Hibernate 'til Spring
Benefits of Spring MVC, Hibernate and Struts for the Development of a Web Application

Chris O'Brien

March 18, 2014

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 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.

1.1.1 General Introduction

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 Objectives

1.3 Scope

1.4 Methodoloy

The methodology chosen as the foundation for this project is the Russo and Graham (1998) design methodology. It focuses on 9 iterative steps, each with feedback loops. The steps are outlined below

- Identification of the problem
- Analysis
- Design of the Application
- Resource Gathering
- Coding
- Testing
- Implementation
- Post Implementation Review and Maintainance

Other methodologies that were examined such as Balasubramanin and Bashian (1997), Siegel (1997), Iskawitz et al (1995) and Cranford-Teague (1998). The pros and cons of these methodologies were examined by Howcroft and Carroll (Howcroft and Carroll 2000), and after an examination of their findings, the Russo and Graham methodology best suited the nature and scale of this project. While the other methodologies are strong, they are geared towards large scale web development projects, or towards document-centred websites, and would not suit this project.(Howcroft and Carroll 2000) Using these as a guide, the following methodology was established.

- Identification of the problem
- Structured Literature Review

- Statement of the FYP Objectives
- Design of the Test Suite
- Development of the Prototype

    Analysis

    Design of the Application

    Resource Gathering

    Design Review

    Coding

    Testing

    Implementation

    Post Implemetation Review and Maintainance

- Emperical Study
- Critical Evaluation of the Results

1.5 Overview of Report

1.6 Motivation

The motivation behind this project for me was to examine, understand and work with software frameworks and methodologies that would be commonly used in industry, and to develop a software application from them. The module, Distributed Systems, touched on some of the tools and technologies, Netbeans and EJB respectively, used in relation to Java Enterprise development, and this formed the foundation of my interest in the area. I felt the FYP was a perfect vehicle to supplement my knowledge of this subject, with particular attention being paid to popular and in demand technologies.

Chapter 2

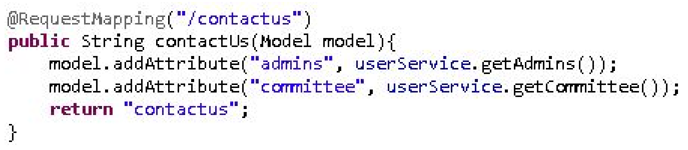
Background

2.1 Technologies

There are a number of components needs 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.

2.1.1 Web Application Framework

The WAF chosen for this project is Spring MVC [Model View Controller]. Shan and Hua defined a WAF as a defined support structure in which other software applications can be organized and developed. (Shan and Hua 2006). Model-View-Controller is a software pattern that facilitates the use of a user interface. The Model manages the behaviour and data of the application. The View will manage the information obtained from the model and display it to the user. The Controller takes user input, such as key strokes, mouse movements or a touch display, and can interact and invoke functionality within the Model and/or View.



```
@RequestMapping("/contactus")
public String contactUs(Model model){
    model.addAttribute("admins", userService.getAdmins());
    model.addAttribute("committee", userService.getCommittee());
    return "contactus";
}
```

Figure 2.1: Contoller adding Model to View

2.1.2 Application Server

2.1.3 Project Management Tool

2.1.4 Database Model

2.1.5 Source Control

2.1.6 Integrated Development Environment

2.1.7 Logging

2.1.8 Web Pages

Structure

Language

2.2 Usability Studies

2.2.1 Case Study: Monaleen GAA Tennis Club

2.2.2 Case Study: Tralee Tennis Club

Chapter 3

Requirements

3.1 Method for Requirements

3.2 Application

3.3 Use Cases

3.4 Functional Requirements

3.5 Non Functional Requirements

Chapter 4

Design

4.1 Key Features

4.1.1 Users

Spring handles security a number of ways. 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.
- ROLE SUSPEND
  - A further restriction placed upon a member.

4.1.2 Tournaments

Events

4.1.3 Timetable

Events

4.1.4 Administration

Logs

Analysis

4.1.5 News

4.1.6 Look and Feel

Chapter 5

Implementation and Testing

5.1 Introduction

This chapters focuses on the implementation of the application, with the focus on the application entities, and how they were configured. Due to the overlapping nature of the application, concepts may be mentioned earlier than explained, but references will be made.

