JStack:
A Rapid Application Development Solution for Java MVC Applications

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in

Computer Science

by

Colin J R King

Dr. Sergiu Dascalu/Thesis Advisor

December, 2010
Copyright © 2010 by Colin J R King

All rights reserved.
Abstract

Rapid application development (RAD) is used to save time in the development process of new applications. While RAD can be used for a variety of applications, it is especially useful in database-driven applications designed on a large scale, where repetitive code can severely lengthen the development time of the project. The effectiveness of a RAD tool can be affected by architectural factors, such as programming languages and frameworks.

This thesis describes a Java framework known as Spring and explains how Spring is well suited for RAD. Spring is a popular extension to Java Enterprise Edition, a framework for developing multi-tier, distributed applications. Its simplicity leads to faster development of software applications, and easier integration with RAD and other tools. Scaffolding is a process in which data is exposed and manipulated by HTTP clients in a distributed, networked environment. Spring is complemented by an existing RAD tool named Roo, which is able to generate scaffolded applications with fully-functional web interfaces and database-driven classes that can be modified to suit the domain in which they operate. Roo's most prominent setback is the lack of a graphical user interface. Without it, Roo is less likely to attract new developers, more likely to slow the development processes, and more likely to produce error conditions that are difficult to correct.

This thesis presents a software application, named JScaffold, that addresses this
and several other problems in Roo, and provides an enhanced user experience where
users can be comfortable in developing their applications. The results of JScaffold,
supported by several developers familiar with database-driven applications, show that
this software tool can improve the usability of the Roo tool, making Spring and Java a
more attractive platform for RAD. Its functionality resembles the features of other RAD
tools which generate scaffolded applications. The results show that with JScaffold,
Spring-based RAD becomes more effective and attractive for application developers.

**Keywords:** Java, Java EE, RAD, Spring Framework, Roo, JScaffold
Acknowledgements

I would like to thank my advisor, Dr. Sergiu Dascalu, who on multiple occasions allotted time out of his busy schedule to answer my questions and pointed me in the right direction.

To my committee members, Drs. Yantao Shen and Mehmet Gunes, for their valuable feedback on this thesis.

To my single largest contributor of technical knowledge and experience regarding multi-tier applications, RAD, and database-driven development, Theo McCormick.

To my reviewers and fellow professionals, whose feedback were probably the most valuable artifacts from this entire process: Bei Yuan, Chris Leonard, Hema Andal Jayaprakash Narayanan, John Bond, Jonathan Sass, and Theo McCormick.

To my friends and family whose encouragement and support showed me that no goal was unreachable when I put my mind to it.

To the UNR faculty who provided me with the foundation of wisdom that allowed this process to begin.

And finally, to my wife Cindy who I could not have done any of this without.

Thank you all.
# Table of Contents

1 INTRODUCTION ............................................................................................................. 1

2 BACKGROUND – Java EE ................................................................................................. 4
   2.1 Introduction ........................................................................................................ 4
   2.2 n-tier Applications .............................................................................................. 4
   2.3 Java EE Features .................................................................................................... 5
   2.4 JNDI ....................................................................................................................... 7
   2.5 Resource Management ......................................................................................... 8
   2.6 Servlets ................................................................................................................ 9
   2.7 JSP ....................................................................................................................... 13
   2.8 RMI .................................................................................................................... 16
   2.9 EJB ..................................................................................................................... 18

3 BACKGROUND – Spring ................................................................................................. 22
   3.1 Introduction ........................................................................................................ 22
   3.2 Inversion of Control ........................................................................................... 22
   3.3 Resource Management ....................................................................................... 27
   3.4 JDBC .................................................................................................................. 28
   3.5 Aspect-oriented Programming ............................................................................. 32
   3.6 Remote Objects .................................................................................................. 35
   3.7 Spring MVC ........................................................................................................ 38

4 PROBLEM DEFINITION .................................................................................................. 41
   4.1 Observed Challenges ......................................................................................... 41
   4.2 Rapid Application Development ....................................................................... 42
   4.3 Spring Roo ......................................................................................................... 43

5 PROPOSED SOLUTION .................................................................................................. 54
   5.1 Overall Design ...................................................................................................... 54
   5.2 Development Process ........................................................................................ 55
   5.3 Target Users ....................................................................................................... 56
   5.4 Features ............................................................................................................. 57

6 SOFTWARE MODEL ..................................................................................................... 58
   6.1 Introduction ......................................................................................................... 58
   6.2 Architecture of Roo-generated MVC applications .............................................. 58
   6.3 Architecture of JScaffold .................................................................................. 60
   6.4 Roo commands implemented ............................................................................ 64

7 PROTOTYPE/RESULTS ................................................................................................. 71
   7.1 Introduction ......................................................................................................... 71
   7.2 Interface ............................................................................................................... 71
   7.3 Address Book Demo ........................................................................................... 92
   7.4 Peer Review ....................................................................................................... 97
List of Tables

Table 1: JScaffold to Roo mappings................................................................. 72
Table 2: Responses from peer review........................................................... 101
List of Figures

Fig. 1: Roo startup screen and example of 'hint' command ................................................. 44
Fig. 2: Iron Speed Designer application wizard [10] .......................................................... 46
Fig. 3: Iron Speed Designer preview [10] .......................................................................... 47
Fig. 4: Starting a Roo project .............................................................................................. 48
Fig. 5: Using the 'hint' command during project ................................................................ 49
Fig. 6: Generating stub database parameters ....................................................................... 50
Fig. 7: Defining fields in an entity ....................................................................................... 51
Fig. 8: Directory listing of target application ....................................................................... 52
Fig. 9: Modified MVC architectural pattern ........................................................................ 59
Fig. 10: Class diagram of one-to-many relationship ............................................................ 60
Fig. 11: Typical program flow ............................................................................................. 61
Fig. 12: Use cases ................................................................................................................ 62
Fig. 13: Internal representation of the target Roo application .............................................. 63
Fig. 14: Jscaffold welcome screen ..................................................................................... 73
Fig. 15: Database configuration with some example settings configured ............................. 74
Fig. 16: Test connection result ............................................................................................ 74
Fig. 17: Internal database used by the Roo demo ................................................................. 75
Fig. 18: Empty entities/fields tab ......................................................................................... 76
Fig. 19: Adding a new entity to the project model ................................................................ 76
Fig. 20: GUI response to renaming an entity ....................................................................... 77
Fig. 21: Selecting a Roo type in the fields interface ............................................................... 78
Fig. 22: Configuring type-specific attributes for the String type ........................................ 79
Fig. 23: Adding multiple fields to an entity ........................................................................ 79
Fig. 24: Adding a second entity to a project ....................................................................... 80
Fig. 25: Creating a derived entity class ................................................................................ 80
Fig. 26: Defining an enumerated type ................................................................................ 81
Fig. 27: Empty controllers tab ............................................................................................. 82
Fig. 28: GUI response to automatically generated controllers .............................................. 83
Fig. 29: Empty properties tab ............................................................................................. 84
Fig. 30: Adding a property to a properties file .................................................................... 85
Fig. 31: Empty messaging tab ............................................................................................. 86
Fig. 32: Example destination and recipients defined ............................................................ 86
Fig. 33: Empty code generator output window ..................................................................... 87
Fig. 34: Top half of code generator output displaying the project model summary ............. 88
Fig. 35: Bottom half of code generator output displaying the Roo script ............................. 89
Fig. 36: Roo output displayed in a terminal emulator ......................................................... 90
Fig. 37: Roo compile output ............................................................................................... 91
Fig. 38: Target application welcome page .......................................................................... 93
Fig. 39: Adding a new contact ............................................................................................ 94
Fig. 40: Viewing a contact ................................................................................................. 95
Fig. 41: Listing contacts in a table view.......................................................................................... 96
Fig. 42: Creating an OrgPerson object with enum values shown........................................... 97
Fig. 43: Peer review survey (page 1)........................................................................................... 99
Fig. 44: Peer review survey (page 2).......................................................................................... 100
Fig. 45: Grails target application welcome page........................................................................ 107
Fig. 46: Listing entities in a table in the Grails target application........................................... 108
Fig. 47: Adding a new entity in the Grails target application..................................................... 109
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4GL</td>
<td>4th Generation Language</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
</tr>
<tr>
<td>ASP</td>
<td>Active Server Pages</td>
</tr>
<tr>
<td>CGI</td>
<td>Common Gateway Interface</td>
</tr>
<tr>
<td>CORBA</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>CRUD</td>
<td>Create, Read, Update, and Delete</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>DAO</td>
<td>Data Access Object</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>EAR</td>
<td>Enterprise Archive</td>
</tr>
<tr>
<td>EE</td>
<td>[Java] Enterprise Edition</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBeans</td>
</tr>
<tr>
<td>GNU</td>
<td>GNU is Not Unix</td>
</tr>
<tr>
<td>GPL</td>
<td>[GNU] General Public License</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>ISD</td>
<td>Iron Speed Designer</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>IoC</td>
<td>Inversion of Control</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JAX</td>
<td>Java API for XML</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Messaging Service</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>JPA</td>
<td>Java Persistence API</td>
</tr>
<tr>
<td>JSF</td>
<td>JavaServer Faces</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>JTA</td>
<td>Java Transaction API</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>ORM</td>
<td>Object Relational Mapping</td>
</tr>
<tr>
<td>PERS</td>
<td>Public Employees' Retirement System [of Nevada]</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP: Hypertext Preprocessor</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Object</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid Application Development</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SE</td>
<td>[Java] Standard Edition</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>VB</td>
<td>Visual Basic</td>
</tr>
<tr>
<td>WAR</td>
<td>Web Archive</td>
</tr>
<tr>
<td>WS</td>
<td>Web Services</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transformation</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

This thesis addresses the issues of rapid application development for the Java platform, a popular portable virtual machine and language combination created in 1990 by Sun Microsystems. The Java platform incorporates numerous components, APIs, and standards ranging from concerns such as database libraries to graphical user interfaces [6]. Since its inception, Java has added support for various programming paradigms including client/server, web programming, to full n-tier multi-user applications [11]. For this reason, it has become popular with large organizations for in-house software development [2][13].

Nevada PERS (Public Employees' Retirement System) is a state-sponsored, non-profit organization that manages the retirement accounts for all state employees. The majority of their staff is located in Carson City, NV. Most of their 70+ users interact daily with an internal business system written in Java. This system is a good example of n-tier applications, as it runs on 3 tiers and is distributed across multiple servers. The application contains approximately 5 million lines of source code and is the result of a conversion from an identical application written in a now-obsolete 4th generation language. The application uses the Spring Framework, a popular enhancement/substitute for Java EE (Enterprise Edition), and the Apache Tomcat web server, as its platform. It
also uses Microsoft SQL to store its persistent data and a separate Tomcat server to
operate its public website, and employs techniques such as batch processing and
workflows for much of its back-office functionality.

This thesis is inspired by the work that I and others have done on this business
system. It is heavily dependent on data, which consists of hundreds of SQL tables and
approximately 80 gigabytes of raw data. Java EE and the Spring Framework are specially
designed for such environments. With use of the n-tier architecture, applications such as
this can efficiently distribute resources across multiple servers and clients, reducing
maintenance costs and increasing productivity [2][13].

The type of application written for Java EE or the Spring Framework typically
includes a significant amount of non-domain logic code known as skeleton or boilerplate
code. When written by hand, this type of code can cost developers valuable time in
writing their application, and often does not add any benefit to the application other than
to utilize the features of the chosen framework [3][5][14]. Some languages, known as 4th
generation languages, address the specific limitation that others have in the way of
relational database support by adding this support to the language itself. This is usually
accompanied by a tradeoff in versatility, but often suits the needs of organizations that
make heavy use of such features.

An alternative to these approaches is the use of code generators, or specifically,
rapid application development tools. These tools provide programmers with a shortcut to
developing their application by generating skeleton and boilerplate code automatically [3]
[5][14]. This involves gathering information from the programmer and converting it into
meaningful source code optimized for the chosen framework. Such tools can significantly reduce development and maintenance costs without the tradeoffs of 4th generation languages.

This thesis explains how the Spring Framework is a good candidate for rapid application development. It identifies the necessary components from Java EE and Spring that are used for RAD, and addresses the issues behind an existing RAD tool designed specifically for the Spring Framework, named Roo. Its focus is on a key usability issue of Roo, the console-based interface, and provides a solution to this and other usability issues in the form of a graphical frontend application to Roo. This solution is named JScaffold. To demonstrate the improvement in usability that JScaffold offers, this thesis includes an example application generated using JScaffold and compares this process with using Roo alone.

