:intercept-url pattern="/createNews"
      access="hasAnyRole('ROLE_ADMIN', 'ROLE_COMMITTEE')" />
7 <security:intercept-url pattern="/members" access="isAuthenticated()" />
8 </security:http>
```

Figure 5.18: Security Context File Excerpt

An issue that arises with this implement ion is that all resources, such as images, would automatically be blocked from appearing on the site unless explicitly defined. In the case of images, this would be very time consuming, especially when new images are added to the application frequently. In order to overcome this, Lines 2 and 3 are implemented. This syntax specifies that all files and folders within the *static* and *images* folder are to be granted access to all pages within the application. Site access can also be granted on an authentication basis, rather than a role based system, and this is implemented as show in Line 7 of Figure 5.18.

In addition to the expression level security, service-level security annotations are also in place. This gives the application an extra layer of security, as it restricts the use of methods within the service layer to defined roles. It is possible to use either expression level security or service level security independently. This is configured within the *security-context.xml* file, Figure 5.19, and is implemented, as shown in Figure 5.20, within the service layer. These annotation use the same roles as defined previously within the application.

```
1 <security:global-method-security
      secured-annotations="enabled"></security:global-method-security>
```

Figure 5.19: Security Context Service Layer Annotation Configuration

```
1  @Service("timetableService")
2  public class TimetableService {
3
4  private TimetableDAO timetableDAO;
5
6          @Autowired
7          public void setTimetableDAO(TimetableDAO timetableDAO) {
8                  this.timetableDAO = timetableDAO;
9          }
10
11         @Secured("ROLE_ADMIN")
12         public void create(Timetable t){
```

```
13                 timetableDAO.createTimetable(t);
14         }
15
16         @Secured({"ROLE_ADMIN", "ROLE_MEMBER", "ROLE_COMMITTEE",
               "ROLE_WARNING", "ROLE_SUSPEND"})
17         public void update(Timetable t){
18                 timetableDAO.updateTimetable(t);
19         }
20
21         public List<Timetable> getAllTimetables(){
22                 return timetableDAO.listTimetables();
23         }
24 }
```

Figure 5.20: Security Context Service Layer Annotation Implementation

In the implementation shown above, there are some items to note: the *Service* and *Autowired* annotations. The *Service* annotation allows implementation classes to be auto-detected through class-path scanning. The package that contains the Service classes is defined within another XML file, *service-context.xml* within this application, Figure 5.21. This file configures Spring to look for Service classes via annotations, and where to look for them.

```
1 <context:annotation-config></context:annotation-config>
2 <context:component-scan base-package="service"></context:component-scan>
```

Figure 5.21: Service Context Configuration

The *Autowired* annotation is used within the framework to mark a constructor or setter. Spring then passes the needed dependencies into the application with its dependency injection facilities.

5.3 Model View Controller

5.3.1 Controller Layer

A Controller is a class within the application that creates and modifies a model, and passes it onto a View. A View will also pass information to a Controller, which will communicate with the Service layer to persist any relevant data within the application.

A Controller is annotated, as illustrated in Figure 5.22, Line 1. This allows the DispatcherServlet to identify the controllers and inject them into the application. This behaviour is displayed in Figure 5.23, which shows the DispatcherServlet being configured to scan the controllers package looking for annotations.

```
1 @Controller
2 public class TimetableController {
3         //.........
4 }
```

Figure 5.22: Service Context Configuration

```
1 <context:component-scan base-package="controllers,
      email"></context:component-scan>
2 <mvc:annotation-driven></mvc:annotation-driven>
```

Figure 5.23: Service Context Configuration

In order for the application to choose the correct method within a specific controller, it is necessary to define a mapping. The *RequestMapping* annotation is used for this purpose. This annotation maps a web request onto a handler method within a controller, as depicted in Figure 5.24. It is also important, within the scope of the Spring Security configuration, to ensure that this mapping has access rights defined within the application or access will be denied.

```
1 @RequestMapping("/")
2 public String showHome(Model model) {
3         logger.info("Showing Home Page....");
4         News news = newsService.getLatestStory();
5         model.addAttribute("newsHeader", news.getSummary());
6         model.addAttribute("newsContent", news.getContent());
7         return "index";
8 }
```

Figure 5.24: Request Mapping

This mapping responds to a request with the value '/', Line 1. On Tomcat servers, this the home page, as the URL will be *localhost:8080/¡application-name¿/*. In this

example, the latest news story is retrieved on Line 4, and values from it are added to the model displayed on the returned view.

Any value specified after *localhost:8080/¡application-name¿/* will search for a mapping within all controllers in the application. If a mapping does not exist, a resource not found will be displayed. This is default behaeviour for a denied access attempt. In order to ensure the site can deal with invalid requests, an access denied handler can be defined within the Spring Security context file, as depicted in Figure 5.26. This allows a mapping to be specified for invalid requests,

```
1 <security:access-denied-handler error-page="/denied" />
```

Figure 5.25: Access denied Handler

In this example, the *denied* mapping, shown in Figure **??** is called, and the corresponding view displayed.

```
1 @RequestMapping("/denied")
2 public String showDeny(Model model){
3        model.addAttribute("message", "The page you are looking for either
             does not exist, or you do not have access. Please contact an
             administrator if you believe this is an error.");
4 return "error";
5 }
6
7 //Error Page
8
9 <center>${message}</center>
```

Figure 5.26: Error Page Mapping

Line 3 shows a message being added to a model, which is added to a standard error page. This message will change depending on what error is being caught by the application. The message is then displayed by the 'error' View for the user, and the site appearance is not disrupted by a generic error page.

5.3.2 Models within the Spring MVC Framework

A model is a map containing a key-value pair that can be modified by a Controller and displayed by a View. A model is implicitly created by the framework by including a reference to it within the method signature in a controller, as illustrated in

Figure 5.27. The below example adds two lists to a model: a list of disabled events and a list of enabled events, shown in Lines 5 and 6. The Controller then returns the value *viewEvents* to the ViewResolver. The ViewResolver displays the relevant page that displays the information held within the model for the user, as depicted in Figure 5.28

```
1  @RequestMapping("/viewEvents")
2  public String viewEvent(Model model) {
3         List<Event> eventsDisabled = eventService.listDisabledEvents();
4         List<Event> eventsEnabled = eventService.listEnabledEvents();
5         model.addAttribute("eventsEnabled", eventsEnabled);
6         model.addAttribute("eventsDisabled", eventsDisabled);
7     return "viewEvents";
8         }
```

Figure 5.27: Model Definition

```
1  <c:if test="${not empty eventsEnabled }">
2  <table class="members" align="center">
3  <tr><th>Enabled Events</th></tr>
4  <tr>
5  <th>ID</th><th>Name</th><th>Author</th><th>Action</th></tr>
6  <c:forEach var="row" items="${eventsEnabled}">
7  <sf:form method="post"
8  action="${pageContext.request.contextPath}/changeEventStatus"
9  commandName="eventsEnabled">
10 <tr>
11        <td><input type="hidden" value="${row.id}" name="eventID"
              />${row.id}</td>
12        <td>${row.name}</td>
13        <td>${row.author}</td>
14        <td><input value="Disable" type="submit" name="${row.id}" />
15 </sf:form>
16 </c:forEach>
17 </table>
18 <center><font color="red">${message }</font></center>
19 </c:if>
```

Figure 5.28: JSP View of model

The page contains a conditional loop, Line 1, that checks if the model attribute, eventsEnabled, is empty. In the event that it is not, it will create a table to display

the information to the user, as shown in Lines 2 through 17. The *c:forEach* tag is the JSTL version of a for loop, which will be looked at in a later section. Line 18 is a message, that will display if the value is not null. This is used to display a confirmation or error message to the user when an action is performed on an event. An example would be an attempt to disable the *Free Court* event. This will display a message, shown below in Figure 5.29, that the action cannot be performed.

```
1 if (e.getName().equalsIgnoreCase("Free Court")){
2         model.addAttribute("message", "You cannot modify the Free Court
              event");
3         return viewEvent(model);
4 }
```

Figure 5.29: User message added to model

5.3.3 View Layer

Apache Tiles Configuration and Implementation

Apache Tiles is configured within the web application core XML file. There are two classes that the configuration is concerned with: TilesViewResolver and TileConfigurer. Both are declared as beans within the configuration file and automatically created when the application is launched. The primary function of the ViewResolver is to take in a String value, and return the relevant *RequestMapping* value within the application. These mappings are defined within the Controller classes of the application. The TilesConfigurer object, see Figure 5.30, takes one parameter: a location of the template that the default tile will use. The default tile will then be used by other pages as a template.