5.2 Application Entities

5.2.1 Users - Persistence

The *User* class represents every user account within the application. (Link to appendice 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, as 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.

```
@Component
@Entity
@Table(name="users")
public class User {
        @Id
        @GeneratedValue
        int id;

        @NotNull(groups={PersistenceValidationGroup.class,
            FormValidationGroup.class})
        @Pattern(regexp=".+\\@.+\\..+", message="This does not appear to be a valid
            email address", groups={PersistenceValidationGroup.class,
            FormValidationGroup.class})
        @Column(name="username")
        String username;

        @Size(min=5, max=45, message="Named must be between 5 and 45
            characters",groups={PersistenceValidationGroup.class,
            FormValidationGroup.class})
        @Column(name="name")
        String name;

        @Column(name="password")
        @Size(min=5, max=15, message="Password must be between 5 and 15
            characters", groups=FormValidationGroup.class)
        String password;

        @Column(name="gender")
        String gender;

        @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})
        @Column(name="contact_num")
        String contact_num;
        //Class truncated. Some repetitive attributes omitted
        //Getters and Setters below here.
```

Table 5.1: User Class Definition and Configuration

The *Entity* and *Table* annotations that 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 *PersistenceValidationGroup*. These are interface classes with no attributes that just 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 5.2.

Constraint Name	Description
Pattern	Ensures the value within the attribute conforms to a regular expression
Min	Ensures the value within the attribute has a minimum length
Max	Ensures the value within the attribute has a maximum length
NotNull	Ensures the value within the attribute does not have a null value

Table 5.2: 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. When using this attribute within the application, such as the creation of a user within a form, the MemberController class applies validation to the user input. 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. 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 *'acb172137243c0b931321d7645dc4b2f5575a30ab48c31c2efb8346385ae0547d11b1d8de333215b'*, 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 5.4.

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.

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
<property name="dataSource" ref="dataSource"></property>
<property name="hibernateProperties">
       <props>
              <prop
                  key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>
       </props>
</property>
<property name="packagesToScan">
       <list>
              <value>users</value>
       </list>
</property>
</bean>
```

Table 5.3: Hibernate SessionFactory Configuration

The Form Validation is provided by the Spring Security file, which will be examined in detail in the Administration Implementation section on page 27. 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, see Figure 5.4 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.

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

//Method from the MembersController class
//This method is responsible for validating the form that users complete to
    register.
@RequestMapping(value = "/register", method = RequestMethod.POST)
public String doRegister(Model model,
@Validated(FormValidationGroup.class) @ModelAttribute("member") User member,
    BindingResult result) {
if (result.hasErrors()) {
       return "createmembers"; // if the result has errors, go back to create page
}
if (userService.exists(member.getUsername())) {
       result.rejectValue("username", "Duplicate Key",
       "This email address has already been used");
       return "createmembers";
       //if the email address already exists, return with this message.
}
       else {
              try {
                     member.setAuthority("ROLE_MEMBER");
                     userService.create(member);
                     return "registerSuccess";
                     //successful creation of member
                     } catch (Exception e) {
                            return "error";
                     }
              }
}
```

Table 5.4: User Registration Form

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 5.5 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.

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

Table 5.5: 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, shown in Figure 5.6. Spring provides a class that allows passwords to be encoded, and the bean for this class is defined within a security-context file.

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

Table 5.6: 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 5.7. Once the password is encoded (Line 4, 5.7), 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.

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

Table 5.7: 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 GAA Tennis Club, it was desirable that the timetable retain the ability to have some dynamic features, not pre-set features that were not user definable. One issue this raised was how to handle a varying number of slots in each day. If a user could defined 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 5.8.

```java
@Entity
@Component
@Table(name = "timetable")
public class MonaleenTTV1 implements Timetable {
	@Id
	@GeneratedValue
	private int id;
	/**
	* Other Attributes Here
	**/

	@ElementCollection
	@CollectionTable (name = "monday", joinColumns=@JoinColumn(name="id"))
	private List<String> monday;

	@ElementCollection
	@CollectionTable (name = "tuesday", joinColumns=@JoinColumn(name="id"))
	private List<String> tuesday;

	@ElementCollection
	@CollectionTable (name = "wednesday", joinColumns=@JoinColumn(name="id"))
	private List<String> wednesday;

	@ElementCollection
	@CollectionTable (name = "thursday", joinColumns=@JoinColumn(name="id"))
	private List<String> thursday;

	@ElementCollection
	@CollectionTable (name = "friday", joinColumns=@JoinColumn(name="id"))
```

```
        private List<String> friday;

        @ElementCollection
        @CollectionTable (name = "saturday", joinColumns=@JoinColumn(name="id"))
        private List<String> saturday;

        @ElementCollection
        @CollectionTable (name = "sunday", joinColumns=@JoinColumn(name="id"))
        private List<String> sunday;

        //getters and setters
}
```

Table 5.8: 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. 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 5.9, 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 display the next number of timetables. In this example, 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	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 5.9: Timetable Core Database Table

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 5.10: Timetable Storage Table for Monday Collection

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.