The remainder of the thesis is broken up into chapters as follows:

- Chapter 2: Background information about Java EE.
- Chapter 3: Background information about the Spring Framework.
- Chapter 4: Challenges observed while using Spring and an introduction to RAD.
- Chapter 5: A proposed solution to the problems raised in chapter 4.
- Chapter 6: The software model for a prototype, JScaffold, of the solution.
- Chapter 7: JScaffold details, an example application, and a peer review
- Chapter 8: Introductions to related works Iron Speed Designer and Grails
- Chapter 9: Future work for JScaffold
- Chapter 10: Conclusions
2 BACKGROUND – Java EE

2.1 Introduction

Java Enterprise Edition, also known as Java EE or J2EE, exists as a set of specifications among many other specifications that are part of the Java Community Process (JCP) [1][11]. These specifications date back as early as 1998, when a public review and committee process was formed to add enhancements to the Java platform. Java EE is a framework that can be used to create n-tier, scalable, and distributed applications for use in large organizations, often where multiple users are accessing the same data concurrently [11]. These are described as enterprise-level applications. Without a framework, developing such applications can be costly and time-consuming. Java EE attempts to meet the needs of developers writing applications for this type of environment.

2.2 n-tier Applications

An n-tier model, or at minimum a 3-tier model, can supply many of the powerful features required in a large multi-user environment. This model incorporates concepts such as security, modularity, distribution, load balancing, and compatibility. This model generally assumes a networked environment containing one or more servers and clients,
each of which may use different platforms and languages, as opposed to the mainframes of the previous generation. A web-based interface may be included, both for internal and external clients, but is not always used. Indeed, Java EE supports the use of traditional desktop GUI applications as clients to its n-tier model [11][13]. The 3-tier model closely resembles the Model-View-Controller (MVC) architectural pattern, except that communication between the model and view is forced to act through the controller; no direct communication between these two components is allowed [7][14][20]. While this restriction may incur a performance cost, it adds security and other features to the model.

The three tiers are known as the Data Tier (Enterprise Information System or EIS Tier in EE), Application Tier (also Middle, Business, or Logic), and Presentation Tier. The Data Tier consists of information storage and exists outside of the application server software, usually implemented as a relational database. The Application Tier contains the majority of the applications logic and processing functions and serves as a link between the client and the data. The Presentation Tier can be represented by a GUI application or website and is used to interact with the users. One could also describe the presentation and application tiers as a form of client and server, though in the case of a web presentation, the actual client (a web browser) exists outside of the scope of this model [11].

### 2.3 Java EE Features

Java EE is a large collection of individual components, most of which exist in specifications rather than specific implementations from Sun [11]. The implementation of each of these components is left to the individual application server. Therefore, unlike
Java SE, it is impossible to download and run a complete version of the Java EE platform without an application server. Sun bundles its own reference implementation of an EE application server named Sun Java System Application Server\(^1\), also known by its open source name GlassFish [11]. Other popular servers include Red Hat's JBoss AS, IBM's WebSphere, and Adobe's ColdFusion, each varying in licensing from closed/proprietary to GPL and other open source licenses. These servers populate the middle tier of the 3-tier architecture, similar to a controller in the MVC pattern, and the web tier, which can be considered as either part of a view or a controller depending on how it is used. Another server, called Apache Tomcat, is a web server and operates as part of the presentation tier in its default installation, but can be extended using additional frameworks such as Spring to act as a full middle-tier server [1][24]. The following is a brief list of components included in the GlassFish server [11]:

- An EJB (Enterprise JavaBeans) container, used to encapsulate business classes using the EJB specification, in which the majority of the application's logic is implemented.
- An XML, DOM, and XSLT processing library.
- An implementation of the Java Messaging Service, used to pass messages between EJBs and/or tiers.
- An implementation of the Java Transaction Architecture, used to ensure transaction-safe operations in database programming where transaction safety is required.

---

\(^1\) With the recent acquisition of Sun by Oracle Corporation, many of the Sun products have been re-branded under the Oracle name. For historical reasons, the original Sun names will be used throughout this thesis.
• An interface for configuration and management of web services, which are special Java classes used to handle remote procedure calls using HTTP as a transport and XML as a language.

• Interfaces and a configuration GUI for JDBC (database) connections.

• A low-level remote procedure call library using protocols such as RMI and CORBA IIOP.

Also, both GlassFish and Apache Tomcat support standalone web applications using the following technologies: [1][11]

• JavaServer Pages, an extension to HTML and XML containing embedded Java code in a similar style to PHP and ASP.

• The Servlet interface.

• Support for filter classes which implement a unique form of aspect-oriented programming on the web.

2.4 JNDI

In Java EE, resource and dependency injection are available from the server using Java Naming and Directory Interface (JNDI). The original JNDI specification supports several directory services including Novell, LDAP, and NIS. A directory is described as an association of names with objects and attributes. The list of attributes defined in NIS is fixed, whereas attributes may vary depending on a schema in LDAP. LDAP also supports a tree structure where objects may be grouped according to an organizational hierarchy.
SQL-style tables are not supported by JNDI and are left to other technologies such as Java Database Connectivity (JDBC). JNDI specifies only an interface necessary to access all of these types of directories, and like JDBC, does not necessarily provide individual drivers to the directory implementations themselves [11].

Application servers such as Tomcat provide their own JNDI implementations to manage dependencies and resources. For web applications, resources may be specified in a standard way in the application's web.xml file. Items in this file are matched by name with resources specified in the server's configuration, each server having its own way of configuring these resources. The server then provides an implementation of JNDI and populate it with these resources, allowing the application to acquire references to these objects by name instead of instantiating them from within the application [1][11].

2.5 Resource Management

Java EE programming practices make use of the resource concept in order to provide several features to the application. In the case of JDBC, this allows a system administrator to configure database connections without modifying the application's source code or its embedded configuration files. Some application servers, such as GlassFish, provide a web-based interface to perform this sort of configuration. The web interface also allows an administrator to test the settings outside of the application. In addition, this system grants global scope to these objects either inside a web application or across the entire server. This provides the necessary scoping for objects that act in a shared fashion, such as JDBC connection pools, without the need for direct access of
static objects by an application. Such objects can also be created using a singleton or factory pattern, where the implementation of these patterns are part of the server itself [1][7][11][16].

2.6 Servlets

One of Java EE's links between application and HTTP requests resides in the Servlet interface. Specifically, this is `javax.servlet.Servlet` and `javax.servlet.http.HttpServlet`. There are other types of servlets in Java, but for the handling of HTTP requests, an application developer will want to extend from the HttpServlet class. The servlet is similar to a CGI application, but with several advantages over the usual CGI-based web service. In a Java web application or EE application containing at least one web component, the servlets are configured using an XML deployment descriptor and instantiated at the beginning of the application's lifetime. This is either upon server startup if an application is already deployed, or upon deployment of the application if it is not. Likewise, the servlet's lifetime ends upon server shutdown or removal of the web application. The servlet interface provides methods `init()` and `destroy()` in order to provide graceful startup and shutdown of the servlet object [1][11][25].

Traditional CGI applications are instantiated by the operating system as separate processes for each HTTP request. Although it is possible to shift business logic into a separate persistent layer, the amount of overhead required for the management of heavyweight processes is considerable and applications may suffer significant
performance loss in heavily trafficked websites. To address this and other CGI-related problems, additional layers were written into the server. Examples of such layers are mod_perl, mod_php, and FastCGI. In a Java web application, the servlet object is persistent in memory and its control over other resources, and the web server will automatically manage threading and named resources [11].

An HTTP servlet class is expected to override at least one of the following methods: doGet(), doPost(), doPut(), doDelete(), doTrace(), and doOptions(). These methods correspond to the possible HTTP requests. The HEAD method, which returns only header information for a document that would otherwise be retrieved with GET, will also call the servlet's doGet() method, but all non-header data written to the output object is ignored. The “do” methods all contain two parameters: HttpServletRequest and HttpServletResponse. In the HttpServletRequest object, it is possible to get information about the request being made to the HTTP server. This information includes resource identifier, GET or POST parameters, user agent string, remote IP address, proxy headers, and more. This object also contains the getSession() method, which will return a session object capable of storing named objects and variables that can be made available to subsequent calls made from the same HTTP client. The session is managed by the servlet container and usually associated with a unique identifier that is stored in a cookie on the HTTP client. The HttpServletResponse object provides two main features. One is an object to modify outgoing header information, and another is to access the HTTP output body as a stream, allowing stream-based I/O calls to produce the resulting document. As with other HTTP frameworks, the servlet library requires all header information to be
written first, attempting to write an additional header item after any part of the document is sent will produce an exception. Also, the servlet class is not required to produce any header information at all, the library will automatically produce default headers to the client. Unless explicitly specified in the response object, any valid servlet call will return HTTP status code 200, an invalid servlet call will return 404, and a servlet call that throws an exception will return 500 [1][11].

The following code demonstrates some of the servlet's abilities:

```java
public class ExampleServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/plain;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("Example servlet output...");
            out.println("Your request URI: "+ request.getRequestURI());
            out.println("Your query string: "+ request.getQueryString());
            out.println("Your method: "+ request.getMethod());
            out.println("Your host: "+ request.getRemoteHost());
            out.println("Servlet path: "+ request.getServletPath());
            out.println("Path info: "+ request.getPathInfo());
            out.println("Path translated: "+ request.getPathTranslated());
            out.println();
            out.println("Request attributes...");
            Enumeration e = request.getAttributeNames();
            while (e.hasMoreElements()) {
                String name = (String) e.nextElement();
                String v = request.getAttribute(name).toString();
                out.println(name + ": "+ v);
            }
            out.println();
            out.println("Input parameters below...");
            e = request.getParameterNames();
            while (e.hasMoreElements()) {
                String name = (String) e.nextElement();
                String[] v = request.getParameterValues(name);
                if (v.length > 0) {
                    out.println("" + v[0]);
                    for (int i = 1; i < v.length; i++) {
                        out.println("" + v[i]);
                    }
                }
            }
        }
    }
}
```
out.print("", " + v[i]);
}
out.println();
}
}
out.println("Headers below...");
e = request.getHeaderNames();
while (e.hasMoreElements()) {
    String name = (String) e.nextElement();
    out.println(name + ": " + request.getHeader(name));
}
out.println();
out.println("Session variables below...");
e = request.getSession().getAttributeNames();
while (e.hasMoreElements()) {
    String name = (String) e.nextElement();
    out.println(name + ": " + request.getSession().getAttribute(name).toString());
}

// Demonstrates a simple session variable that is changed
// each time the page is accessed.
Integer counter = (Integer)
request.getSession().getAttribute("counter");
if (counter == null) {
    counter = new Integer(0);
    request.getSession().setAttribute("counter", counter);
} else {
    counter = new Integer(counter.intValue() + 1);
    request.getSession().setAttribute("counter", counter);
}

} finally {
}

The servlet class allows an application to generate simple HTTP responses as well as complex responses, even granting some low-level access to the underlying protocol when such access is required by the application. The servlet container provides almost all of HTTP's functionality to the application. The servlet container has a few responsibilities on its own. The first is matching URLs with calls to individual servlets, since it is possible to have more than one servlet running in a web application. This feature is
controlled through an XML deployment descriptor supplied with the web application. The descriptor will specify a URL pattern and match it with the fully qualified name of the servlet class that will receive requests to that pattern. The servlet container then instantiates the servlet class when the web application is deployed and maps all URLs matching the pattern to that instance. Another responsibility of the servlet container is to enable authentication and security parameters for a given URL pattern. Each application server implements security in its own way, but the configuration with respect to the web application is standard and part of the XML descriptor. The descriptor may also contain initialization parameters for each of the servlets, which is done using the JavaBeans setter pattern. Finally, it is possible to add one or more filter classes to a web application that can intercept and, optionally, rewrite information that is sent to and received from the servlet classes. This adds a very AOP-like (aspect-oriented programming) scheme to web programming [1][11][25].

2.7 JSP

Although servlets may be used to implement the “view” portion of an MVC application, this functionality is usually reserved for the “controller” portion. As seen in the example above, all output is written to the HTTP response body using stream I/O. While this method is as efficient as writing a cached text file to the HTTP stream, it makes a document difficult to visualize to the programmer and does not allow document-based editors to assist the programmer. Therefore, it is recommended that the “view”
portion as well as text documents be implemented using a friendlier language, such as JavaServer Pages (JSP) or JavaServer Faces (JSF) [1][11][21].