```
<bean id="tilesViewResolver"
       class="org.springframework.web.servlet.view.tiles2.TilesViewResolver">
</bean>

<bean id="tilesConfig"
       class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
       <property name="definitions">
       <list>
              <value>/WEB-INF/layout/default.xml</value>
       </list>
       </property>
</bean>
```

Figure 5.30: Apache Tiles Configuration

The default tile consists of a number of sections identified by a specific tag. These tags correspond to values within the tile layout configuration file. Using a version of inheritance, these can be overwritten and replaced with other pages in order to change the content of a page, while maintaining cohesion across the design of the application.

The following examples shows the implementation within the configuration file. The first section of code is the overall template. This specifies the default values that make up a JSP page within the application. The second segment of code is the the definition for the initial home page for the web application. By the inclusion of the *extends="users.base"* within the definition tags, it is defining the index as a sub class of the users.base definition. Consequently, it is possible to override any of the attributes within the users.base definition. In this example, the title and content of the default page are being overridden with different values in order construct a more suitable page. The header, links and footer however remain the same, and will do so will all pages following this format, as shown in Figure 5.31.

```
<definition name="users.base" template="/WEB-INF/templates/default.jsp">
        <put-attribute name="title" value="Monaleen Tennis Club - Default
            Template"></put-attribute>
        <put-attribute name="header"
            value="/WEB-INF/tiles/header.jsp"></put-attribute>
        <put-attribute name="links"
            value="/WEB-INF/tiles/links.jsp"></put-attribute>
        <put-attribute name="content"
            value="/WEB-INF/tiles/content.jsp"></put-attribute>
        <put-attribute name="footer"
            value="/WEB-INF/tiles/footer.jsp"></put-attribute>
</definition>

<definition name="index" extends="users.base">
        <put-attribute name="title" value="Monaleen Tennis Club - Home
            Page"></put-attribute>
        <put-attribute name="content"
            value="/WEB-INF/tiles/index.jsp"></put-attribute>
</definition>

<definition name="admin" extends="users.base">
        <put-attribute name="title" value="Monaleen Tennis Club -
            Admin"></put-attribute>
        <put-attribute name="content"
            value="/WEB-INF/tiles/admin.jsp"></put-attribute>
</definition>
```

Figure 5.31: Apache Tiles Configuration

JSTL

JSTL is used within the application to manage how information was displayed. It was preferred, during the development of the application, that all of the logic be handled at the Controller level, and that the JSP pages would resolve the models passed to it into the view seen by the user. It was not desirable for the pages to contain JSP directives, or to use the implicit objects contained within JSP pages.

The main tags used within the application were the JSTL Core tags. These tags allow the usage of conditional statements and the definition of parameters within the JSP page. In order to use this technology, the relevant jar must be made available in the build path or within the Maven dependencies of the project. A declaration, as shown in Figure 5.32 must be included in all JSP pages that wish to make use of the tags also.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
```

Figure 5.32: JSTL Tag Library Declaration

Within the application, the controller will create a model and pass it to the JSP page. The page uses the JSTL tags to manage and display relevant information from the model, as shown in Figure **??**, and user actions based on the information contained within. The example below is taken from the Timetable display page, Figure 5.33 from the application.

```
@RequestMapping(value = "/gotoCourt", method = RequestMethod.POST)
public String chooseCourt(Model model, HttpServletRequest request) {
        //abbreviated method to display court, logic removed
        //highlighting the attributes within the model
        model = addDateToTimetable(model, id));
        model.addAttribute("series",timetableService.getById(id).getSeries());
        model.addAttribute("name",
            SecurityContextHolder.getContext().getAuthentication().getName());
        model.addAttribute("court", current);
        model.addAttribute("realname", name);
        model.addAttribute("bookings", left);
        if (seriesMatch(courtID, nextCourt)) {
                model.addAttribute("next", (current.getId() + 1));
        }
        if (seriesMatch(courtID, prevCourt)) {
                model.addAttribute("prev", (current.getId() - 1));
        }
        return "court";
}
```

Figure 5.33: Timetable Controller chooseCourt() method

Model Name	Summary
name	The username of the currently authenticated user
realName	The real name of the currently authenticated user
bookings	The number of remaining bookings of the currently authenticated users
date	The current week of the year and the current date. Calculated using separate method.
next	The id number of the court following the current court, if applicable
prev	The id number of the court preceding the current court, if applicable
court	The current court, determined by the current week, provided by the java.util.Date class

Table 5.4: Model Attributes

This example is an excerpt from the TimetableContoller class. The logic determining the values has been removed. This is to highlight how attributes are added to the model from within the controller. This is the information that the JSP page will have access to once it has been displayed.

The above code deals with the display of *Monday* within the Timetable display page. In the *c:forEach* tags, it loops through each value in the *court.monday* list that has been passed to it by the controller. The size of this list is determined by the user when the timetable is created, and the number of slots per day is specified. If the current value being examined in the loop is equal to the value "Free Court", it will display a link to the Book Form mapping. This aspect of the Timetable Controller will check that a user has any remaining bookings left and respond as appropriate. In the event that the value in the list does not equal "Free Court", it will make a choice. If the currently authenticated user made the booking, it will display an option to remove their booking from the slot. Otherwise, it will give any other user an option to report the user as a "no show" should a user fail to show for a previously booked slot.

```
<c:forEach var="row" varStatus="loop" items="${court.monday}">
<c:choose>
<c:when test='${row eq "Free Court"}'><tr>
<td class="inner"><form
    action="${pageContext.request.contextPath}/bookCourt"
method="POST">
<input type="hidden" value="${loop.index}" name="position" />
<input type="hidden" value="monday" name="day" />
<input type="hidden" value="${court.id }" name="ttid" />
<input type="submit" value="Book">
</form></td></tr>
</c:when>
<c:otherwise><tr><td class="inner">${row}
<c:choose>
<c:when test="${name eq pageContext['request'].userPrincipal.name && row
    eq realname }">
<form action="${pageContext.request.contextPath}/unbookCourt"
    method="POST">
<input type="hidden" value="${loop.index}" name="position" />
<input type="hidden" value="monday" name="day" />
<input type="hidden" value="${court.id }" name="ttid" />
<input type="submit" value="Unbook">
</form></c:when>
<c:otherwise>
<form action="${pageContext.request.contextPath}/reportNoShow"
    method="POST">
<input type="hidden" value="${row}" name="bookedUser" />
<input type="hidden" value="monday" name="day" />
<input type="hidden" value="${court.id }" name="ttid" />
<input type="submit" value="Report User">
</form></c:otherwise>
```

Figure 5.34: Code Showing Display of Timetable

5.4 Logging the Application

The logging for this application was provided by *log4j*. Logging became very useful for tracking down, and isolating bugs, throughout the application. Since there were a considerable number of dependencies and different technologies working together, it rapidly became very difficult to see where errors originated from. Stack-traces quickly became unmanageable. *Log4j* works by allowing the developer to view a number of logs of varying types within the application.

Log Type	Description
INFO	Messages that highlight the progress of the application at coarse-grained level
DEBUG	Fine-grained informational events that are most useful to debug an application
TRACE	Finer-grained informational events than the DEBUG
WARN	Potentially harmful situations
ERROR	Error events that might still allow the application to continue running
FATAL	Very severe error events that will presumably lead the application to abort

Table 5.5: Log Types

Log4j is configured with a properties file that allows you to see the various levels of logs displays by the applications running. Implementation of a logging system resolved a number of Spring Security issues within the web application. Spring Security, concerned with access rights to mappings within the application, did not output failed access attempts to the console. This made it very difficult to debug. When configuring *Log4j* to catch the logs created by the security components, the application became much easier to debug. The properties file for this web application is detailed below.

```
1 log4j.rootLogger=INFO, CONSOLE
2 log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
3 log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
4 log4j.logger.org.hibernate.SQL=DEBUG
5 log4j.logger.org.hibernate.type=TRACE
6 log4j.logger.org.springframework.security=DEBUG
```

Figure 5.35: Log4j Configuration

Logging can be implemented on a class by class basis. Within this application, it was used to display informational messages to the developer. These included items such as database access, objects being created and updated. In order to enable logging on a class, a logger must be instantiated with reference to the class that requires logging. The logger object then is called with a method corresponding to the type of log you wish to throw along with a message.

```
private static Logger logger = Logger.getLogger(UserDAO.class);

public Session session(){
        logger.info("Session Factory returning current session.....");
        return sessionFactory.getCurrentSession();
}

public List<User> getUsers() {
        logger.info("Selecting All Enabled Members....");
        return session().createQuery("from User where enabled =
            '1'").list();
}

public User getUserByName(String name) {
        Criteria crit = session().createCriteria(User.class);
        crit.add(Restrictions.eq("name", name));
        try{
        User user = (User) crit.uniqueResult();
        }
        catch(Exception e){
        logger.error{"Must be unique result : Thrown from
            UserDAO.getUserByName(String name));
        }
        return user;
}
```