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

Table 5.11: Hibernate Save Timetable

5.2.3 Tournaments and 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.

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 5.12, and to ensure that no ambiguity was present when dealing with multiple tournaments and possible multiple timetables. It was also necessary for this to be self managing, in order to minimize administrator 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);
}
```

Table 5.12: Tournament Creation

Deleting a tournament needed similar logic, as displayed in Figure 5.13. 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";
	}
```

Table 5.13: 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.

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 5.14) 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.

```
@RequestMapping(value = "/saveEvent", method = RequestMethod.POST)
public String saveEvent(Model model, @ModelAttribute("event") Event e,
	BindingResult result) {
	if (eventService.getEventById(1).equals(null)){
		e.setName("Free Court");
		e.setAuthor(userService.emailToName(SecurityContextHolder
		.getContext().getAuthentication().getName()));
		e.setEnabled(true);
		eventService.createEvent(e);
```

```java
    }
    logger.info("Event Save Method....");
    if (result.hasErrors()) {
        return "createEvent";
    }
    if (eventService.exists(e.getName())) {
        result.rejectValue("name", "Duplicate Key",
        "This Event Name has already been used.");
        return "createEvent";
    } else {
        e.setAuthor(userService.emailToName(SecurityContextHolder
        .getContext().getAuthentication().getName()));
        e.setEnabled(false);
        eventService.createEvent(e);
        return viewEvent(model);
    }
}
```

Table 5.14: Event Creation

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 5.15. This implementation allows the administrator to pre-create events for later use without overcrowding the timetable.

```java
@RequestMapping("/changeEventStatus")
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);
    }
}
```

Table 5.15: Change Event Status

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 4.1.1. 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 5.16, 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>
```

Table 5.16: Spring Role Configuration

The security within the application is controlled by the *security-context.xml* file, which used Expression Based Access Control in order to restrict site access to relevant roles. The base class used within this application is *SecurityExpressionRoot*

5.3 Model View Controller

5.3.1 Controller Layer

5.3.2 Models within the Spring MVC Framework

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 5.17, takes one parameter: a location of the template that the default tile, and its subsequent children, will use.

```
<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>
```

Table 5.17: 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 5.18.

```
<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>
```

Table 5.18: 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 5.19 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"%>
```

Table 5.19: 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, and user actions based on the information contained within. The example below is taken from the Timetable display page from the application.

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 clas

Table 5.20: Model Attributes

```java
@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";
}
```

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>
```

Table 5.21: 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 5.22: 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.

```
log4j.rootLogger=INFO, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.SimpleLayout
log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE
log4j.logger.org.springframework.security=DEBUG
```

Table 5.23: 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.

```java
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;
}
```

Table 5.24: Logger Usage within UserDAO.class

5.5 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 to produce different development environments, such as production and testing environments. The configuration of this file within the application is shown in Figure 5.25

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

Table 5.26: Hibernate Configuration

```xml
<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>
```

Table 5.25: 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 configure the Service classes.

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

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>
```

5.6 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.

```java
@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
}
```

Table 5.27: 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.

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

Table 5.28: 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".

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

Table 5.29: 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.