JavaServer Pages is an extension supported by EE compliant web containers, or the same applications capable of hosting servlets. JSP does not add new classes or interfaces to the Java library, other than classes written specifically for the web container software that implement support of JSP in the server, and does not modify the Java language specification itself. Instead, JSP files are made up of Java code mixed with HTML (or some other document-specific) markup. This format is similar in design to PHP and ASP. However, while PHP is considered an interpreted language, JSP files are compiled into Java bytecode before the page may run. However, before the JSP is compiled, it is fed into a preprocessor that will convert it into regular Java source code. The result of the preprocessor is an individual class definition that closely resembles a servlet class. The name of this class is generated in such a way that it reduces the possibility of name conflicts with other servlets and classes. The web container then automatically compiles this servlet class and maps all requests to the JSP location to the servlet. The application developer does not need to add any additional entries to the XML deployment descriptor of their web application in order to support JSP [1][11][25].

Depending on the JSP tags used, code may be placed in one of two locations. JSP designers are allowed to define class-level methods and fields by enclosing them in a specific JSP tag: <!-% ... %> . As with servlets, the programmer must design any variable access in a thread-safe way since there is one instance of the JSP object created by the web container. The other JSP tag used to enclose Java code is in the form: <% ... %>. This
form resembles PHP, meaning that the code is executed in this specific point in the output of the page. It is possible to stall the output of a page by inserting Java code that takes a significant amount of time to execute [1][11].

When the JSP file is preprocessed into Java source code, the text and markup residing outside of the JSP tags are inserted into the code as calls to the response output stream of the servlet. For example, take the following JSP segment:

```jsp
<h1>Hello World!</h1>
<% out.print("You have " + msgBean.getNumMessages() + " message(s)<br/>
" ); %>
```

When preprocessed, the resulting Java code will resemble:

```java
out.print("<h1>Hello World!</h1>
");
out.print("You have " + msgBean.getCount() + " messages<br/>
");
```

This design allows markup, text, and code to reside next to each other in the same file. It also allows certain HTML editors to seamlessly integrate executable code directly into a page. Another feature that JSP shares with PHP is the ability to interrupt the syntax of the Java code with text and then continue the syntax after the text, such that the resulting preprocessed Java code still makes sense. For example, it is possible to iterate through a list of names in the following way:

```jsp
for(String name: names) { %>
<td>Name: <%= out.print(name); %></td>
<% } %>
```

This format may seem absurd for a small amount of code as shown in the example, but can help improve the overall cleanliness of the source code as it further reduces the number of explicit calls to out.print(). Finally, an additional tag format is supported by JSP that allows simple inline variable and method access. When a variable identifier or method call is surrounded by the `<%= ... %>` tag, it is evaluated and inserted directly into
the output document. This further simplifies the code and allows the programmer to see a cleaner representation of what the resulting document will look like [1][11].

2.8 RMI

The ability to make a remote procedure call, or RPC, is essential to multi-tier and distributed applications such as those written using the Java EE framework. In Java, RPC is supported in multiple ways, each of which meet various needs of the client. For example, CORBA/IIOP may be used across different architectures and programming languages [11][13]. SOAP and XML-RPC use the popular HTTP protocol as an underlying transport, allowing for greater compatibility with restrictive network environments [6]. Java itself provides its own framework for remote objects called Remote Method Invocation (RMI) [6].

This framework comes in several components. First, an RMI registry is required to be able to map names to instances of objects. The registry is used both locally, from within the same process or from two or more processes running on the same machine, and remotely over the network. The registry is the first point in which any RMI application gains access to remote objects. Next, an interface is written by the application developer to provide the method calls available to the remote clients. The implementation of this interface resides on the server and this object must be explicitly registered with the RMI registry by the application. It is possible to export objects from an RMI server without using the registry by passing it as a return value from an object that is. This allows RMI servers to act as a factory to clients, each client receiving a unique instance
of an object that resides on the server and using RMI to call this object on the server [6][11].

It is not possible for the client to pass references to its own objects to the server. Each parameter to a method call must be serializable, meaning that it implements the Serializable interface in the Java library. Serializable objects are those that may be represented as a sequence of bytes, which may be saved to a file or transmitted over a stream or network connection. Objects that are not serializable will usually contain some form of state information dependent on the operating system and include file handles, handles to open network connections, internal state information, or any other form of data that may become invalid sometime after being stored to a file. When an RMI client sends one of its objects to the server as a parameter, it is copied using serialization and the byte sequence that results is sent over the RMI protocol as part of the call. Changes made to these objects are not sent back to the client, only the return value is sent over the network connection. Objects that are references to the server's objects are passed by reference, and any changes are visible to the client and server. Objects that are neither serializable nor references to server objects cannot be passed over an RMI connection [6][11].

In previous versions of Java, a special utility called rmic (RMI compiler) was required as an additional step in the build process for any application using RMI features, both servers and clients. This utility generated stub classes from the remote interfaces in the application, where the network and framework code would be executed in order to perform the actual RMI call. This procedure is no longer necessary as of version 5 as the Java Virtual Machine will automatically generate these stub classes at runtime. The stub
classes still exist, but now in the form of proxy classes and are managed by the JVM internally instead of the application [6][11].

    The inclusion of RMI is a reliable way to add remote procedure calls to an application. This framework requires minimal amounts of additional programming from the application developer and removes the need to write socket code. RMI is handled by the JVM and is fully compatible across multiple architectures that support Java. However, being a minimal approach to RPC, RMI has many shortcomings that are addressed by other technologies in the Java platform [6][11].

2.9 EJB

    In Java EE applications, the middle tier in a 3-tier application is known as the business layer, and contains most of the backend logic in the application. This part of the application is implemented by a set of classes following the Enterprise JavaBeans (EJB) specification and runs in the EJB container of an application server. Before version 5 of Java, writing EJB classes required the use of XML deployment descriptors and calls to the framework to access other EJB classes. Version 5 of Java introduced the annotation language feature, which allows elements of the Java source to be decorated by special empty classes known as annotations. The EJB annotations are then accessed at runtime to determine the behavior of the EJB classes. This version of the EJB specification included other modifications that brought EJB closer to resembling business objects developed in the Spring framework, ultimately reducing the amount of complexity in developing EJB applications [2][11].
Like RMI, EJB classes are accessed by the client using an interface available to the client. Application servers use RMI or a similar protocol such as IIOP to facilitate the communication between remote and local objects. However, EJB requires multiple interfaces for an EJB class. One interface is specified as the local interface for the EJB, and is only available to objects running within the same instance of Java. The other type of interface is the remote interface, and is available to remote instances of Java, either on the same machine or across a network connection. This may lead to a greater amount of access, such as the ability to call secure methods, to the EJB class from within the server. These types of beans are known as session beans in EJB 3.0 and newer. A session bean may be either stateless, stateful, or singleton, which is specified using an annotation attached to the bean class definition. The stateless, stateful, and singleton semantics are similar to the request, session, and application scopes of a web application. A stateless bean is reinitialized each time it is requested from the server and does not implicitly persist any information after the bean itself loses scope. Such beans are often used to access services that can function entirely using the method parameters given to it from the client. A stateful bean is associated with the client session in a way that data stored within the bean can be re-accessed when the same client calls it. This can include cached data, configuration, or any other information that may need to be retrieved from the client at a later time. This information may be permanently stored depending on the server's configuration, and the server is responsible for determining the lifetime and storage method of the state information. The singleton bean is similar to stateful beans except that the data stored within a singleton bean is global to all clients. Another type of EJB exists
in practice under the EJB 3.0 framework and is known as the message bean. This bean
acts as a front end to the Java Messaging Service (JMS), which is a framework used to
deliver asynchronous messages in a publish/subscribe pattern throughout an EJB
container. Such beans may be used to submit jobs in a workflow environment or to
process form submissions from a web page. In contrast to these types of EJB classes,
using RMI to access remote objects provides either a singleton object, when an object is
looked up from the RMI server by name, or a stateful object when an object is returned
by a singleton object in the RMI server. Other EJB features are not automatically
available using the RMI framework [2][11].

Before EJB 3.0, dependency injection was performed manually in the XML
deployment descriptor and by calling the JNDI component of the application server. 3.0
introduced a form of dependency injection similar to the Spring IoC container. While
Spring developers may choose from several methods of dependency injection in their
application, an EJB 3.0 application uses the @EJB annotation to declare dependent

The following 3 code segments are an example of a stateful session bean
resembling a login module of a networked application, starting with the remote and local
interfaces and then the class implementation:

```java
@Remote
public interface StatefulRemote {
    void login(String userName, String remoteHost);
    String getLoginName();
    String getRemoteHost();
}

@Local
public interface StatefulLocal {
    String getLoginName();
```
String getRemoteHost();
}

@Stateful
public class StatefulBean implements StatefulRemote, StatefulLocal {

    private String userName;
    private String remoteHost;

    public void ejbCreate() throws CreateException {
        userName = "nobody";
        remoteHost = "localhost";
    }

    public void login(String userName, String remoteHost) {
        this.userName = userName;
        this.remoteHost = remoteHost;
    }

    public String getLoginName() {
        return userName;
    }

    public String getRemoteHost() {
        return remoteHost;
    }
}

Note the ejbCreate() method in the implementation class. This method is called by the EJB container automatically when the EJB is instantiated. The EJB is then accessed by either a local client, such as a web application running from the same application package, or a remote client, which can either be a separate web server or GUI application [11]. The bean is accessed by the client using the JNDI implementation provied by the EE library:

    private StatefulRemote lookupStatefulBean() {
        try {
            Context c = new InitialContext();
            return (StatefulRemote)
                    c.lookup("java:comp/env/StatefulBean");
        } catch (NamingException ne) {
            java.util.logging.Logger.getLogger(getClass().getName()).log(java.util.logging.Level.SEVERE, "exception caught", ne);
            throw new RuntimeException(ne);
        }
    }
}
3 BACKGROUND – Spring

3.1 Introduction

The Spring Framework was released in 2003 as an application framework running on top of the Java EE platform, though it can run on a Java SE distribution as well. Currently, its most recent production version is 3.0.2. It requires Java SE 5 in order to support the annotation language feature and the newer proxy-based RMI system, both introduced in version 5. The developers of the framework operate a community website, www.springsource.org, where Spring programmers can access the framework itself, view documentation, learn about upcoming events and training sessions, and interact with other developers. In 2009, VMware, Inc. acquired SpringSource. The framework itself is released under the Apache License, but several commercial products are now available from SpringSource and VMware such as a modified Apache Tomcat web container, and the SpringSource Tool Suite, a modified version of the open-source Eclipse IDE with features related to Spring Framework development [24][25].

3.2 Inversion of Control

The first, and most important of the core technologies provided by the framework is the inversion of control container, commonly referred to as the IoC Container. This
feature implements several design patterns for your application, including dependency injection, factory, and singleton. Without dependency injection, if an object has some kind of dependency on an external object, it is the responsibility of the first object to resolve the dependency itself before using the external object. This will result in additional code being written for each object that do not fulfill the responsibility of the object. Additionally, this code must be consistent throughout the application or it may result in incompatibilities later on. This leads to excessive boilerplate code [15]. Both Java EE and Spring attack this problem in similar ways. In Java EE, dependency injection is done inside the EJB layer of the application. Before EJB 3.0, dependencies were specified in XML files called deployment descriptors. These dependencies were resolved by the container as part of the deployment process, before any application code is allowed to run [11]. However, EJB programming introduces additional complexity into the program. For example, since remote objects were an essential feature in EJB, the container required three interfaces to be written for a single EJB class: remote, local, and home. The EJB model also uses several reserved words, such as an initialization method ejbCreate(). The Spring Framework takes a different approach to dependency injection than EJB 2.x and below. Instead of using framework-provided interfaces and specifications, each Spring “bean” may be written as a POJO (Plain-Old Java Object). That is, a class without any references to the framework itself [15]. Also, since Spring beans do not implicitly support remoting, it was possible to use as little as one interface per bean for beans that need to be accessible by other beans [22]. This made Spring an
easier alternative to EJB. Eventually, EJB 3.0 was released by the Java community and closely resembled the Spring POJO bean paradigm [15].

Take the following example MVC application into consideration: a database-driven, web-based, restaurant management toolkit. This program keeps track of several aspects of restaurant management: table status, guest list, menu items, hosts and hostesses, and reservations. These components are managed by three main objects, FloorDb, MenuDb, and ReservationDb. Each of these objects provides methods regarding database manipulation of the five individual components of the restaurant, application logic to manage the components and their relationships to each other, and a layer between the model and view parts of the application. The following XML configuration creates a dependency relationship between these three main objects and the rest of the application:

```xml

  <bean id="DataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://db.m202.net/restaurant"/>
    <property name="username" value="user"/>
    <property name="password" value="password"/>
  </bean>

</beans>
```

2 Required header [15]
<bean id="FloorDb" class="net.m202.restaurant.db.FloorDb">  
    <property name="DataSource" ref="DataSource"/>  
</bean>

<bean id="MenuDb" class="net.m202.restaurant.db.MenuDb">  
    <property name="DataSource" ref="DataSource"/>  
</bean>

<bean id="ReservationDb" class="net.m202.restaurant.db.ReservationDb">  
    <property name="DataSource" ref="DataSource"/>  
    <property name="FloorInterface" ref="FloorDb"/>  
</bean>

<context:component-scan base-package="net.m202.restaurant.web"/>

Although this may seem lengthy and verbose, it clearly describes the relationships between the objects in the application. All objects refer to a bean called DataSource, which is instantiated as library class which implements basic database connectivity to a JDBC-capable database class. In this case, the restaurant application is using a MySQL database to store information, but this can be easily changed to any of the other popular database packages that have JDBC drivers. The XML file also describes an additional dependency between ReservationDb and FloorDb.