Figure 5.36: Logger Usage within UserDAO.class

5.5 Web Services

Web services are defined as a "self contained, modular business applications that have open, Internet-oriented, standards based interfaces" (Alonso et al. 2004). The interface used within this application was JavaScript Object Notation [JSON]. JSON is an open standard, and provides a language independent data format. It consists of key-map pairs. For this project, the timetables for the current day were exposed as a web service. This allows the information to be potentially parsed by a mobile application or another web site with ease. The implemenation for this is showin in Figure 5.6.

```
@RequestMapping(value="/jsoncurrentday", method=RequestMethod.GET,
    produces="application/json")
@ResponseBody
public Map<String, Object> getCurrentDay(){
```

Table 5.7: JSON output statement

```
4         int day_of_week = Calendar.getInstance().get(Calendar.DAY_OF_WEEK)
              - 1;
5         Map<String, Object> data = new HashMap<String, Object>();
6         for (int i = 1; i <= timetableService.getNewSeries(); i++){
7                 String key = "court" + i;
8                 List<Timetable> series =
                      timetableService.getTimetableSeries(i);
9                 Timetable current =
                      series.get((Calendar.getInstance().get(Calendar.WEEK_OF_YEAR)
                      - 1));
10                switch(day_of_week){
11                        case 1: {
12                                data.put(key, current.getMonday());
13                        }
14                        // repeat for 2 through 7
15                }
16        }// end for
17        return data;
18  }
```

Table 5.6: JSON in Spring

The *RequestMapping* annotation must be changed to include a *produces* attribute, shown in Line 2. This ensures that Spring returns a valid JSON statement to the HTTP response. This is ensured through the use of the *ResponseBody* annotation in Line 2. An example of the output is illustrated in Figure 5.7.

```
1  {"day2":["Free Court","Free Court","Free Court","Free Court","Free
       Court","Free Court","Free Court","Free Court"],"day1":["Free
       Court","Free Court"]}
```

5.6 Configuration of the Application

In order to begin implementation with the Spring MVC framework, there are a number of configuration files that are necessary. The core file is the *web.xml* file. This file is responsible for the configuration for the framework. One of the key responsibilities is the definition of the context xml files, whose purpose will be elaborated on later. Different development profiles can be configured within this file in order

```
<property name="hibernateProperties">
<props>
<prop key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
</props>
</property>
```

Figure 5.38: Hibernate Configuration

to produce different development environments, such as production and testing environments. The configuration of this file within the application is shown in Figure 5.37

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
        classpath:beans/dao-context.xml
        classpath:beans/service-context.xml
        classpath:beans/security-context.xml
</param-value>
 </context-param>
```

Figure 5.37: Spring Configuration

Of particular importance are the definition of the context parameters. In this project, there were three main context files.

- Data Access Object Context
- Service Context
- Security Context

The DAO Context file specifies the packages that contain the various DAO classes within the application. It also contains configurations for both the database connection details, and Hibernate configurations. Packages containing entity classes for Hibernate are specified within this context also.

The Service Context file is responsible for specifying the base package containing the Service classes necessary to facilitate the collaboration between the Controller classes and the DAO classes. This file specifies that annotations will be used to

```
<context:annotation-config></context:annotation-config>
<context:component-scan base-package="service"></context:component-scan>
```

Figure 5.39: Service Context Configuration

configure the Service classes.

The Security Context file is the larger of the three files, and is responsible for the security configuration of the web application. There are four main areas within the file that were used to configure the web application created in this project.
The User Service aspect of the configuration file is responsible for retrieving users and their authority within the scope of the web application.
The URL access configuration ensures that only users who are authorised to access certain portions of the site are allowed access.
The Security Annotations allow the creation of an extra level of security into an application. At class level, annotations can be placed on methods to further ensure that proper access is enforced throughout the application.
Lastly, the Security Context is responsible for creating the password encoder bean in which passwords are encoded, and decoded, upon account creation and login. This ensures that no passwords in plain text form are ever stored on either the server or the database within the web application

- User Service

```
<security:authentication-manager>
        <security:authentication-provider>
        <security:jdbc-user-service data-source-ref="dataSource"
                id="jdbcUserService"
                    authorities-by-username-query="select username,
                    authority from users where binary username = ?" />
        <security:password-encoder
            ref="passwordEncoder"></security:password-encoder>
        </security:authentication-provider>
</security:authentication-manager>
```

- URL Access

```
<security:intercept-url pattern="/timetable" access="permitAll"/>
<security:intercept-url pattern="/reportNoShow" access="permitAll"/>
<security:intercept-url pattern="/admin"
    access="hasRole('ROLE_ADMIN')"/>
<security:intercept-url pattern="/approveMembers"
    access="hasRole('ROLE_ADMIN')"/>
```

- Security Annotation for Service Class

```
<security:global-method-security
    secured-annotations="enabled"></security:global-method-security>
//Java Code from TimetableService class.
//This code is invoked when booking a slot on the timetable and is
    only accessible by registered members.
@Secured({"ROLE_ADMIN", "ROLE_MEMBER", "ROLE_COMMITTEE",
    "ROLE_WARNING", "ROLE_SUSPEND"})
        public void update(Timetable t){
                timetableDAO.updateTimetable(t);
        }
```

- Password Encoding

```
<bean id="passwordEncoder"
class="org.springframework.security.crypto.password.StandardPasswordEncoder">
</bean>
```

5.7 Test Driven Development

The primary method of testing was implemented using JUnit. A Test Suite of JUnit tests were prepared to test the key features of the application. A separate test database was constructed. It was important to ensure that the testing environment was using the same context files as the production environment. The test class had to be annotated to enforce this. While the context files were the same, the DataSource file has changed as a different database is being using in this environment.

```
@ActiveProfiles("dev")
@ContextConfiguration(locations = {
    "file:src/main/java/beans/dao-context.xml",
"file:src/main/java/beans/security-context.xml",
"classpath:test/config/datasource.xml" })
@RunWith(SpringJUnit4ClassRunner.class)
public class HibernateTests {

        @Autowired
        private UserDAO userDAO;

        @Autowired
        private TournamentDAO tournamentDAO;

        @Autowired
        private DataSource dataSource;

        //Class truncated
}
```

Figure 5.40: JUnit Test Example

The database is then initialised to ensure the tests are being run against the same database, and that repeat tests are consistent.

```
@Before
public void init(){
        JdbcTemplate jdbc = new JdbcTemplate(dataSource);
        jdbc.execute("delete from users");
}
```

Figure 5.41: JUnit @Before Test Configuration

In these example tests, the UserDAO is being tested to ensure that it returns true

when the exists() method is called on it. This is important within the scope of the application to ensure that primary keys are not duplicated. The method is annotated with *@Test*. The methods *assertTrue* and *assertFalse* expect a return value of true and false respectively. They take two parameters: an error message and a boolean value, or a method that returns a boolean value. In the *assertTrue* method below, the UserDAO will return true if the user exists. In the event that the user does not exist, it will fail the test and return the message "User should exist".

```
1  @Test
2  public void testExists(){
3          userDAO.createUser(user1);
4          assertTrue("User should exist",
               userDAO.exists(user1.getUsername()));
5          assertFalse("User should not exist",
               userDAO.exists("jkljfksakfjahghdsopjclkhfkjafhkjdshFHajhgouwe"));
6  }
```

Figure 5.42: JUnit UserDAO Exists() Test

Another test with the UserDAO was to ensure that users were being saved correctly. In this example, users are being created and saved to the database. The method *assertEquals* checks two interger values and returns an error message if they do not match.

```
1  @Test
2  public void testCreateRetrieve(){
3          userDAO.createUser(user1);
4          List<User> users1 = userDAO.getAllUsers();
5          assertEquals("One user should have been created and retrieved", 1,
               users1.size());
6          assertEquals("Inserted user should match retrieved", user1,
               users1.get(0));
7          userDAO.createUser(user2);
8          userDAO.createUser(user3);
9          userDAO.createUser(user4);
10         List<User> users2 = userDAO.getAllUsers();
11         assertEquals("Number of users should be four", 4, users2.size());
12 }
```

Figure 5.43: JUnit Create and Size Test

5.8 External Code

There are two aspects of external code being used within this web application. The first is the CSS file that the site is using. This was a free template obtained from Skylines Templates

This template was modified in order to fit in with certain aspects of the site, but the look, feel and images remain consistent with those of the template.

The other code used in this application that was not original is the Google Maps script, visible in the Contact Us page of the web application. This is provided at Google Maps Simple Map Example and is available to use freely. The code was slightly modified, as depicted in Figure 5.8, to change both the GPS co-ordinates and the initial zoom level of the map. The page is included in the Contact Us tile.