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

Table 5.30: JUnit Create and Size Test

5.7 Conclusion

Chapter 6

Software Quality

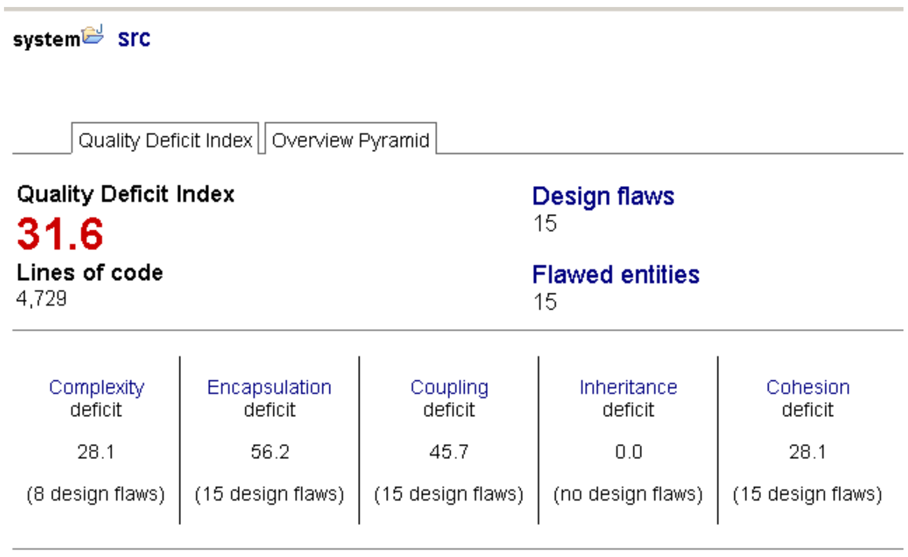
6.1 Application Summary

The following is a summary of the various metrics of the application. A detailed breakdown by package in contained within the appendices.

Metric	Total	Mean	Std. Deviation	Maximum
McCabe Cyclomatic Complexity	n/a	1.262	.862	8

Table 6.1: Application Metrics

6.2 Software Quality Tools and Visualisations



system src

Quality Deficit Index | Overview Pyramid

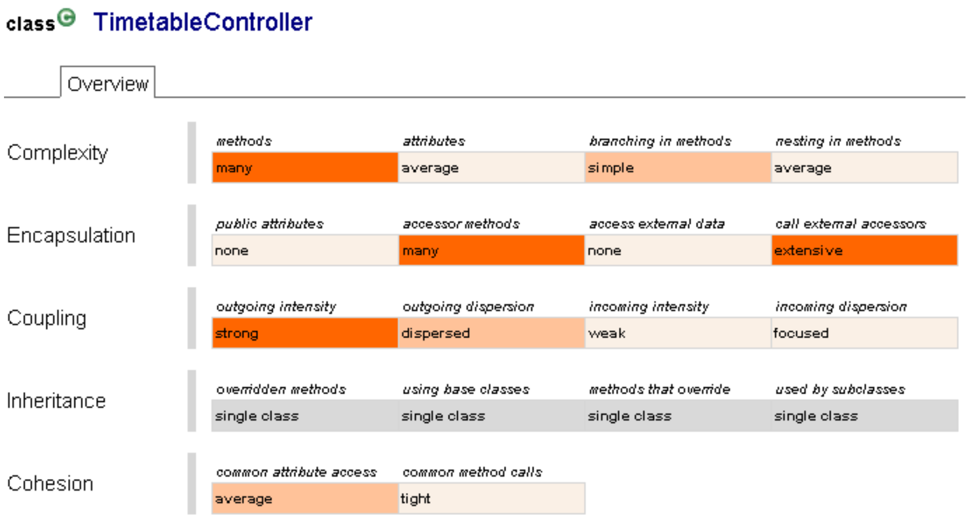
Quality Deficit Index
31.6
Lines of code
4,729

Design flaws
15

Flawed entities
15

Complexity deficit	Encapsulation deficit	Coupling deficit	Inheritance deficit	Cohesion deficit
28.1	56.2	45.7	0.0	28.1
(8 design flaws)	(15 design flaws)	(15 design flaws)	(no design flaws)	(15 design flaws)