The JavaBeans convention defines three main requirements that all bean classes should follow. First, the bean should have a zero-argument public constructor. Second, the bean may have zero or more properties assigned to it, and using specific naming conventions, these properties may be access by other objects. Third, the bean should be serializable, or specifically, implement the Serializable interface in the Java language. Spring deals specifically with the second requirement for beans, properties. In JavaBeans, a property is defined as a private member object or primitive in the bean that is accessed using “setters” and “getters.” The setters/getters are written in a specific way to match the
name of the property. For example, if a property of a class is named firstName and of type String, it may be read by other classes using a method with the following signature:

```java
String getFirstName(). It may be modified using a method: void
setFirstName(String x). Note that the case of the first character in the property name is inverted in order to separate the set and get keyword from the name of the property. This standard applies to all properties with one exception – a boolean property's getter method is prefixed with “is” instead of “get” [1][6][11].
```

The Spring Framework uses the JavaBeans properties component to configure Spring beans and their dependencies. This is seen in the above XML file as the different property elements. A property may be either a string value, such as the url property for the DataSource class, or a reference to another Spring bean. Each property in the XML file must match with a JavaBeans property in the target class [15]. In the ReservationDb class of our restaurant toolkit, the two dependency properties are sent to the following method calls:

```java
public void setDataSource(DataSource dataSource) {
    this.ds = new JdbcTemplate(dataSource);
}

public void setFloorInterface(FloorInterface floor) {
    this.floor = floor;
}
```

In EJB 2.x, it would be necessary for a class to call the framework in order to look up another EJB class [11]. Spring provides the advantage of allowing the programmer to write code that is not dependent on the framework classes, but still access Spring's IoC functionality [15].
3.3 Resource Management

In Java SE and EE, resources are files packaged with the application. Resource files may contain text for message translations, online help, configuration, or other formats such as graphics and sound used in a GUI application. In Java, all compiled code and resource files are stored within a directory structure, and this directory structure is often compressed and packaged into a ZIP archive. These archives are assigned different file extensions depending on their purpose. A regular Java application is assigned the JAR extension, web applications are assigned the WAR extension, and enterprise applications (client, web, and EJB combined) are assigned the EAR extension. There is also a required META-INF directory inside each archive and a WEB-INF directory for web archives [6] [11].

Although Java provides an interface to access resources, it is limited to the directory structure used to access compiled code. Spring extends this functionality to include other types of resources, such as files accessed using HTTP and filesystem objects [16]. For example:

```java
Resource template =
ctx.getResource("http://myhost.com/resource/path/myTemplate.txt");
```

Resources can also be referenced in the IoC container using the XML configuration. This is accomplished by setting a property of type Resource to the string location of the resource in the same format as the getResource() call [16].
3.4 JDBC

Both Standard and Enterprise editions of Java include an API called Java Database Connectivity (JDBC), which provide a link from Java to SQL databases. Unlike the similarly-named Open Database Connectivity (ODBC), it does not directly share database configurations with other applications. However, it does provide a uniform API for all JDBC-enabled databases. It is used as a component of a Java application in the same way as any other library, such as Logging or XML. SQL connections using JDBC are supported for multiple vendors through connectors. Many popular SQL servers, including Microsoft, MySQL, Oracle, and Postgres have JDBC connectors written for them by the vendors themselves. This allows a single application, given that it is written in a way that follows standard SQL semantics, to support multiple SQL servers simultaneously. JDBC includes many features that are common to database client applications. Using an additional layer, EE servers add support for transactions to database-driven applications. This has been standardized in an API called Java Transaction API (JTA). JDBC is also supported by connection pooling, a technique used to optimize the operating system handles used to connect the application to a SQL server. In EE application development, it is preferable to use a connection pool rather than to access JDBC connections directly [6][11][16].

A JDBC driver can be connected to an application in a number of different ways, depending on the environment that the application is running under (SE, EE, Spring, etc.). The simplest and least flexible way to instantiate JDBC classes is to refer to the driver classes directly by name using the Java built-in method Class.forName("..."), which
performs a runtime binding to the driver class. Although it is possible to make this call use external instead of hard-coded configuration, it is preferred to use an application server's JDBC binding whenever such capability is available. Any EE server, including Apache Tomcat and Sun's own EE server, GlassFish, provide their own ways to configure JDBC objects at runtime. The EE specification provides a way for applications to access these objects, but each application server may implement the configuration of these objects in their own way [11]. For example, Apache Tomcat uses XML descriptors outside of the application package to declare server resources, including JDBC connections [1]. GlassFish provides tools, both command-line and GUI, to declare and configure JDBC connections as well as other resource objects. Since JDBC drivers are required to be implemented as JavaBeans, the application server may provide additional configuration parameters to the JDBC driver through set*() method calls. For example, most drivers will implement the setUrl(String), setUserName(String), and setPassword(String) methods to provide basic connectivity to the application. Each JDBC driver contains its own set of properties, many of which are specific to the implementation of SQL being used, and this variation of property names is what makes it preferable to use the application server's configuration interface instead of coding direct calls to the JDBC driver inside the application [1][11].

Spring uses a similar technique to declare and configure JDBC objects. As the JDBC and pooling objects follow the JavaBeans standard, it is possible to inject them into a Spring application using the XML descriptor of the application's IoC container [15]
The following XML bean definition describes how a JDBC connection can be made available to the rest of the application. This bean uses connection pooling:

```xml
<bean id="DataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/restaurant"/>
    <property name="username" value="dbuser"/>
    <property name="password" value="dbpassword"/>
</bean>
```

This bean configures parameters that are passed to the BasicDataSource object, which are in turn passed to the MySQL JDBC driver (com.mysql.jdbc.Driver). The Url, Username, and Password properties are all supported by the connector and will correspond to appropriate connection parameters in this database server. Note that since this is a pooled connection, the BasicDataSource object will determine when to make a new connection to the database server and not the application.

The DataSource bean is now available to the rest of the Spring application as a singleton bean. The IoC container will automatically instantiate and reference this bean at runtime when beans referring to the DataSource are instantiated. Using the same example application, the following XML descriptor element shows how the DataSource bean is injected into another application bean:

```xml
<bean id="MenuDb" class="net.m202.restaurant.db.MenuDb">
    <property name="DataSource" ref="DataSource"/>
</bean>
```

Finally, now that the DataSource object has been injected into the application bean MenuDb, the application may now begin to use it to access the database. The BasicDataSource implements an interface named DataSource (should not be confused with our bean name “DataSource”) which provides a method to access connections from
the pool. However, instead of accessing the connection directly, Spring provides an additional wrapper class to interface with the connection object. The class used in this example is JdbcTemplate. The template class provides 4 major operations, call() to execute SQL stored procedures, execute() to execute a generic SQL statement most often a DDL statement such as “CREATE TABLE ...”, query() to execute a SQL statement that will return data such as “SELECT ...”, and update() to execute a SQL statement that will return a number of rows affected such as “UPDATE ...” and “DELETE FROM ...”. Using these 4 methods, an application developer will have the majority of SQL features needed to write a database-driven application. The following code is taken from the MenuDb bean in the example application, and demonstrates the use of JdbcTemplate to perform a simple update:

```java
private static final String ITEMS_UPDATE =
    "UPDATE items SET menu=?,name=?,description=?,price=? WHERE id=?";

public void setDataSource(DataSource dataSource) {
    this.ds = new JdbcTemplate(dataSource);
}

public void updateItem(Item item) {
    ds.update(ITEMS_UPDATE, item.getMenu(), item.getName(),
              item.getDescription(), item.getPrice(), item.getId());
}
```

Spring provides an interface for retrieving data from rows in a SQL SELECT statement. This is done through the RowMapper interface. Using this interface, it is possible to retrieve a set of data from a SQL table without performing any iteration in the application itself, Spring will iterate through SQL rows on its own and allows the application developer to write code that maps row data into the application's data objects [11][17].
The following code in the MenuDb bean selects all items from the menu table and maps them into a list of menu items:

```java
private static final String ITEMS_SELECT_ALL = 
    "SELECT i.id,i.menu,i.name,i.description,i.price FROM items i";
...
private RowMapper<Item> menuRowMapper = new RowMapper<Item>() {
    public Item mapRow(ResultSet rs, int rowNum) throws SQLException {
        Item i = new Item();
        i.setId(rs.getInt(I_ID));
        i.setMenu(rs.getString(I_MENU));
        i.setName(rs.getString(I_NAME));
        i.setDescription(rs.getString(I_DESCRIPTION));
        i.setPrice(rs.getFloat(I_PRICE));
        return i;
    }
};
...
public List<Item> getAllMenuItems() {
    return ds.query(ITEMS_SELECT_ALL, menuRowMapper);
}
```

In this case, the RowMapper being used is an anonymous inner class. The mapRow() method is called by Spring for every row returned in a JDBC result set. Compared to traditional JDBC coding practices, using Spring to manage data objects significantly reduces the amount of boilerplate code required to develop a database-driven application while allowing for the flexibility and loose coupling expected for EE application design.

### 3.5 Aspect-oriented Programming

The Spring Framework includes support for Aspect-Oriented Programming (AOP) as an optional feature of the IoC container. While AOP is a broad topic that extends beyond the scope of the Spring Framework, it integrates well with the IoC container and the type of programming strategies employed by Spring developers. AOP
is an extension to the Object-Oriented Programming principles, and thus requires OOP to function. Whereas OOP is concerned with classes and objects, the basic unit of AOP is the aspect. An aspect is a concern that cuts across classes in an application, or a "cross-cutting concern." Examples of such concerns include transaction safety, security, caching, logging, and filtering [4][13][19].

In AOP, aspects are implemented as separate classes that can modify the behavior of other classes without making any modifications to the classes themselves or the context in which they are called. The point in which an object's method is called is known in AOP as a join point. One component of AOP takes effect in the join points of an application. A developer may specify an action that is to be taken either before, after, or around the execution of a join point for a set of classes. This is known as advice, and in Spring it is attached to method calls in special classes other than the classes in which advice is given. AOP is powerful in the way that advice can be given to entire groups of classes without specifying each individual class by name. Advice can be assigned in one of five ways in Spring. The simplest forms are "before" and "after," which are actions executed before or after a join point is executed. In the case of "before," the class receiving advice will always be executed. In the case of "after," the advice will always be executed. Two other forms of advice are "after returning" and "after exception," which will be executed depending on whether the class receiving advice returns normally or throws an exception. The most powerful form of advice is "around." This form is executed in place of the actual class, but may execute the join point from within the
advice. This provides the same amount of functionality as the other four types of advice, but adds the ability to choose whether or not to execute the actual join point [4][19].

AOP uses a special syntax to configure the interaction of advice and advised classes in the form of pointcut expressions. Once an advice method is declared, a pointcut expression is assigned to it in order for Spring to determine when this method is to be executed. Spring borrows from another AOP framework called AspectJ to utilize pointcut expressions. The components of a pointcut expression may include class name, method name, and arguments. The pointcut expression also supports the use of wildcards such as *, which in turn allow advice to be assigned to multiple classes and methods using the same pointcut expression. An example of such an expression may be

"execute(net.m202.transaction.* doAction(..))" which would affect any class in the transaction package and any call to the doAction method of this set of classes. It is also possible to use Java 5.0 annotations in a pointcut expression, which would affect classes possessing the annotation. For example, a class that performs a network operation can have the @Secure annotation attached to it, and AOP would then be able to modify its behavior and add security features to the class [4][19].

Another major feature of AOP is introduction. This allows an interface to be added to a class without modification of the class itself. Additional interface imply the addition of new methods to the class, and these methods are implemented by the aspect and not the affected class. This can also be used to access any persistent information stored within the aspect that is modified at join points [4][19].
One critical thing to note about AOP is that the rules of OOP, such as encapsulation and inheritance, are preserved in AOP applications. AOP does not make any actual modifications to the objects receiving advice, it instead takes effect either at the join points or during the instantiation of an object. A major difference between Spring AOP and other AOP frameworks is that Spring does not require separate compilation and it does not control the class loader. Spring uses proxy classes, similarly to how RMI is able to dynamically generate the stub objects for a remote object, to wrap advised classes, and takes advantage of the IoC container to provide these proxies. The programmer must use a proxy instead of the original class in order to take advantage of the AOP features. This slightly limits Spring's ability to serve as an AOP framework, which Spring's documentation states that it will never be a complete implementation of AOP, but provides most of the common needs of AOP developers [4][19].