```
1  //code to include maps.jsp in contactus.jsp
2  <div align="center"><%@include file="maps.jsp"%></div>
3
4  //code of the maps.jsp change with modified Google javascript
5  <script src="http://maps.googleapis.com/maps/api/js?sensor=false">
6  </script>
7  <script>
8  function initialize()
9  {
10 var mapProp = {
11 center:new google.maps.LatLng(52.6565176,-8.5537577), //GPS Co-ords
12 zoom:18, // Zoom level
13 mapTypeId:google.maps.MapTypeId.ROADMAP
14 };
15 var map=new google.maps.Map(document.getElementById("googleMap"),mapProp);
16 }
17 google.maps.event.addDomListener(window, 'load', initialize);
18 </script>
```

Table 5.8: Code Showing Google Maps Integration

5.9 Conclusion

Chapter 6

Software Quality

6.1 Introduction

The use of software metrics within this project was to guide the development process and measure the quality of the application. Measurement is concerned with "capturing information about *attributes* of *objects*" (Fenton 1991).

6.2 Application Metrics

The measurement of metrics within the IDE was accomplishing through the use of a metrics plugin for STS/Eclipse called *Metrics 1.3.6*. The plugin scans the source directory and displays a number of metrics, an example of which is shown in Table 6.1.

Metric	Total	Mean	Std. Deviation	Maximum
McCabe Cyclomatic Complexity	n/a	1.262	.862	8
No. Parameters per method	n/a	0.894	1.115	15
Nested Block Depth	n/a	1.126	.588	5
Number of Classes	56	4.538	3.456	10
Number of Attributes	126	2.25	3.537	17
Number of Methods	516	9.21	10.38	39
Method Lines of Code	2134	3.88	6.732	66
Total Lines of Code	4266	n/a	n/a	n/a

Table 6.1: Application Metrics

The *McCabe Cyclomatic Complexity* metric measures the number of linearly inde-

pendent paths through a program's source code. The higher this value, the more complex the code is, thereby increasing the cost of maintenance and affecting application extensibility.

An example of how frameworks may negatively affect metrics in an application is illustrated within the User class. The user class contains the largest number of attributes with 16. In order for Hibernate to persist and retrieve the object correctly, getters and setters must be defined for each attribute. This would result in the creation of 32 methods in this class. These methods may not be used within the application other than by Hibernate. There are 126 class attributes within the application which require 252 getter and setter methods to be created. This accounts for 48.8% of the methods within the application. This would also have an effect on the *Method Lines of Code* metric, as it would add 252 lines of that metric, or 11.8% of the total.

6.3 Software Quality Tools and Visualisations

6.3.1 Software Quality

InFusion, by intooitus, was used to examine the project for architectural and design quality deficiencies. This tool evaluates the project and highlights any problem areas, or bad code smells. A bad code smell, a phrased coined by Kent Beck, is a "surface indication that usually corresponds to a deeper problem in the system" (Fowler 2006). Beck, Beck and Fowler offer refactoring methods that aid in dispersing these smells.

The bad code smells identified within this project as illustrated in Figure 6.1.

1. God Class

    - A God class is an object that uses many attributes from external classes. The main class within this application to be flagged was the TimetableController. This class is responsible for creating models for the Views relating to the timetable, and for engaging the Service layer to persist and retrieve Timetable objects from the database. As such, this object needs to be aware of classes that are necessary for the proper functioning of the Timetable class, such as Users, Events, Logs, Emails and Roles. The reasons for the coupling of these objects is listed in Table 6.2.

Class	Purpose
User	Authentication to use Timetable
Events	System Event to count user bookings
Logs	Record instances of NoShows for analysis
Emails	Notification of admins for NoShows
Roles	Check amount of bookings allowed per role

Table 6.2: Application Metrics

2. Feature Envy

    - This is when a method in a class "seems more interested in a class other than the one it is actually in" (Fowler, Beck, and Beck 1999). The application suggest the use of *Move Method* refactoring. This requires the creation of new method with a similar structure within the second class, which the old method performs a simple delegation or is eliminated entirely.

3. Data Clumps

    - This is when the same data types are passed to methods when it would make sense to encapsulate this data within its own object. The solution is to use *Extract Class* refactoring on the data fields to "turn the clumps into an object" (Fowler, Beck, and Beck 1999). Attention then needs to be paid to the method signatures using *Introduce Parameter Object* refactoring to slim them down. Using these refactoring methods, "you can shrink a lot of parameter lists and simplify method calling" within a system (Fowler, Beck, and Beck 1999).

4. Data Class

    - This is a class that has attributes, and just getters and setters for those attributes. These classes "are dumb data holders and are almost certainly being manipulated in far too much detail by other classes" (Fowler, Beck, and Beck 1999). In this application however, the use of getters and setters is as a result of the use of the Hibernate framework which relies on these methods to function efficiently. The inFusion application only flagged Hibernate entity classes as possessing this design flaw, and it is a flaw that is created by the use of this ORM framework.

Figure 6.1: Bad Code Smells

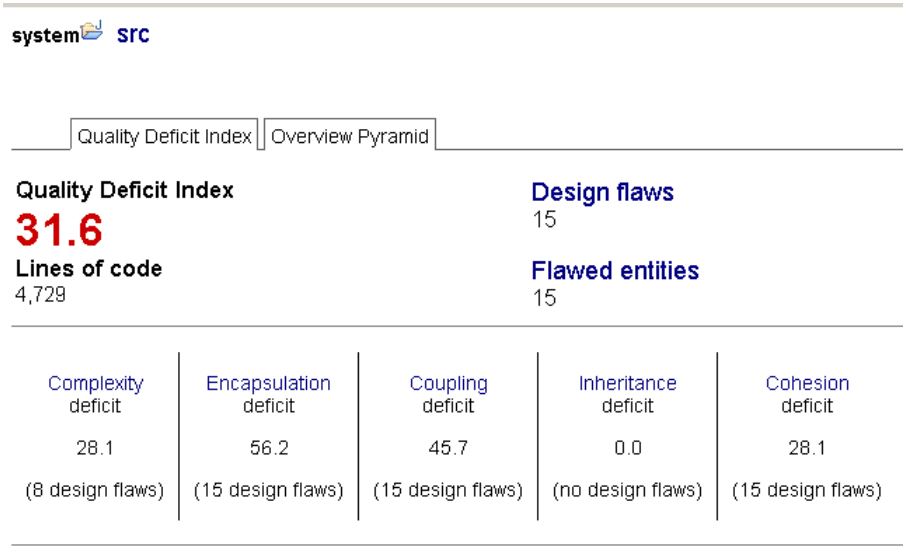

Figure 6.2: Pre Re-factoring using Infusion

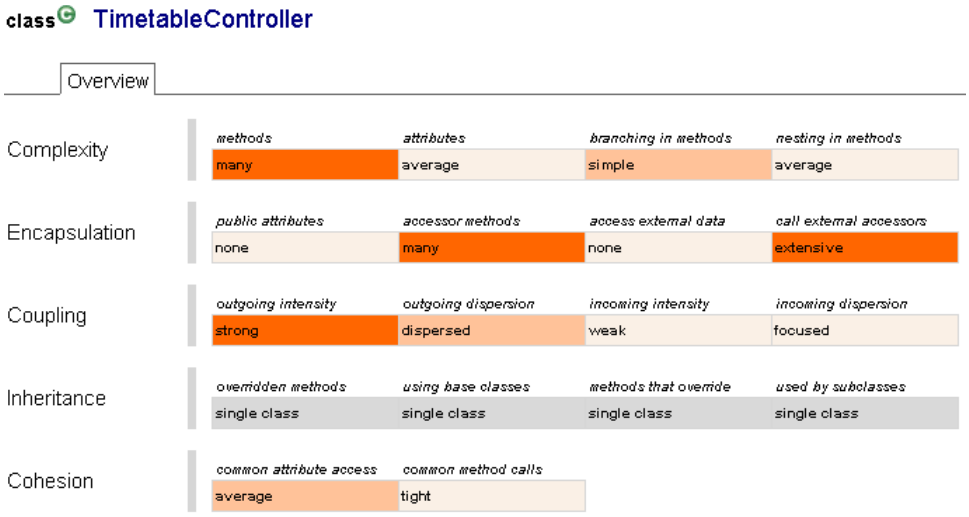

Figure 6.3: Example of Class 'Bad Code Smell' Breakdown using Infusion

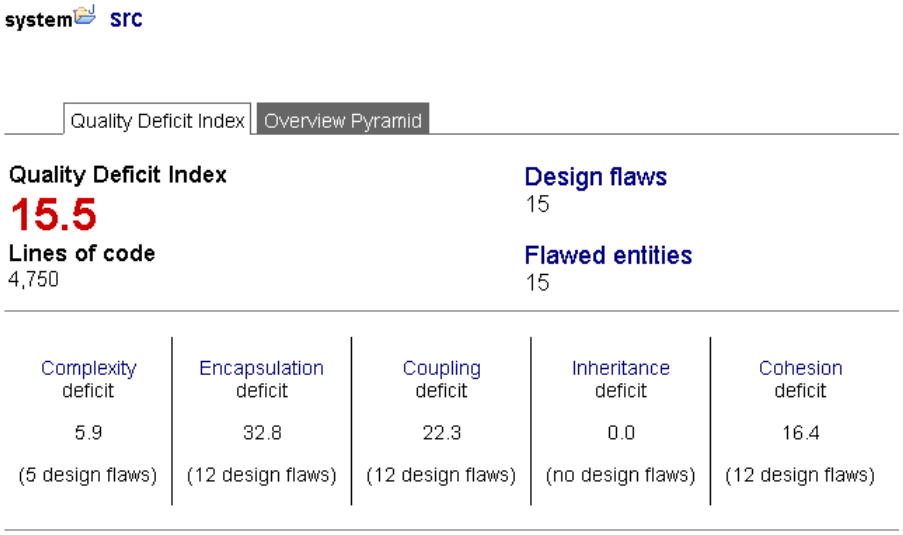

Figure 6.4: Post Re-factoring using Infusion

6.3.2 Visualisations

The software used to visualise the code was CodeCity, available at http://www.inf.usi.ch/phd/wettel/codecity.
In order to run CodeCity, a MSE file needed to be created by inFusion. The CodeCity application reads this file, and represents your code as a city. The city is broken down into blocks, defined by a package, and within each block, there are a number of buildings, that represent classes. The size of the building is related to the size and complexity of the class it represents. Figures **??** and **??** show the 2D and 3D representations of the application created for this project.

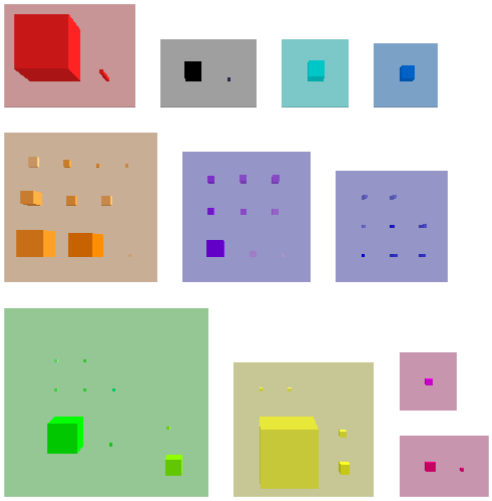

Figure 6.5: CodeCity 2D Visualisation of Application

Figure 6.6: CodeCity 3D Visualisation of Application

Visualisation is a quick way to highlight potential issues and bad code smells, such as data classes, by looking at the size of a building. While this does not fix any problems, it aids in drawing attention to potential issues through the development

cycle.

6.4 Sample Refactorings

Chapter 7

Evaluation

7.1 Security

7.1.1 OWASP Vulnerabilities

In analysing support for security, the Open Web Application Security Project [OWASP] was examined. OWASP provides a list of the top 10 vulnerabilities that exist within web applications. The following vulnerabilities are addressed directly by the frameworks used in this project.

1. Injection
    - Injection occurs when "untrusted data is sent to an interpreter as part of a command or query" (Wichers 2010). This can allow unauthorised access to a database, and the exposure of the data within. Hibernate aids the prevention of Injection by providing support for named parameters. Hibernate further prevents injection by eliminating SQL queries entirely, and by creating objects through mapping of attributes to columns within the database. JDBC does not support the use of named parameters natively, but Spring does provided a *NamedParameterJdbcTemplate* class which provides this functionality to the JDBC class.
2. Broken Authentication and Session Management
    - This occurs when "application functions related to authentication and session management are often not implemented correctly" (Wichers 2010). When developers create their own session management schemes, they will inadvertently introduce flaws. Spring provides robust session management and authentication techniques through the use of EBAC, as discussed earlier. This allows the developer to focus more on the design

of the application, and the business logic within, without having to be concerned with created a strong security solution.

3. Cross Site Scripting (XSS)

   - XSS is "whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping" (Wichers 2010). This allows the potential use of malicious scripts in a victims browser, which may enable an attacker to gain access to the user session. Spring provides an extension of the Java EE 6 class *HttpServletRequestWrapper* called *XSSWrapper*. This class takes a HTTP Request object, and checks the values of getParamters(), getParameterValues() and getHeaders(). The XSSWrapper object then performs a stripXSS() method that checked the validity of the parameters with an HttpServletRequest object, and in the event that an attempted XSS attack is present, nullifies it by escaping special characters.

4. Security Misconfiguration

   - Security Misconfiguration is not "having a secure configuration defined and deployed for the application framework" (Wichers 2010). Within Spring MVC, this refers to the configuration for EBAC. An example of poor configuration would be to implement this system by denying access to URL mappings, rather than denying access to all and allowing exceptions. While it is possible to deny access to relevant mappings, it is more likely that a mistake would slip through. A breakdown of the mappings organised by the type of access is depicted in Table 7.1. permitAll refers to mappings that allow access to all users, isAuthenticated requires that the user be logged in, and admin requires that a user be logged in and possess an administrator role.

permitAll	**isAuthenticated**	**admin**
23	13	50

Table 7.1: EBAC Breakdown

5. Sensitive Data Exposure

   - Sensitive Data Exposure deals with the encryption of sensitive user information such as passwords, credit card details and anything that made lead to identity theft. This data "deserves extra protection such as encryption at rest or in transit" (Wichers 2010). Within the scope of this

application, the main concern was user passwords, as no payment information is held. Spring provides methods for encoding data using 256bit AES encryption. The encrypted value is stored in the database, not the actual password. When the user logs in, the plaintext password is encrypted using a key unique to that user. The encrypted password is then compared to the stored value. Each user has a unique key to generate the password so that even if two users have the same password, it will not appear the same in the stored values.

6. Unvalidated Redirected and Forwards

   - Within any web application, they will be scope for redirects. For example, if a user logs in, it is normal to redirect them to the home page. However, the application must be cognisant of where the redirect is coming from, and ensure its validity. "Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages" (Wichers 2010). The ViewResolver, provided in Spring and extended by Apache Tiles, is responsible for validating this information. All redirected are passed to the ViewResolver as a string. If a value corresponding to this string is not found within the ViewResolver configuration, an error will be returned in the form of a 404 Page Not Found value or an access denied value, as defined in the security-context file.

7.1.2 Security Bugs

A significant bug was discovered within the application during user testing. When a user was editing their profile, a model attribute was used to map the data changed in the View to create an object within the Controller. The application did not allow the user to edit all attributes within the user class, such as their role within the application. As a result, any changes to a user object where the attributes were not defined within the view resulted in attributes not present in the view being set as null. This meant that a user would have no role within the system, as the role attribute would be changed to null. To overcome this, a hidden value mapped to the role attribute was inserted into the form. This fixed the problem, but in doing so, introduced a significant security risk.

Using a tool to intercept HTTP Requests, such as WebScarab, it was possible to change the hidden field. This would allow a regular user to change their role from ROLE MEMBER to ROLE ADMIN, giving them full access to the site. This happened as the object was not checked before being persisted by the controller, service and dao layers. This method was tested using the WebScarab tool, and the vulnerability was confirmed. WebScarab is a framework that analysing applications

that communicate using HTTP and HTTPS protocols, and was used to intercept GET and POST requests in this application. In order to circumvent this, the role of the currently logged in user was retrieved from the framework, and this value was saved, rather than a value defined within the view. This example showed that while the framework does provide a considerable layer of security, misuse and bad programming can expose the application.

7.2 Extensibility

The goal of extensibility is "controlling the time and costs to implement, tests and deploy changes" (Bass, Clements, and Kazman 2003). The use of Hibernate in the architectural stack provides a measure of extensibility support for changes in entity classes. Any addition to an entity class would result in a change to a database schema.

An application using Hibernate, in conjunction with Spring MVC, would need only to modify the View, illustrated in Figure 7.2. This would need to be changed to facilitate input of the new attribute. Spring MVC uses spring form tags to map values to the object within the Controller, and the DAO layer persists the object. Since the object is configured with database mapping at class level, no changes are necessary to the DAO. The addition of Line 14 is Figure 7.2 is the only changed needed to allow for the attribute to be modified within the application. The Controller, Service layer and DAO would remain unchanged.