Figure 6.1: Pre Re-factoring using Infusion

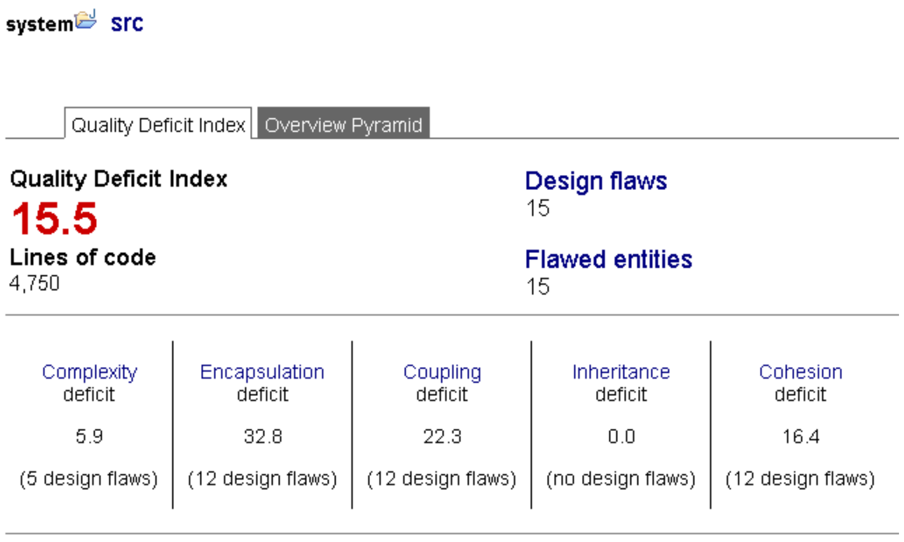


class TimetableController

Overview

Complexity	*methods* many	*attributes* average	*branching in methods* simple	*nesting in methods* average
Encapsulation	*public attributes* none	*accessor methods* many	*access external data* none	*call external accessors* extensive
Coupling	*outgoing intensity* strong	*outgoing dispersion* dispersed	*incoming intensity* weak	*incoming dispersion* focused
Inheritance	*overridden methods* single class	*using base classes* single class	*methods that override* single class	*used by subclasses* single class
Cohesion	*common attribute access* average	*common method calls* tight		

Figure 6.2: Example of Class 'Bad Code Smell' Breakdown using Infusion


system src

Quality Deficit Index | Overview Pyramid

Quality Deficit Index
15.5
Lines of code
4,750

Design flaws
15

Flawed entities
15

Complexity deficit	Encapsulation deficit	Coupling deficit	Inheritance deficit	Cohesion deficit
5.9	32.8	22.3	0.0	16.4
(5 design flaws)	(12 design flaws)	(12 design flaws)	(no design flaws)	(12 design flaws)

Figure 6.3: Post Re-factoring using Infusion

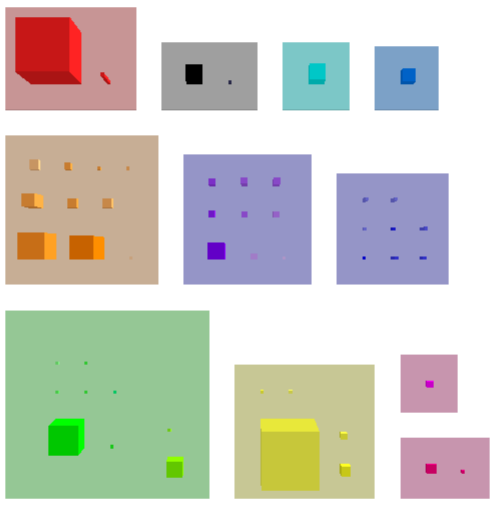


Figure 6.4: CodeCity 2D Visualisation of Application

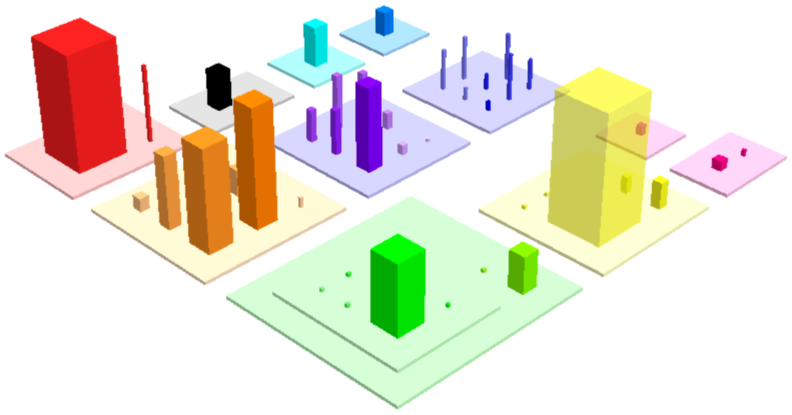


Figure 6.5: CodeCity 3D Visualisation of Application

6.3 Sample Refactorings

Chapter 7

Evaluation

7.1 Usability

Apply frameworks from Chapter 2 to finished product

7.2 Architectural Quality

Performance, Security and Evolution Support

7.3 Process Metrics

Chapter 8

Conclusions

Bibliography

[Martin James] Martin, James (1983) Managing the Database Environment J. Martin, MA: Prentice Hall PTR.

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	N I
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 Li
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
index	JSP	Default Home page
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