3.6 Remote Objects

The Spring Framework meets the requirements for implementing an n-tier application with its support of multiple remote procedure call schemes. These schemes are built on existing Java technology and can easily replace the behavior of other Java EE remoting methodologies, like EJB, with a simpler POJO-based development strategy, in keeping with the overall Spring design. A purely Java solution, Remote Method Invocation (RMI), exists in Spring and uses the existing Java RMI framework as its underlying transport. This form of RMI can be used with the IoC container to add the support of remote objects as dependencies of other Spring beans [15][22]. In contrast,
using RMI or EJB requires programmatic queries to the RMI name server or JNDI context in order to acquire references to remote objects [11]. Using Spring, all dependencies and remote lookups are handled by the IoC container and their corresponding proxy objects are injected just as any other bean class. This differs from EJB in the way that the notion of local EJBs and local interfaces are not present in the Spring RMI service, however it is possible to implement such a feature using the non-RMI functions of the IoC container [11][15][22]. The following example IoC configuration demonstrates how to export a Spring bean with RMI:

```xml
<bean id="restaurantServiceBean" class="net.m202.restaurant.Service"/>

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="RestaurantService"/>
    <property name="service" ref="restaurantServiceBean"/>
    <property name="serviceInterface" value="net.m202.restaurant.IService"/>
    <property name="registryPort" value="1199"/>
</bean>

Then, the IoC configuration on the client is set up to access the RMI object:

```xml
<bean class="net.m202.restaurant.Client">
    <property name="restaurantService" ref="restaurantServiceClient"/>
</bean>

<bean id="restaurantServiceClient" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://server_addr:1199/RestaurantService"/>
    <property name="serviceInterface" value="net.m202.restaurant.IService"/>
</bean>

Note that it is possible to use SPEL (Spring Expression Language) in this configuration to perform variable substitution in any of these attributes, such as ${RMIServerAddress} [18]. Also, there are no special changes in the XML for this functionality to be enabled, it is simply using Spring framework classes as beans which in turn will manage the RMI
configuration of the application beans. This example essentially performs the same steps as would be to instantiate a Service object (implementing the IService interface) and export it using the RMI registry's bind() method on the server [15][22]. On the client, this example does the same as locating the remote object using the registry's lookup() method, then injects the reference into the Client bean. The result is an injected dependency that acts as if it was within the same instance of the application [22].

While RMI is the simplest protocol for accessing remote objects in Java, the Spring Framework supports other RPC architectures in order to better inter-operate with other languages and platforms. Some of these architectures are based on the HTTP protocol and XML. In Java EE, these are known as Web Services and are supported by GlassFish and other application servers. Depending on your version of Java EE, this can be either JAX-RPC (Java API for XML) or JAX-WS. By exporting Spring beans in this way, any client capable of making HTTP requests and interpreting XML can access these objects. In the IoC container, these remote beans are configured similarly to RMI-export beans, but use different framework classes to configure the remoting capability [11].

As the Spring RPC architectures are similar in design to Java EE, the same advantages and weaknesses exist in the Spring application. RMI is a well-known and stable protocol in Java, but suffers from its own inefficiency by serializing data over the network connection and its lack of security and session features. JAX exposes all features of the bean to anyone who can access it, which can be useful for public services but would not be acceptable in more controlled circumstances. Ultimately, the burden is on
the programmer to determine which remoting features are necessary for an application and the trade-offs in selecting an architecture [11][22].

3.7 Spring MVC

The Spring Framework includes a MVC development platform based on Java EE web design. In Java EE, there are a few ways to implement this pattern. Java EE uses entity beans in EJB 2.x and persistence in EJB 3.x as the model. The model is often backed by a database engine using JDBC for its low-level storage. The controller is implemented as a Servlet object. Servlets are classes all implement a standard interface which allows them to respond to HTTP requests, similar to traditional CGI applications. However, unlike CGI applications, the servlet's lifetime can extend far beyond the HTTP session itself. The servlet object and its related classes have the ability to access many features of the HTTP session including request and response headers, dispatchers to other HTTP connections, cookies, session variables, and form submissions. It is possible to write an entire website using servlets, however this is not an efficient development methodology. Java EE usually considers the servlet to act as the controller in our MVC pattern. Since EE applications are often web-based, the view of our MVC application is written in a form that is friendly to web development architecture. Java EE provides the JavaServer Pages (JSP) language extension to Java. JSP is similar to other web-based languages, such as ASP and PHP, that it allows the programmer the ability to mix markup and executable code within a file. Java EE web containers, such as Apache Tomcat, generate intermediate Java source files that implement the servlet interface and wraps all
of the markup into method calls to produce output. This results in fast precompiled code in contrast to interpreters such as PHP. This also allows the powerful features of the servlet to be accessible from JSP code, but in a way that is more friendly toward developers [11][20][21].

Spring Web MVC is similar to Java EE in this regard. When using Spring in a web container such as Apache Tomcat, Spring itself is instantiated as a servlet [1][15]. However, the Spring servlet is responsible for pre-loading beans in the IoC container and dispatching HTTP requests to specific controller objects. The controller objects can either be specified in XML, or more preferably, scanned automatically and indicated by the @Controller annotation in Java 5.0 and newer [15]. Although Spring controller objects are not as powerful as servlets with regard to accessing the HTTP session, they follow the Spring programming methodology of using POJOs to encapsulate application logic and dependency injection to resolve external components at runtime. Spring controllers have several different ways to choose from to pass information to the view, from returning a string of text which is written to the HTTP response to highly optimized parameter maps using the ModelAndView class provided by Spring [20][21]. In the interest of simplicity, the restaurant toolkit uses HTTP response strings to pass data to the view. The following example demonstrates a controller for the restaurant menu, in which the contents of the menu are sent to the view in XML format:

```java
@RequestMapping(value = "/items.get")
@ResponseBody
public String getItems(String id, String search) {
    StringBuilder sb = new StringBuilder();
    int idNumber = 0;
    idNumber = Integer.parseInt(id);
    ... 
    sb.append("<items>");
```
List<Item> items = menudb.getAllMenuItems();
for (Item i : items) {
    sb.append(i.toXml() + "\n");
}
...
sb.append("</items>\n");
return sb.toString();
}

The main difference between this controller and a servlet-based controller reflects the differences between writing Spring applications and Java EE applications. In this example, the controller is written inside a method call that is part of a POJO [15][20][21]. No framework calls are necessary except for the annotations used to: 1. Identify that this class is a controller by annotating the class with the @Controller annotation, 2. Map this specific method call to an HTTP request matching the path “items.get” and 3. Indicate that this method call will return a string that is sent over the HTTP connection. The parameters “id” and “search” are mapped automatically by Spring to take the values of parameters, also named “id” and “search”, from the HTTP request itself. Therefore, if a user requests the URL http://myserver/servlet_path/controller/items.get?id=100, the 100 is automatically converted to the appropriate type and passed into the method call in whatever parameter name matches [20][21].
4 PROBLEM DEFINITION

4.1 Observed Challenges

While developing the sample web MVC application, the restaurant simulator described in the previous chapter, a few challenges arose from using the Spring platform and development methodology. The application followed many of the example code shown in the Spring documentation, which suggests that this application was developed in a way consistent with the intent of the Spring developers. As with any application, there were deviations from what could be considered standard practice. For example, the restaurant simulator uses a partially XML-based web interface instead of HTML. This was done to demonstrate easy customization of web interfaces. Since XML represents data only, instead of data and presentation seen in HTML, the presentation elements of the web interface are stored in a separate file. In order to translate XML into HTML, the XML output from the middle layer is filtered through a separate class which performs the transformation process (XSLT). This output is then read as text by the view code and embedded into the web page. Using XML was the single most significant difference from the development strategy shown in the Spring documentation.
The simplest model class in this application was responsible for managing the menu items. Unlike other controllers, the menu model operates independently and manages a single database table. This class contains 97 lines of source code, 15 of which were string definitions for identifiers and SQL statements. The class itself provides a simple CRUD (create, read, update, delete) interface to the menu database table as well as 2 search methods and 1 select-all method. Each method contained between 1 and 10 lines of code.

This application uses a single controller class called RestaurantController. This controller manages all models (menu items, tables, orders, parties, and reservations) and is 801 lines of source code. However, unlike the menu model class, a significantly higher percentage of lines correspond to executable code because the SQL strings are not a part of this class. There are 105 lines for the menu controller methods, though similar controllers are approximately the same line count. This adds up to 187 lines of executable source code for just the menu table, not including the view code.

4.2 Rapid Application Development

As this application was written completely from scratch using NetBeans IDE for context assistance only, it demonstrates the usefulness of code generators and RAD (Rapid Application Development). RAD can be used to describe many types of applications, from IDEs themselves to specialty code generators used for specific domain projects [3][5][14]. The type of RAD used in this context refers to rapid prototyping, code generation, and database-driven applications that utilize Spring's Web MVC.
framework. In a more generic sense, other RADs generate database-driven and MVC applications for other languages and frameworks [2]. Although the restaurant application can be written faster and using less lines of code by an experienced Spring developer, this type of application can also benefit greatly from using an RAD tool. An RAD-generated application will likely contain large amounts of code for simple database operations, such as the menu model code, but this is usually unavoidable in Java since it is not specifically optimized for database programming. Certain other languages are better optimized for database-driven applications, they are usually called 4th Generation Languages. However, due to the powerful features offered by Java and its libraries, many developers may consider this a trade-off from using 4GLs. The same argument can be made for other languages such as PHP and ASP, where RAD tools also exist [2][5].

4.3 Spring Roo

SpringSource, the group behind the Spring platform, has already published an RAD tool called Roo (Fig. 1). This program is available from their website, springsource.org, under the Projects menu. Roo does not come pre-packaged with the Spring platform download, it is only available as a separate program. Roo is freely available and licensed under GPLv3, however use of Roo does not force any licensing obligations on the target source code. This is similar to the exception in GNU Bison, a C code generator. Roo promotes itself as “easy to use” and “standards-based” among other features. The code generated from using Roo does not require any additional platform libraries; a program using Spring without Roo will require the same dependencies. Roo
can remove itself from a project once its features are no longer required by the developer, or if the developer should decide to abandon using Roo for the project. Roo will only generate Java and JSP code, but can also manage build instructions. Roo can generate complete web MVC applications using default templates. These templates will produce a web interface that uses some of the same design elements from springsource.org, and any customizations will need to be done by the programmer in the generated web code. The documentation specifically mentions modifying the Cascading Stylesheet (CSS) portion of the web code to change the design of the generated web pages [23][24].

Fig. 1: Roo startup screen and example of 'hint' command
Roo is designed to work with the Spring Tool Suite, a customized version of Eclipse IDE enhanced with features to specifically interpret Spring descriptor files [23][24]. Roo does not require use of Eclipse or any other tool and does not place any restrictions on the programmer's choice of tools, the developers of Roo decided to make usage of the tool flexible enough that it fits into the programmer's methodology rather than imposing its own. The developers of Roo also chose to implement it as a text interface rather than graphical. This was done in order to allow Roo commands to be executed in any order and to make the interface more intuitive. The text interface functions similarly to any other program based on GNU readline and includes command history with arrow navigation, tab completion, and internal help. However, unlike many other similar applications, Roo contains a 'hint' command that evaluates the current state of the project and provides assistance to the programmer. The Roo developers make frequent reference to this command as a major contribution to its usability [24].

Iron Speed Designer is another RAD tool that contrasts Roo in several ways. First, ISD is a commercial product that boasts a large footprint in the IT industry. Their website, ironspeed.com, provides a trial version of the software, or offers tiered per-seat licenses from around $1000 and above for each developer in an organization. It does not impose charges for applications developed using ISD, only for the tool itself. The applications generated by ISD are in either VB.NET or C#, and the web interfaces utilize AJAX and Web 2.0 features for high amounts of interactivity. ISD's interface (Figs. 2-3) is graphical rather than text and offers previews of the generated interface. The interface also contains some IDE features such as UI design and a source code editor. ISD provides
a wizard interface for generating new projects, rather than the un-ordered design of Roo [10].

Fig. 2: Iron Speed Designer application wizard [10]
Fig. 3: Iron Speed Designer preview [10]
The following examples (Figs. 4-8) demonstrate the usage of Roo in creating Java objects for the menu database.