```
<sf:form id="details" method="post"
    action="${pageContext.request.contextPath}/adminProfileUpdate"
    commandName="member">
<table class="formtable">
Name:<sf:input name = "name" path="name" type="text"/>
Email:<sf:input name = "username" path="username" type="text"/>
Password:<sf:input name = "password" path="password" type="text"/>
Address Line 1<sf:input name = "ad_line1" path="ad_line1" type="text"/>
Address Line 2<sf:input name = "ad_line2" path="ad_line2" type="text"/>
City/Town<sf:input name = "ad_city" path="ad_city" type="text"/><br/>
County<sf:input name = "ad_county" path="ad_county" type="text"/><br/>
Contact Number<sf:input name = "contact_num" path="contact_num"
    type="text"/>
Emergency Contact Name<sf:input name = "em_con_name" path="em_con_name"
    type="text"/>
Emergency Contact Number<sf:input name = "em_con_num" path="em_con_num"
    type="text"/>
Role<sf:select path="authority"><sf:options items="${roles }"/></sf:select>
New Attribute<sf:input name = "new_attribute" path="new_attribute"
    type="text"/> //new line added
<input type="hidden" value="${member.username }" name="username"/>
<input value="Change Details" type="submit"/>
```

Table 7.2: Change to view to support new attributes

Using JDBC, again in partnership with Spring MVC, the View and the DAO object would need to be changed. Within the DAO, two actions must be performed, as depicted in Figure 7.3. Line 18 shows the addition of an extra parameter to map the value to a position in the SQL statement. The SQL statement must be modified also in order to correctly insert or update the persisted value in the database. This is shown in Lines 23 through 26, with the changes being shown in capital letters.

```
@Transactional
public boolean createUserJDBC(User user) {
MapSqlParameterSource params = new MapSqlParameterSource();
params.addValue("username", user.getUsername());
params.addValue("name", user.getName());
params.addValue("password", passwordEncoder.encode(user.getPassword()));
params.addValue("gender", user.getGender());
params.addValue("member_type", user.getMember_type());
params.addValue("grade", user.getGrade());
params.addValue("ad_line1", user.getAd_line1());
params.addValue("ad_line2", user.getAd_line2());
params.addValue("ad_city", user.getAd_city());
```

```
13 params.addValue("ad_county", user.getAd_county());
14 params.addValue("contact_num", user.getContact_num());
15 params.addValue("em_con_name", user.getEm_con_name());
16 params.addValue("em_con_num", user.getEm_con_num());
17 params.addValue("role", "ROLE_MEMBER");
18 params.addValue("NEW_ATTRIBUTE", "user.getNEW_ATTRIBUTE());
19 return jdbc
20        .update("insert into users (username, name, password, gender,
21        member_type, grade, ad_line1, ad_line2, ad_city, ad_county,
22        contact_num, em_con_name, em_con_num, enabled, authority,
23        bookings_left, NEW_ATTRIBUTE) values (:username, :name, :password,
24        :gender, :member_type, :grade, :ad_line1, :ad_line1,
25        :ad_city, :ad_county, :contact_num,:em_con_name, :em_con_num,
26        false, :role, 0, :NEW_ATTRIBUTE)",params) == 1;
27        }
```

Table 7.3: JDBC DAO changes with entity change

Hibernate provides better support for extensibility in this regard, allowing the developer to focus on database and view changes, with no need to modify the persistence layer between the application and the database.

7.3 Usability

The deliverable from this project was subjected to the same usability criteria as illustrated in Section 2.5. The results are shown in Tables 7.4 through 7.8. The overall results are compared in Figure 7.9.

Criteria	**Project Application**
Links are updated?	Yes
Short-cuts for frequent users?	Yes
Warning if link leads to large file?	n/a
Indication of restricted access for link	Yes
Link text indicates nature of target	Yes
Total	4/5

Table 7.4: Table 1 - Links

Criteria	**Project Application**
Contact Details for website maintainer	Yes
Link to page maintainer on each page	Yes
Forms provided for users to enter details	Yes
FAQ provided	Yes
Feedback mechanisms are fully operational	Yes
Total	5/5

Table 7.5: Table 2 - Feedback Mechanisms

Criteria	**Project Application**
Speed of site adequate	Yes
Site Availability High?	Yes
Website made known through search tools**	n/a
Link back to parent entity	Yes
Name of entity reflected in URL	Yes
URL is not over complex,likely to be mistyped	Yes
Total	5/8

Table 7.6: Table 3 - Accessability

Criteria	**Project Application**
Format,Graphic Design Appropriate	Yes
Pages appropriate length, clear, uncluttered	Yes
Consistent format through website	Yes
Site compatible with multiple browsers	Yes
Site can be used without graphics	No
Total	4/5

Table 7.7: Table 4 - Design

Criteria	**Project Application**
Website organised by anticipated user need	Yes
Navigation options are distinct	Yes
Conventional navigation models*	Yes
Navigation links are provided on all pages	Yes
Browsing facilitated by menu or site map	Yes
Reach any point in appropriate number of clicks**	Yes
Search Engine Provided	No
Total	6/7

Table 7.8: Table 5 - Navigability

**eg menu on top or left of screen*
***defined as 3,* Smith 2001

Site	Total
Monaleen Tennis Club	14/30
Tralee Tennis Club	22.5/30
Project Application	24/30

Table 7.9: Table 6 - Overall Results

7.4 Productivity and Performance

7.4.1 Hibernate

One of the core technologies used within the application was Hibernate. One of the main benefits Hibernate provided to the project was simplifying the code in order to persist objects. When developing the application, the only concern was with objects as a whole, not on their individual attributes. With JDBC, there was a need to use SQL syntax within the application. In the case of the User class, there were 16 attributes which made the SQL INSERT statement for that class, shown in Figure 7.1, difficult to read and troubleshoot.

```
@Transactional
public boolean createUserJDBC(User user) {
	MapSqlParameterSource params = new MapSqlParameterSource();
		//other params cut out to save space
		params.addValue("role", "ROLE_MEMBER");
		return jdbc
			.update("insert into users (username, name, password,
			    gender,
			member_type, grade, ad_line1, ad_line2, ad_city,
			    ad_county,
			contact_num, em_con_name, em_con_num, enabled,
			    authority,
			bookings_left) values (:username, :name, :password,
			:gender, :member_type, :grade, :ad_line1, :ad_line1,
			:ad_city, :ad_county, :contact_num,:em_con_name,
			    :em_con_num,
			false, :role, 0)",params) == 1;
	}
```

Figure 7.1: JDBC SQL INSERT

However, it could be arguing by supplying a framework, such as Hibernate, to automate the creation of a query language that performance of an application could be affected. To examine this briefly, a small test was created. This this test, the application would use JDBC to create 10,000 records, update those records and then delete them. The application would then perform the same actions with the same data using Hibernate. The time taken to perform these tasks as a whole would be recorded in nanoseconds. The test is illustrated in Figure 7.2 below.

```
@RequestMapping("/testdatabases")
public String testdatabases(Model model){
        User user = new User(); // create a new user
        long jStart = System.nanoTime();
        model.addAttribute("jdbcStart", jStart);
        for (int i = 0; i < 10000; i++){
                user.setUsername(i + "@test.ie");
                userService.createJDBC(user);
                userService.enableUserJDBC(user);
                userService.deleteUserJDBC(user);
        }
        model.addAttribute("jdbcFinish", (System.nanoTime() - jStart));

        long hStart = System.nanoTime();
        model.addAttribute("hibernateStart", hStart);
        for (int i = 0; i < 10000; i++){
                user.setUsername(i + "@test.ie");
                userService.create(user);
                user.setEnabled(true);
                userService.updateUser(user);
                userService.delete(user);
        }
        model.addAttribute("hibernateFinish", (System.nanoTime() - hStart));
        return "testdatabases";
}
```

Figure 7.2: JDBC Vs Hibernate Test

The results are displayed in Table 7.10 below. NS refers to NanoSeconds, and MM:SS refers to the total in minutes and seconds. The test was run three times and an average was created. The test machine was an Amazon EC-2 server, which was a 32-bit micro instance, with 650mb of RAM with one virtual CPU. This is running a modified version of Red Hat Enterprise Linux.

Run	Database	No Queries	Total (NS)	Total (MM:SS)
1	Hibernate	10,000	198469890740	03:20
1	JDBC	10,000	211883464327	03:30
2	Hibernate	10,000	188055485374	03:07
2	JDBC	10,000	200521721809	03:22
3	Hibernate	10,000	188423897510	03:06
3	JDBC	10,000	193541735201	03:20
Avg	Hibernate	10,000	191649757878	03:12
Avg	JDBC	10,000	201982307112	03:21

Table 7.10: Hibernate Vs JDBC Results