Fig. 4: Starting a Roo project
Fig. 5: Using the 'hint' command during project
Fig. 6: Generating stub database parameters
Fig. 7: Defining fields in an entity
This process creates an Item class similar to the handwritten Item from the restaurant example. Also, the database code required to manage the menu database is automatically generated by Roo. This code differs significantly from the handwritten version as it uses the Hibernate persistence library to manage database entries instead of direct queries to JDBC. Hibernate and other persistence libraries are a type of Object-
Relational Mapping (ORM) which assist in storing objects in a relational database such as those using SQL. The code generated by Roo also makes use of @AspectJ-style AOP, which further ties the code to Eclipse IDE. There is also some required manual modifications of the generated files, such as database parameters [23][24].

The developers of Roo have a clear justification for designing it with a text-based interface. Roo is effective at quick generation of code for web-based and other MVC applications [24]. However, other RADs can be seen as more intuitive and better suited for “rapid” development especially by inexperienced (with Roo) users and non-programmers, leading to quicker adoption in organizations. According to the IronSpeed website, many developers have been able to save thousands of man-hours and a wide range of costs in their projects using the ISD software. Since Roo lacks this major component, the potential for Spring as a viable platform for RAD-generated application is diminished. The best solution to this problem is one that encompasses the most commonly used features of Roo, if not all features, and an graphical interface in the style of ISD and other RADs.
5 PROPOSED SOLUTION

5.1 Overall Design

The primary requirements of this RAD solution are a GUI interface and the use of the Spring framework by generated applications. This tool, named JScaffold, serves as a GUI interface between the developer and the Roo tool, also known as a frontend. As Roo is designed with a modular architecture, introducing this GUI application as a frontend gives the user the same flexibility as Roo itself, including lock-in avoidance and third-party plugin development. This design also builds upon an existing and stable platform for RAD processes in the Spring framework and can be easily modified to provide new features added to Roo.

Since Roo is a console-based application and supports scripted text input, JScaffold communicates with Roo using a script file. The process of creating projects with the GUI application is slightly different from using the Roo command line. One example is an entity bean with multiple fields. In this case, several Roo commands are executed, depending on the number of fields. A GUI implementation of this process would consist of one form with multi-item elements, then map the interface state into multiple Roo commands when instructed by the user.
The initial plan for designing the GUI was to reconstruct an interface similar to the application wizard interface of IronSpeed Designer. The reasons for this decision are the simple wizard-like user interface, independent categories, and the similarities in features between ISD- and Roo-generated applications. The available ISD screenshots, such as the one used in the previous chapter, were used as a reference to build the new interface.

### 5.2 Development Process

The first step in developing this new application was to build a catalog of all relevant features available from Roo. This was done by examining the online reference manual and Roo's own internal help system. The help system lists all possible Roo commands and parameters, as well as a brief explanation of a given command's behavior. This part of the development process contributed much of the information available in the Software Model chapter.

The interface for ISD's application wizard is categorical and linear. The process in developing an application in ISD closely resembles the 5-minute instructional video of Roo. Each of the concerns of the application, such as database connections, entities, controllers, and forms are addressed individually during the design process. ISD's interface suggests that it is possible to return to previous forms and perform modifications to information previously entered by the user. JScaffolding performs complex operations as late as possible in order to allow the user to make such changes.
Since the GUI application is mostly a mapping of features from another application, the next step in the design process was GUI development for each of the features that were planned to be implemented. Documentation for this step consisted mostly of screenshots and tables linking GUI elements to Roo features. This information is available in the Prototype chapter.

5.3 Target Users

This application is targeted toward the same users as Roo. There are two main groups that this application will appeal to: users who are inexperienced with Spring but know Java, and users who are experienced with Spring and would like to develop applications faster than hand-coding. Because of the graphical nature of this application, it should appeal to inexperienced users more than Roo alone. However, an added benefit of a GUI frontend to Roo is the possibility of learning Roo through the behavior of the GUI. This can be accomplished by allowing the user to watch the Roo commands as they are being sent and displaying the Roo output. There are other GUI frontends to console applications that operate in a similar way.

All Spring Web MVC programmers should be familiar with Java, Java EE, and the MVC architectural pattern. A code generator, even an advanced RAD tool, can only generate an application with a limited set of features. Although the generated application may compile and run without error, it is usually far from complete. At some point, the programmer must add additional code to the generated code in order to add the required functionality to the application.
5.4 Features

This list contains the features available from the RAD tool and the applications that are generated:

- Model-View-Controller relationship between classes
- Entity classes using Java Persistence API (JPA)
- Field definitions for entity classes
- Database configuration
- JPA configuration
- Management of properties
- Web form generation
- Controller class stub generation
- E-mail configuration
- Build process management

This list summarizes the new features added to Spring RAD by JScaffold:

- GUI development
- Verification of form elements before committing to source code
- Modification and removal of elements
- Testing and validation of configuration items
- Visualization of interaction between GUI frontend and Roo
6 SOFTWARE MODEL

6.1 Introduction

This chapter describes the architecture and features of JScaffold. Roo is an interactive command-line shell application which operates mostly as a set of stateless commands that manipulate target files. These “target files” contain a portion of the source code for the generated application, the rest of the code is written outside of Roo. The goal of this design is to address as many Roo commands as are necessary to generate an MVC application. Roo's flexibility allows it to be used for other types of applications, and its plugin system can allow third-party developers add an indeterminate amount of additional functionality. But as this falls outside of the scope of MVC development, it was not supported by JScaffold [7][14].

6.2 Architecture of Roo-generated MVC applications

The architecture of Roo applications closely follows the modified model-view-controller pattern (Fig. 9) in Java EE. In this pattern, all data is passed through the controller and the model/view components have no direct interaction with each other. This is also the preferred MVC architecture described by the Spring documentation [7][14][24].
In this architecture, the model is represented by a set of classes known as entities. Many implementations of MVC in Java use objects known as DAOs (data access objects). Spring instead uses JPA (Java persistence API), an standard interface representing ORM (object-relational mapping), and AOP to store and retrieve data, thus eliminating the need for DAOs [23][24].

*Fig. 9: Modified MVC architectural pattern*
While simple classes can easily map into a single SQL table, ORMs that implement JPA such as Hibernate and OpenJPA support more complex classes and one-to-many relationships. In the case of a user-defined class existing as a field in another user-defined class, Roo and Spring generate an additional unique identifier field used to perform a mapping from one SQL table to another. Roo can also generate complex one-to-many relationships (Fig. 10). These types of relationships are defined as the set and reference entity types [23][24].

Fig. 10: Class diagram of one-to-many relationship

6.3 Architecture of J Scaffold

The initial plan for J Scaffold was an interface similar to the wizard-like design of ISD. This type of interface allows the user to have both a guided experience (Figs. 11-12) as well as the ability to review and correct information [12].
Fig. 11: Typical program flow
Fig. 12: Use cases
The internal representation (Fig. 13) of Roo commands is similar to the target application. A Spring MVC application contains a set of entities, controllers, and views (usually implemented in JSP), each of which is managed individually by Roo commands. Since the GUI application maps a visual representation of the target application into the appropriate Roo commands, the GUI application stores a structure of the application using classes matching the individual Roo commands.

Fig. 13: Internal representation of the target Roo application
6.4 Roo commands implemented

Since this application is a GUI front-end for Roo, most of the features match a corresponding Roo command. The figures below detail the planned Roo commands supported by JScaffold.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>controller scaffold</td>
<td>Create a controller that responds to the REST pattern</td>
<td>class: Java class name, entity: name of object exposed to web tier, path: REST base path, create, update, delete: enabled/disabled operations</td>
</tr>
<tr>
<td>controller class</td>
<td>Create an empty controller class</td>
<td>class: Java class name, preferredMapping: web controller mapping path</td>
</tr>
<tr>
<td>controller all</td>
<td>Automatically create controllers for all entity classes</td>
<td>package: Java package name for destination</td>
</tr>
<tr>
<td>database properties list</td>
<td>List all properties configured for the database layer</td>
<td>none</td>
</tr>
<tr>
<td>Command: database properties set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purpose: Set a property for the database layer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>key: property name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value: property value</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Command: database properties get |
| Purpose: Retrieve a database property |
| Parameters |
| key: property name |

| Command: email sender setup |
| Purpose: Configure outgoing e-mail settings |
| Parameters |
| hostServer: mail server name/IP |
| protocol: mail server protocol |
| port: TCP port number |
| encoding: message encoding |
| username: authenticated user name |
| password: user password |

<p>| Command: email template setup |
| Purpose: Configure outgoing e-mail headers |
| Parameters |
| from: origin ('from') e-mail address in outgoing messages |
| subject: subject text in outgoing messages |</p>
<table>
<thead>
<tr>
<th>Command:</th>
<th>entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>Create a new entity class</td>
</tr>
</tbody>
</table>
| Parameters | class: Java class name  
extends: superclass  
abstract: (boolean) enable to make entity class abstract  
testAutomatically: (boolean) enable to generate integration tests  
table: name of JPA table  
identifierField: name of JPA field to use as identifier  
identifierColumn: name of JPA column to use as identifier  
identifierType: name of Java class of identifier (def: Long) |

<table>
<thead>
<tr>
<th>Command:</th>
<th>field boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>Create a new boolean field in the current entity</td>
</tr>
</tbody>
</table>
| Parameters | fieldName: field name  
class: Java class name  
notNull: enable to prevent field from being assigned NULL  
nullRequired: enable to assert NULL value  
transient: (boolean) enable to make object transient  
assertFalse: (boolean) enable to assert false value  
assertTrue: (boolean) enable to assert true value  
column: JPA column name  
primitive: (boolean) enable to generate built-in primitive type |

<table>
<thead>
<tr>
<th>Command:</th>
<th>field date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
<td>Create a new date field in the current entity</td>
</tr>
</tbody>
</table>
| Parameters | fieldName: field name  
class: Java class name  
notNull: (boolean) enable to prevent field from being assigned NULL  
nullRequired: (boolean) enable to assert NULL value  
transient: (boolean) enable to make object transient  
future: assert future date  
past: assert past date  
column: JPA column name |
<table>
<thead>
<tr>
<th>Command</th>
<th>field email template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Create a new MailTemplate object field in the current entity</td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>fieldName</strong>: field name</td>
</tr>
<tr>
<td></td>
<td><strong>class</strong>: Java class name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>field enum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Create a new enum field in the current entity</td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>class</strong>: Java class name</td>
</tr>
<tr>
<td></td>
<td><strong>extends</strong>: superclass</td>
</tr>
<tr>
<td></td>
<td><strong>abstract</strong>: enable to make entity class abstract</td>
</tr>
<tr>
<td></td>
<td><strong>testAutomatically</strong>: enable to generate integration tests</td>
</tr>
<tr>
<td></td>
<td><strong>table</strong>: name of JPA table</td>
</tr>
<tr>
<td></td>
<td><strong>identifierField</strong>: name of JPA field to use as identifier</td>
</tr>
<tr>
<td></td>
<td><strong>identifierColumn</strong>: name of JPA column to use as identifier</td>
</tr>
<tr>
<td></td>
<td><strong>identifierType</strong>: (default: java.lang.Long) name of Java class of identifier</td>
</tr>
<tr>
<td></td>
<td><strong>type</strong>: Java enum type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>field jms template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Create a new JmsTemplate object field in the current entity</td>
</tr>
<tr>
<td>Parameters</td>
<td><strong>fieldName</strong>: field name</td>
</tr>
<tr>
<td></td>
<td><strong>class</strong>: Java class name</td>
</tr>
</tbody>
</table>
**Command:** field number

**Purpose:** Create a new numeric field in the current entity

**Parameters**
- **fieldName:** field name
- **class:** Java class name
- **notNull:** (boolean) enable to prevent field from being assigned NULL
- **nullRequired:** (boolean) enable to assert NULL value
- **transient:** (boolean) enable to make object transient
- **type:** Java type
- **decimalMin:** assert minimum value (BigDecimal string)
- **decimalMax:** assert maximum value (BigDecimal string)
- **min:** assert minimum value
- **max:** assert maximum value
- **column:** JPA column name
- **primitive:** (boolean) enable to generate built-in primitive type

**Command:** field other

**Purpose:** Create a new user-defined object field in the current entity

**Parameters**
- **fieldName:** field name
- **class:** Java class name
- **notNull:** (boolean) enable to prevent field from being assigned NULL
- **nullRequired:** (boolean) enable to assert NULL value
- **transient:** (boolean) enable to make object transient
- **type:** Java type

**Command:** field reference

**Purpose:** Create a new reference ('many' of a many-to-one relationship) field in the current entity

**Parameters**
- **fieldName:** field name
- **class:** Java class name
- **notNull:** (boolean) enable to prevent field from being assigned NULL
- **nullRequired:** (boolean) enable to assert NULL value
- **transient:** (boolean) enable to make object transient
- **type:** Java type of entity to reference
- **joinColumnName:** JPA column name to join
- **cardinality:** relationship cardinality (default: MANY_TO_ONE)
- **column:** JPA column name
<table>
<thead>
<tr>
<th>Command: field set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong> Create a new set ('one' of a many-to-one relationship) field in the current entity</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>fieldName: field name</td>
</tr>
<tr>
<td>class: Java class name</td>
</tr>
<tr>
<td>notNull: (boolean) enable to prevent field from being assigned NULL</td>
</tr>
<tr>
<td>nullRequired: (boolean) enable to assert NULL value</td>
</tr>
<tr>
<td>transient: (boolean) enable to make object transient</td>
</tr>
<tr>
<td>element: name of entity contained within the set</td>
</tr>
<tr>
<td>sizeMin: assert minimum size</td>
</tr>
<tr>
<td>sizeMax: assert maximum size</td>
</tr>
<tr>
<td>cardinality: relationship cardinality (default: MANY_TO_MANY)</td>
</tr>
<tr>
<td>fetch: JPA fetching semantics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: field string</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>fieldName: field name</td>
</tr>
<tr>
<td>class: Java class name</td>
</tr>
<tr>
<td>notNull: (boolean) enable to prevent field from being assigned NULL</td>
</tr>
<tr>
<td>nullRequired: (boolean) enable to assert NULL value</td>
</tr>
<tr>
<td>transient: (boolean) enable to make object transient</td>
</tr>
<tr>
<td>sizeMin: assert minimum size</td>
</tr>
<tr>
<td>sizeMax: assert maximum size</td>
</tr>
<tr>
<td>regexp: assert regular-expression pattern match</td>
</tr>
<tr>
<td>column: JPA column name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: jms listener class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong> Create a new consumer class for JMS message passing</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>class: Java class name</td>
</tr>
<tr>
<td>destinationName: JMS destination name</td>
</tr>
<tr>
<td>destinationType: JMS destination type (default: QUEUE)</td>
</tr>
<tr>
<td>Command: jms setup</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
<tr>
<td><strong>Purpose:</strong> Install a JMS provider to receive messages</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>destinationName: JMS destination name</td>
</tr>
<tr>
<td>destinationType: JMS destination type (default: QUEUE)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: project</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong> Create a new project (first command used in the Roo process)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>topLevelPackage: uppermost package name</td>
</tr>
<tr>
<td>projectName: project name (default: last segment of package name)</td>
</tr>
<tr>
<td>java: version of Java (example: 5)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: properties list</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong> Retrieve details of a properties file</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>name: properties file name</td>
</tr>
<tr>
<td>path: location of properties file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: properties remove</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong> Remove a property from a property file</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>name: properties file name</td>
</tr>
<tr>
<td>path: location of properties file</td>
</tr>
<tr>
<td>key: property name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command: properties set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose:</strong> Modify a property value</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
</tr>
<tr>
<td>name: properties file name</td>
</tr>
<tr>
<td>path: location of properties file</td>
</tr>
<tr>
<td>key: property name</td>
</tr>
<tr>
<td>value: property value</td>
</tr>
</tbody>
</table>
7 PROTOTYPE/RESULTS

7.1 Introduction

The process for which this application is named after, scaffolding, is the process in which code is generated to create database-driven applications. The original name was SpringBuilder, and was changed to avoid trademark issues. It is a desktop application that runs on the Java SE 6 platform. The application was written in NetBeans IDE 6.8 and includes a significant amount of generated code from the NetBeans GUI builder. It was developed and tested on the Ubuntu 10.04 Linux distribution.

7.2 Interface

The application GUI consists of one main window with six tabs and two additional dialog boxes. The design was inspired by the wizard-like interface of Iron Speed Designer. Each tab contains a major feature from Roo supported by JScaffold (Table 1). The main window contains a banner area used to display a brief description of the currently-selected tab's function and its relationship to Roo-generated applications. Most UI elements also contain tooltips, which are short descriptions of a control displayed when the user rests the mouse cursor over the control for a few seconds.
<table>
<thead>
<tr>
<th>Tab Name</th>
<th>Roo Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>project</td>
</tr>
<tr>
<td>Database Configuration</td>
<td>persistence setup</td>
</tr>
<tr>
<td>Entities/Fields</td>
<td>entity, field, enum</td>
</tr>
<tr>
<td>Controllers</td>
<td>controller scaffold</td>
</tr>
<tr>
<td>Properties</td>
<td>properties set</td>
</tr>
<tr>
<td>Messaging</td>
<td>jms listener class, jms setup</td>
</tr>
</tbody>
</table>

**Table 1: JScaffold to Roo mappings**

The Start tab (Fig. 14) contains the basic information about the target application. This information includes the name of the application, which is displayed in several places in the Web MVC forms, and the top-level Java package for the project. In order to ensure global uniqueness, a Java package name often corresponds to the DNS name of the project's originators followed by the name of the application and sub-packages within the project. Examples include org.apache.tomcat and com.sun.java, note that the DNS portion is in reverse. The package name field in this tab does not enforce any specific naming standard, but users should use discretion in choosing a package name. The directory field specifies the top-level directory name where project files are produced. This directory must not exist when during the code generation process, and JScaffold enforces this policy in order to prevent errors from occurring within Roo. The default value is a directory named after the project and within the platform-defined user home directory. After the user clicks the New Project button, other tabs will be enabled. If a project is already open, this button will clear the project model entirely.
The Database Configuration tab (Fig. 15) contains the next set of required parameters in the project model. Roo-generated applications use the Java Persistence API (JPA) to store objects in a relational database. Roo supports both Hibernate and OpenJPA as implementations for this layer, Hibernate is currently automatically selected by JScaffold. Roo also supports several database vendors. JScaffold is designed to easily allow definitions of these various database drivers such that the driver is also supported by Roo, currently only MySQL and Hypersonic are supported. The form contains server name or address, username, password, and database name as fields. These fields are mapped into parameters for Roo, and Roo maps these parameters in driver-specific properties. JScaffold will also scan the driver for any additional properties that the driver may support. These properties follow the JavaBeans standard, allowing JScaffold to easily identify and manipulate them.

Fig. 14: JScaffold welcome screen
For some drivers, such as the MySQL driver, JScaffold can attempt a connection with the specified parameters as a convenience to developers (Fig. 16). This is similar to the functionality of the ODBC control panel in Windows and other operating systems and helps the developers catch configuration errors early. Since database configuration parameters tend to change more than application code, these parameters are stored in a text file that can be modified by site administrators after the target application is compiled. The Hypersonic driver, as seen in the screen shot below (Fig. 17), is an internal memory-only database used for development. The Roo tutorial video makes use of this driver, and it is therefore supported by JScaffold as well. This driver accepts no configuration parameters.
The third tab, Entities/Fields (Fig. 18), is used to define (Figs. 19-20) the data classes for the Roo-generated application. This tab is the most important in the application because the majority of generated code is derived from the elements specified here. The left side of the tab is used to define entities, which are the data classes themselves. Entities are defined by a class name, which is a Java class name and can be optionally specified as a member of a package, an optional superclass, a flag indicating the class should be abstract, and the name of the database table used to store the instances of this entity class.
Fig. 18: Empty entities/fields tab

Fig. 19: Adding a new entity to the project model
After selecting the entity on the left side of the Entities/Fields tab, the right side will become active and the user may begin adding, modifying, or removing fields from the selected entity. Entity classes initially have no fields, and it is not a requirement of Roo for them to have fields. Both in the case of adding a entity and adding field to an entity, certain attributes are automatically set by the GUI. Each operation will assign a predefined name to the new entity/field, focus the cursor on the name, and highlight the name, allowing the user to simply type over the placeholder. Modifying the name of an entity or field will also immediately update the corresponding entry in the entities or fields list.

An entity may contain zero or more fields. The management interface for fields operates with the same behavior as the entities interface. Updates are immediately applied to the selected field just as they are with entities. The field type attribute is
populated with several types supported by Roo, but also allows user defined types to be entered into the GUI field. If a Roo type is selected, the GUI will present a customization panel for the field selected (Figs. 21-23).

After adding the first entity, the user may choose to click the Add Entity button again to create a second (or third, etc.) entity to the project model (Fig. 24). When one or more entities exist in the project, the superclass field is automatically populated with existing entity types, allowing the user to select from a list instead of re-typing the name of the superclass (Fig. 25). This allows the user to easily create entity types that inherit fields from other entity types, both in the database and the object model.

![Fig. 21: Selecting a Roo type in the fields interface](image)
Fig. 22: Configuring type-specific attributes for the String type

Fig. 23: Adding multiple fields to an entity
Fig. 24: Adding a second entity to a project

Fig. 25: Creating a derived entity class
Roo supports a special enumeration field type, known as enum. This type of field can be assigned from a limited set of values (Fig. 26). This is similar to enum types from SQL, C, and Java 5.0. When the enum type is selected from the dropdown list, a multi-line text field will appear in the customization panel allowing the user to specify the enum's possible values. The behavior of the enum type differs slightly from other types due to the additional steps necessary to define an enum in Roo. Roo defines enums and enum fields in separate stages, the GUI interface does this in one stage.

![Figure 26: Defining an enumerated type](image)

The next tab (Fig. 27) defines controllers in the project model. Roo automatically generates a view in the form of several JSP web pages for every controller as well as the necessary logic to allow the create, read, update, and delete operations from a web interface. The controllers also implement the REST pattern to transport data between the view and controller components. The interface of the controller tab operates similarly to
the entities/field tab with an additional feature, the Auto-Generate Controllers. In Roo, it is possible to instruct Roo to automatically generate controllers from all entities in the project model. The Auto-Generate Controllers button does not use this feature from Roo. Instead, it scans the project model defined in the GUI and populates the controllers tab with classes named after the entities (Fig. 28). An alternative to this method is to declare controllers individually using the GUI. Each controller has a controller name, which is a Java class name. Each controller may be associated with an entity class, however this is not required by Roo. Controllers also use a base path to implement the REST pattern and permissions for a set of operations that may be performed on the respective entities.

Fig. 27: Empty controllers tab
The next two tabs are not used for the generated Web MVC application, but are useful for programming the application, and each represents a feature available from Roo. The first tab, Properties (Fig. 29), is used to manage Java “properties” files. These are text files used to store configuration variables and can be easily parsed and modified using a Java library class. Properties files are a cross-platform alternative to other configuration systems such as Windows registry. Properties files also use a human-readable format, allowing easy modification after the file is created. The GUI application uses Roo to generate properties files rather than directly creating the files using the Java library.
After a properties file is selected (or added), the table interface is used to create entries in the file (Fig. 30). These entries are typically referred to as properties, and each consists of a key/value string pair. A blank line is maintained by the GUI to allow new entries to be typed directly into the table. Properties can be removed by clearing the name from the table. Otherwise, modifications to the table are instantly saved to the project model.
The final tab in this application is the Messaging tab (Fig. 31), and is used to declare destinations and recipients for JMS (Java Messaging Service). JMS is an asynchronous message passing library and service used to pass messages between components in a Java application. This service is often used for transactional applications and to queue messages for slower components. The messaging is broken into two halves like Entities/Fields. The left side defines messaging destinations, which are string identifiers used by other components to route the messages. Each destination may have one or more recipients, defined on the right side, which are Java classes that receive messages for the destination (Fig. 32). The messaging service is not used by the generated web-based applications, but can be programmed after the code generator is finished.

Fig. 30: Adding a property to a properties file
Messaging destinations can receive asynchronous messages sent from other parts of the application. These messages are delivered to a set of recipient classes defined for each message destination. Use of this form is optional.

**Fig. 31: Empty messaging tab**

**Fig. 32: Example destination and recipients defined**
The final interface in the GUI application is the code generation window (Fig. 33). This window displays a comprehensive listing of the project model (Fig. 34) and the list of commands passed to the Roo interpreter (Fig. 35). The Generate button will perform the necessary operations and generate a script for Roo to execute, then it will execute Roo in a separate console (Fig. 36). This allows the programmer to review any errors reported by Roo. Clicking the Compile button will instruct Roo to compile and package the target application (Fig. 37) ready for deployment to an appropriate Java EE web container such as Apache Tomcat. This is expected to be the last step in using the GUI application, from this point the programmer will begin customizing the source code in the target application.

Fig. 33: Empty code generator output window
Fig. 34: Top half of code generator output displaying the project model summary
Fig. 35: Bottom half of code generator output displaying the Roo script
Fig. 36: Roo output displayed in a terminal emulator
Fig. 37: Roo compile output
7.3 Address Book Demo

After the Generate and Compile buttons are used, Roo generated a web application named TestProject. The source code for this application is contained within a single directory that can be loaded as an Eclipse or Spring Tool Suite project. It contains a project model that can be used by Apache Maven to build compiled code, documentation, and an archive for deployment to EE web servers such as Apache Tomcat. The structure of a project directory resembles those created by IDEs. Resources and source code are isolated, Java code is separate from JSP code, and a build/test environment is set up.