```
//Excepts from the UserDAO for Hibernate and JDBC actions
//Hibernate
return sessionFactory.getCurrentSession(); //return current session
user.setPassword(passwordEncoder.encode(user.getPassword()));
session().save(user);
session().update(user);
session().delete(user);

//JDBC
MapSqlParameterSource params = new MapSqlParameterSource();//start create
params.addValue("username", user.getUsername());
params.addValue("name", user.getName());
params.addValue("password", passwordEncoder.encode(user.getPassword()));
params.addValue("gender", user.getGender());
params.addValue("member_type", user.getMember_type());
params.addValue("grade", user.getGrade());
params.addValue("ad_line1", user.getAd_line1());
params.addValue("ad_line2", user.getAd_line2());
params.addValue("ad_city", user.getAd_city());
params.addValue("ad_county", user.getAd_county());
params.addValue("contact_num", user.getContact_num());
params.addValue("em_con_name", user.getEm_con_name());
params.addValue("em_con_num", user.getEm_con_num());
params.addValue("role", "ROLE_MEMBER");
return jdbc.update("insert", params) == 1; //end create
MapSqlParameterSource params = new MapSqlParameterSource(); //start update
params.addValue("enabled", enabled);
params.addValue("username", username);
jdbc.update("update",params); // end update
MapSqlParameterSource params = new MapSqlParameterSource(); //start delete
params.addValue("username", user.getUsername());
jdbc.update("delete",params); //end delete
```

Figure 7.3: Hibernate-JDBC Code

In all tests, Hibernate took less time to perform all actions, though there was not much different between both technologies. However, due to the way Hibernate is configured, it is certainly much easier to develop with. Hibernate required 5 lines of code, shown in Figure 7.3, where JDBC requires 23 lines of code, with 3 different SQL queries, one for INSERT, UPDATE and DELETE.

By defining and mapping attributes in a database to the class with Hibernate, the DAO classes become much smaller and easier to manage with no impact on performance within the application. In this application, for example, there are

more than 70 DAO operations, which would mean 70 or more SQL statements. Use of JDBC would also require the definition of SQL JOINs in relation to both the Timetable and Tournament classes, thereby again increasing complexity within the application. There is more configuration to be undertaken with Hibernate, but the end result is a much easier to manage persistence solution that reduces code complexity.

7.4.2 Session and Cookie Management

One benefit of using a framework such as Spring is that the developer does not need to be overly concerned with defining and checking a session in each controller. In a previous project, the developer was responsible to creating and managing the session, as well as ensuring proper access to the site was maintained. Figure 7.4 shows how a session must be manually retrieved from a request object within a servlet.

```
1  @Override
2  protected void doGet(HttpServletRequest request, HttpServletResponse
       response)
3  throws ServletException, IOException {
4      PrintWriter out = response.getWriter();
5      // Print Generic header then check authenitcation and print HTML
6      printHeader(out);
7      HttpSession currentSession = request.getSession(false);
8      checkAuthentication(out, currentSession);
9      // Print search form
10 }
```

Figure 7.4: Manual Session Creation

In the *checkAuthentication()* method, depicted in Figure 7.5, the developer must check the session for any attributes that match a certain value, and act accordingly. In this example, the id of a user. The id then needs to checked against a database to ensure that it is a valid id, and to determine what permissions the account has.

```
1  private void checkAuthentication(PrintWriter out, HttpSession
       currentSession) {
2  //Check Authentication
3          if (currentSession.getAttribute("ID") == null) {
4                  //print registration screen
5      } else {
6          // Use ID and Username to identify users. At this point we know
               there is a session with an ID, verify it is a valid session
```

```
7         int intID = (Integer) currentSession.getAttribute("ID");
8         String username = (String) currentSession.getAttribute("username");
9         if (customerBean.checkCustomerId(intID) == true &&
              customerBean.checkCustomerUsername(username) == true) {
10                    Customer currentCustomer =
                          customerBean.getCustomer(intID);
11                    // Print what customer should see
12            }
13                else if (adminBean.checkAdminId(intID) == true &&
                      adminBean.checkAdminUsername(username) == true) {
14            Admin currentAdmin = adminBean.getAdmin(intID);
15                    // Print what admin should see
16            }
17        }
18        }
```

Figure 7.5: Session Authentication based on user level

An issue with that is that it is reliant on the developer to identify the correct attribute. This is a user defined attribute that may not be unique to the application. Spring manages this through the *security-context.xml* file, which encapsulates security for the application to one file, and the framework itself ensures that the correct session is created for a user. There is no need to specify any of this within the codebase of the application, which again reduces complexity, and minimises the possibility of security leaks being introduced through developer errors.

7.5 Architectural Quality

(Is there a need for this section? Does the preceding section suffice? Evolution support is covered, security can be measured in resisting, identifying and recovering from attacks Bass, Clements, and Kazman 2003

7.5.1 Performance

7.5.2 Security

7.5.3 Evolution Support

Performance, Security and Evolution Support

7.6 Functional and Non Functional Requirement Evaluation

7.7 FYP Process and Learnings

The FYP has been certainly been significant part of the overall college journey. Looking back, and knowing what I know now, there are a number of avenues that I would have liked to explore.

7.7.1 Hibernate in Design

Hibernate was a very powerful framework throughout the development of this project. It was very much a case of learning by doing. Having accumulated experience with the framework, I would like to apply it more in the design stage on an application. One idea that came to me late in the project was the idea of having a top level class that could build the entire application, as depecited in Figure 7.6.

```
@Entity
public class WebApplication(){

@ElementCollection
@CollectionTable (name = "tournament", joinColumns=@JoinColumn(name="id"))
private Tournaments tournaments;

@ElementCollection
@CollectionTable (name = "timetable", joinColumns=@JoinColumn(name="id"))
private Timetable timetable;

@ElementCollection
@CollectionTable (name = "news", joinColumns=@JoinColumn(name="id"))
private News news;

@ElementCollection
@CollectionTable (name = "events", joinColumns=@JoinColumn(name="id"))
private Events events;

@ElementCollection
@CollectionTable (name = "users", joinColumns=@JoinColumn(name="id"))
private User users;

public WebApplication(){
        create();
}

```

```
28 private void create(){
29         tournaments = getTournaments();
30         etc.....
31 }
32
33 //getters and setters
```

Figure 7.6: 'Top Level' Class

(This section needs to be rewritten. I'm tired right now so just doing stream of conciousness)

It would be possible to build the application so that all classes were linked through a hierarchy to one root class. This may impact performance, but it would allow a number of benefits such as ease of deployment, and the ability to take snapshots of the application at any stage.

I also would have like to use Hibernate with more complicated objects rather than just collection classes, but time constraints did not allow this.

7.7.2 Tournament Business Logic

The tournament section of the site is something that I would like to have done a bit more on, but unfortunately, due to the time constraints within a FYP, a decison had to be made to focus more on either the timetable or the tournament. The timetable provides a service for all members of a club, not just those playing competitively. In the interim, third party solutions such as *http://ti.tournamentsoftware.com* exist. The tournament sorting mechanism was designed so that the logic of sorting members into teams is encapsulated in one method in an interface. Implementing a new version would not require any significant changes to the application.

7.7.3 Web Services and HTML5

Other aspects of a web application, such as web services and HTML5 were omitted from the final submission due to time constraints, but are definitely something that I would have an interest in looking at later on. Web services were touched on briefly, through the exposure of part of information about the timetable.

7.7.4 Timetable

While the timetable works well in its current state, I would like to optimize the code to be more readable. I experimented with the use of key-map objects later in the

development cycle, and beleive the use of these could reduce the code size in the View by at least 60%, and help compartmentalize two or three methods into one.

Chapter 8

Conclusions

Bibliography

Alonso, Gustavo et al. (2004). *Web services*. Springer.

Bass, Len, Paul Clements, and Rick Kazman (2003). *Software architecture in practice*. Addison-Wesley Professional.

Bauer, Christian and Gavin King (2005). *Hibernate in action*. Manning Greenwich CT.

Berry, Daniel M et al. (2013). "Requirements Specifications and Recovered Architectures as Grounded Theories". In: *The Grounded Theory Review, Volume 12, Issue 1*.

Davis, Alan et al. (2006). "Effectiveness of requirements elicitation techniques: Empirical results derived from a systematic review". In: *Requirements Engineering, 14th IEEE International Conference*. IEEE, pp. 179–188.

Fenton, Norman E. (1991). *Spftware Metrics: A Rigorous Approach*. Kluwer Academic Publishers.

Fowler, Martin (2006). *Code Smell*. URL: `http://martinfowler.com/bliki/CodeSmell.html`.

Fowler, Martin, Kent Beck, and Grandma Beck (1999). "Bad smells in code". In: *Refactoring: Improving the design of existing code*, pp. 75–88.

Gorton, Ian (2006). *Essential software architecture*. Vol. 14. Springer.

Holzinger, Andreas (2005). "Usability engineering methods for software developers". In: *Communications of the ACM* 48.1, pp. 71–74.

Rochkind, Marc J. (1975). "The source code control system". In: *Software Engineering, IEEE Transactions on* 4, pp. 364–370.

Shan, Tony Chao and Winnie W Hua (2006). "Taxonomy of java web application frameworks". In: *e-Business Engineering, 2006. ICEBE'06. IEEE International Conference on*. IEEE, pp. 378–385.

Smith, Alastair G (2001). "Applying evaluation criteria to New Zealand government websites". In: *International journal of information management* 21.2, pp. 137–149.

Strauss, Anselm and Juliet Corbin (1994). "Grounded theory methodology". In: *Handbook of qualitative research*, pp. 273–285.

Wichers, Dave (2010). "OWASP Top-10 2013". In:

Appendix A

Appendices

A.1 Application Breakdown

The following section is a breakdown of the application into its various packages and files.

A.1.1 Java Source

Package: beans

File Name	File Type	Function	No. Lines
beans.xml	XML	Context Component Scan, Datasource Definition	0
dao-context.xml	XML	Hibernate Configuration, Database Exception Translator	0
security-context.xml	XML	Security Configuration, Access Control	0
service.xml	XML	Service Configuration	0

Table A.1: beans package

Package: controllers

File Name	File Type	Function	No. Lines
ErrorHandler	JAVA	Handles application errors	
EventController	JAVA	Handles Event actions	
HomeControler	JAVA	Displays Home Page	
LoginController	JAVA	Manages user login	
MembersController	JAVA	Handles Member actions	
NewsController	JAVA	Handles display and creation of News	
SiteController	JAVA	Displays site pages with static data	
TimetableController	JAVA	Controls the Timetable Creation and Display	
TournamentController	JAVA	Handles Creation and Management of Tournaments	

Table A.2: controllers package

Package: dao

File Name	File Type	Function	No. Lines
EventDAO	JAVA	DAO for IEvent	
LogDAO	JAVA	DAO for ILog	
NewsDAO	JAVA	DAO for News	
RoleDAO	JAVA	DAO for Role	
TimetableDAO	JAVA	DAO for Timetable	
TournamentDAO	JAVA	DAO for Tournament	
UserDAO	JAVA	DAO for User	
UserRowMapper	JAVA	JDBC Row Mapper to handle multiple User objects	

Table A.3: dao package

Package: email

File Name	File Type	Function	No. Lines
Email	JAVA	Create and Send Email Message	
IEmail	JAVA	Interface for Email class	

Table A.4: email package

Package: events

File Name	File Type	Function	No. Lines
Event	JAVA	Defines an event used by the Timetable	
IEvent	JAVA	Interface for Event class	

Table A.5: event package

Package: events.tournaments

File Name	File Type	Function	No. Lines
Tournament	JAVA	Defines an Tournament object	

Table A.6: events.tournament package

Package: logs

File Name	File Type	Function	No. Lines
Log	JAVA	Defines the structure of a system log file	
ILog	JAVA	Interface for Log class	

Table A.7: log package

Package: news

File Name	File Type	Function	No. Lines
News	JAVA	Defines the structure of a News object	

Table A.8: news package

Package: properties

File Name	File Type	Function	No. Lines
jdbc	PROPERTIES	Holds log in values for the JDBC connection	
mail	PROPERTIES	Holds the log in values for the email system	

Table A.9: properties package

Package: reports

File Name	File Type	Function	No. Lines
IReport	PROPERTIES	Interface for Reports	
CSVCreator	PROPERTIES	Creates a CSV file for User Data	

Table A.10: reports package

Package: service

File Name	File Type	Function	No. Lines
EventService	JAVA	Layer between EventController and EventDAO	
LogService	JAVA	Layer between Controllers and LogDAO	
NewsService	JAVA	Layer between NewsController and NewsDAO	
RoleService	JAVA	Layer between Controllers and RoleDAO	
TimetableService	JAVA	Layer between TimetableController and TimetableDAO	
TournamentService	JAVA	Layer between TournamentController and TournamentDAO	
UserService	JAVA	Layer between MemberController and UserDAO	

Table A.11: service package

Package: timetable

File Name	File Type	Function	No. Lines
Timetable	JAVA	Interface for Timetable	
MonaleenTTV1	JAVA	Defines structure and behaviour of Timetable object	

Table A.12: timetable package

Package: users

File Name	File Type	Function	No. Lines
FormValidationGroup	JAVA	Form Validation Class	
PersistenceValidationGroup	JAVA	Hibernate Validation Class	
Grade	JAVA	Defines structure of a Grade object. Used in User class.	
User	JAVA	Defines structure of a User object	
Role	JAVA	Defines structure of Role object, and its attributes	

Table A.13: reports package

A.2 Apache Struts and JSP Pages

layout

File Name	File Type	Function	No. Lines
default	XML	Defines the structure for each JSP page in the application	

Table A.14: struts layout

templates

File Name	File Type	Function	No. Lines
default	JSP	The default JSP page that is used as the template for all others	

Table A.15: templates layout

tiles

File Name	File Type	Function	No. Lines
accessdenied	JSP	Access Denied page	
admin	JSP	Displays administrator page with admin options	
adminAnalysis	JSP	Displays site analytics	
adminEditProfile	JSP	Allows admin to edit user accounts	
alreadyReg	JSP	Error page when attempting to re-register for tournament	
approveMembers	JSP	Admin approve members page	
blockMembers	JSP	Admin suspend members page	
bookingExists	JSP	Error page to handle duplicate bookings	
checkRegistered	JSP	Displays users registered for a selected tournament	
chooseEdit	JSP	Choice for which Timetable to edit	
confirmEdit	JSP	Detailed layout for editing individual slots in Timetable	
contactus	JSP	Contact Us page for application	
content	JSP	Place-holder page for default JSP template	
court	JSP	Displays selected Timetable and available options	
createEvent	JSP	Admin Create Event page	
createmembers	JSP	User Registration page	
createNewRole	JSP	Admin create new User Role	
createNews	JSP	Admin Create News page	
createTimetable	JSP	Admin Create Timetable page	
createTournament	JSP	Admin Create Tournament page	
deleteNews	JSP	Admin Delete News entry	
deleteTimetable	JSP	Admin Delete Timetable object	
deleteTournament	JSP	Admin Delete Tournament object	
displayUsers	JSP	Admin Displays all users to choose which one to edit	
editDetails	JSP	User Edit Profile	
emailSent	JSP	Email Sent Confirmation Page	
error	JSP	Default Error Page. Displays Class Error.	
fillTimetable	JSP	Page that allows admin to create Timetable template for series.	

File Name	File Type	Function	No. Lines
index	JSP	Default Home page	0
footer	JSP	Place-holder page for default JSP template	
header	JSP	Place-holder page for default JSP template	
links	JSP	Displays Site Navigation links	
loggedout	JSP	Logout Confirmation page	
login	JSP	Login Page - Linked to Spring Security	
maps	JSP	Displays Google Maps Location for Club	
members	JSP	Displays Members Address Book for authenticated users	
membership	JSP	Displays Membership Information	
news	JSP	Displays News Page	
profile	JSP	Displays User Profile	
profileUpdated	JSP	Confirmation of profile being updated	
registerSuccess	JSP	Confirmation that user has successfully registered	
resetAllTimetable	JSP	Removes all bookings for a Timetable	
seriesChoice	JSP	Choose which Timetable series to edit/reset	
timetable	JSP	Displays enabled timetables for users	
timetableStatus	JSP	Allows admin to enable or disable timetables	
tournaments	JSP	Displays Enabled Tournaments for Users	
tournamentStatus	JSP	Allows admin to enable/disable/activate/deactivate tournaments	
tournamentSuccess	JSP	Displays success upon successful tournament creation	
viewAllMembers	JSP	Admin View of Members	
viewEvents	JSP	Admin Event Management	

Table A.16: templates layout

A.3 Requirements Elicitation Questions

1. Are you aware there is a site for Monaleen Tennis Club? If so, what do you you it for?
2. What are the best features of the site? What features are needed?
3. If you were not aware, why? Would you use a website if you had known it was there?
4. Would you see any purpose to an online timetable? Benefits? Issues?
5.

6.
7.
8.
9.
10.
11.
12.
13.
14.
15.
16.