The previous section demonstrated the use of JScaffold (Figs. 14-37) in the creation of a simple Address Book application. Further details were provided in this section. The address book is made up of 3 tables, Person, Organization, and OrgPerson (Organization Person), each of which containing a set of fields and several web interfaces in order to provide CRUD access. Roo generates all of the necessary source code and JSP files. The interface is made up of a welcome page (Fig. 38), a toolbar on the left side of the page to add and list entries in each table. The interface used to add new entries (Figs. 39-40) to a table consists of text entry fields and a submit button. This form will report any errors, such as entering a string for an integer field, and provides hints to the user when a field is selected. The interface used to view a table (Fig. 41) uses an HTML table to display the entries for that table, along with buttons to read, update, and delete those entries. In a later example (Fig. 42), an Organization-Person (OrgPerson object) is created. The form contains all inherited fields from Person, the base class of OrgPerson, and a dropdown for the allowed enum values for the Department field.
Using the same process from the restaurant application, this address book application would likely take weeks or months to write by hand, whereas an experienced Roo user or novice JScaffold user could generate much of the same program in minutes.

Fig. 38: Target application welcome page
Fig. 39: Adding a new contact
Fig. 40: Viewing a contact
Fig. 41: Listing contacts in a table view
7.4 Peer Review

The final stage in development of this application was to submit the application to other programmers for an informal review and comparison with Roo. Six programmers participated and submitted feedback about the GUI application as it compares to using Roo. We already use the Spring Framework in our environment, but do not make...
significant modifications to the application's usage of Spring. Four of the programmers
exercised the Roo and JScaffold interfaces, two observed.

The programmers first watched the 5-minute tutorial video on Roo available from
www.springsource.org, then performed the same commands in Roo to build their
eexample address book application. After this was finished, they then used JScaffold,
without any direct instruction on how to use the GUI, to create an address book
application with the same entities, fields, and controllers as the Roo demo video. The
programmers were then given a survey of their experiences with Roo and JScaffold (Figs.
43-44). They were asked to provide some information about themselves, specifically their
familiarity with the entities/fields concept and their use of command line or console
applications. They were then asked to rate on a scale of 1 to 5 for visibility of features,
the quality of the hints provided by each application, and the amount of time spent in
each application. Finally, each developer was asked about their opinion of Spring, Roo,
and JScaffold, and surveyed on their likeliness of using each tool in future projects.
Fig. 43: Peer review survey (page 1)
8. Were the hints and control tooltips helpful in describing a particular feature of J Scaffold?

___ (1 = not helpful, 5 = very helpful)

The next 2 questions are about a hypothetical future software project. This project is database-driven and uses web-based user interfaces.

9. Would J Scaffold make you more likely to use the Spring Framework in your application?

[ ] Yes [ ] No

10. The Roo program supports many additional commands not available to J Scaffold. Examples include e-mail communication setup, web flow, and test-driven development. In contrast, J Scaffold allows you to review your project before committing it to the code generator, reducing typos and other user errors during the design process.

If you were to choose between Roo and J Scaffold, which is more likely?

[ ] Use Roo exclusively

[ ] Use J Scaffold exclusively and implement the other features by manual coding

Use the space below to write any additional comments:
The survey (Figs. 43-44) responses from participants P1-P6 for questions Q1-Q6 (Table 2) can be summarized as follows:

- Regarding the visibility of available features in Roo, the responses varied but were all rating lower than in JScaffold. The visibility of the features available in JScaffold were rated highest.
- The ratings of quality of the hints provided by Roo varied. The hints provided by JScaffold were rated highest by all of the programmers except for one.
- Two programmers took longer to complete the exercise in Roo than the same exercise in JScaffold. One programmer took the same amount of time for each, and one took longer using JScaffold.
- All programmers indicated that they would be more likely to use Spring in a new application due to the availability of JScaffold.

---

Table 2: Responses from peer review

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Frequently</td>
<td>Occasional</td>
<td>Occasional</td>
<td>Occasional</td>
<td>Occasional</td>
<td>Occasional</td>
</tr>
<tr>
<td>Q2</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Unsure</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Q3 [min.]</td>
<td>6</td>
<td>25</td>
<td>Observed³</td>
<td>Observed</td>
<td>DNC⁴</td>
<td>3</td>
</tr>
<tr>
<td>Q4 [grade]</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Q5 [grade]</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Q6 [min.]</td>
<td>12</td>
<td>5</td>
<td>Observed</td>
<td>Observed</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>Q7 [grade]</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Q8 [grade]</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Q9</td>
<td>JScaffold</td>
<td>JScaffold</td>
<td>JScaffold</td>
<td>JScaffold</td>
<td>JScaffold</td>
<td>Roo</td>
</tr>
</tbody>
</table>

³ This programmer observed another programmer exercise the Roo and JScaffold test
⁴ This programmer did not complete the Roo test
• Five of the programmers indicated that they would prefer JScaffold over Roo, knowing that some features are not supported by JScaffold and would later need to be implemented directly by the programmer.
7.5 Advantages of JScaffold

The GUI interface provides several advantages over Roo:

- Simplified and intuitive interface for inexperienced users.
- Roo commands are executed at the end of development, not during development.
- Ability to correct user mistakes before committing them to the code generator.
- Proactive error detection for some error conditions.
- Enforced uniqueness in identifiers.
- Enforced naming convention for Java class names.
- Fewer keystrokes when using keyboard shortcuts
- Simpler representation of complex Roo elements, such as enums and JMS.

7.6 Disadvantages of JScaffold

- Many features supported by Roo are not implemented in JScaffold.
- Mouse clicks are required in some places. This may slow down advanced users compared to the Roo console.
- Roo can be used multiple times while developing an application. The current design of JScaffold only allows the creation of new projects.
- Roo errors are caught immediately while using the Roo console. In JScaffold, errors may not be caught until the final step.
8 RELATED WORK

8.1 Introduction

This chapter describes two other RAD solutions, Grails [9] and Iron Speed Designer (ISD) [10]. Although ISD was mentioned earlier, it was only in the context of a particular feature, the application wizard. J Scaffold is similar to ISD with respect to the application wizard interface only; this chapter describes many of the other features available from ISD that were not ported to J Scaffold [10].

Grails is another RAD solution and shares several similarities with Roo. However, Grails is also a complete development environment that includes a build system, application server, and language enhancements to Java as well as a RAD component. This chapter presents an overview of Grails and contains an address book demo application of its own, created and executed in the Grails environment [9].

8.2 Grails

Grails [9] is another application framework. It was originally known as Groovy on Rails after the programming language on which it is based, Groovy. Groovy [8] is a programming language built on top of Java and, as a result, it is compatible with many existing Java technologies. This compatibility is available at compile time,
meaning that most Java programs can be built and executed under Groovy. SpringSource acquired the original company behind Grails and is the current maintainer [8][9].

Groovy includes several features not found in the Java language and makes some modifications to the Java syntax. Some of the features provided by the Java library, such as regular expressions, ranges, and collections (lists, maps, etc.) are natively supported by the Groovy syntax. The most notable feature in Groovy is the addition of closures, a feature that has been promised for later versions of Java but is currently not available [8].

Roo users will likely find themselves in a familiar setting with Grails. Grails includes a command-line utility, named grails, which operates in a similar way to Roo. The grails tool is used to initialize project directories, create generic entity classes, and create controllers and web views for the entity classes. Unlike Roo, the grails tool is more integrated with the development process than an RAD tool, and is strongly recommended by the developers to manage the complex and mandatory directory structure that Grails uses for each project. The grails tool may create an entity class, but the fields for this class must be manually entered by the programmer. After this class has been modified, the grails tool is run again to generate supporting code such as the web MVC. Grails also comes integrated with a build and deployment system, the Jetty web server for running applications, and the Hypersonic database driver for in-memory and on-disk data storage [9].
The following code is derived from the Grails introduction, and is used to generate a fully-functional (in the same way as Roo) web MVC application in Grails:

```java
package trip.planner

class Trip {
    String name
    String city
    Date startDate
    Date endDate
    String purpose
    String notes

    static constraints = {
    }
}
```

In this case, the first and last sections of the code are generated by the `grails` tool. The 6 fields in the middle of this code are added by hand, then `grails` is run again to process this file and generate supporting code and interfaces. The application shown in the next 3 screenshots are the generated and compiled version of this application. Grails generates entities/fields, controllers, and web views individually, and displays a welcome page with the list of controllers available in the application (Fig. 45). Each controller is given a generated web interface with the same CRUD functionality as Roo applications (Figs. 46-47) [9].
Welcome to Grails

Congratulations, you have successfully started your first Grails application! At the moment this is the default page, feel free to modify it to either redirect to a controller or display whatever content you may choose. Below is a list of controllers that are currently deployed in this application, click on each to execute its default action:

Available Controllers:
- trip.planner.TripController

Fig. 45: Grails target application welcome page
Fig. 46: Listing entities in a table in the Grails target application
Fig. 47: Adding a new entity in the Grails target application
8.3 Iron Speed Designer

Another RAD tool mentioned earlier is Iron Speed Designer (ISD) [10]. This tool is unrelated to Java, Spring, Rails, and most of the other technologies that have been mentioned so far. This tool, and the applications generated by it are entirely based within Microsoft platforms, such as Windows, .NET, and IIS. ISD is a commercial product and boasts over 200,000 users according to its website. It is usable by both programmers and non-programmers, due to its rich visual interface emulating familiar features from other GUI applications [10].

In some ways, ISD resembles a fully-functional IDE like Visual Studio. ISD allows forms to be re-opened and modified graphically. Form elements are given properties and data bindings as is with other IDEs. Target applications generated by ISD are more complex than Roo and can support a number of additional features compared to Roo [10]. These features include:

- Table view
- Sorting
- Table view
- Menus
- Pagination
- Searching and filtering
- Validations
- Calendar entry forms
• Integration with Microsoft products (Word, Excel, etc.)

The first interface used to generate applications in ISD is a sequential set of interactive prompts to the user, commonly known as a wizard interface. This is used to establish page style, database connection parameters, fields, and application properties. After a project is initially generated, the main interface for ISD allows customization of the elements of the target application. The customization interface uses grid and property lists similar to IDEs and other GUI applications. An embedded text editor with syntax recognition (highlighting, indentation) and can be used to further customize the web by directly modifying the generated source code [10].

While this project did not attempt to emulate ISD, some of the GUI features were inspired by the application wizard interface in ISD. JScaffold contains a single window with tabs as the main interface of the application. This interface sets all parameters and data objects and encourages the user to operate in sequence, from setting up the initial database parameters to configuration any optional elements in the target application. Since JScaffold is largely dependent on its functionality from Roo, its feature set is only a small portion of the features provided by ISD. However, with additional development and less dependency on Roo as a back-end, JScaffold may be able to compete with advanced tools like ISD in the future [10].
After the initial design of JScaffold was determined, several new items were added to the project checklist during development. Several of these features made it into the project incrementally, others did not. The main reasons for why these features are not part of the final version of JScaffold are time constraints and prioritization. The project also contains some bugs that affect the functionality of the GUI.

The GUI contains some artifacts from a load/save feature that was partially implemented. The intent of this feature is to allow the user to save and load project models into a format for use by JScaffold.

There are a few field types in Roo that are not supported by JScaffold. The most notable are set and reference. These are used to represent one-to-many relationships in the database. Support for this field type would likely require a completely new GUI interface in order to establish relationships.

Many of the Roo commands contain parameters that are not supported by JScaffold. These parameters were removed to simplify the interface, however they could be re-added in some form of an advanced options panel. These parameters include messaging queue types, JPA per-field options, and additional Java classes for fields.
Many Roo commands are also absent from JScaffold. These commands enable application support for e-mail communication, web flow, test-driven development, and more. These commands would likely require additional tabs in the main interface, however some of the more advanced commands could be placed in a separate window. An exception is Roo's support for third-party plugins. Since such plugins would not be able to benefit from a graphical interface, it is unlikely that this feature would arrive in future versions of JScaffold.

Currently, only two database drivers are fully supported by JScaffold. Roo supports more drivers, including a disk-based Hypersonic database. Future versions of JScaffold would support all of the drivers supported by Roo.

JScaffold already contains a few methods of error detection and prevention, which are explained in the previous chapter. Some additional user input validations can be added to later versions, such as invalid character prevention. Such errors currently would not be caught until the final stage, when errors are reported by Roo.

Roo forbids the use of SQL reserved words in field names. This is most likely a safety measure to ensure compatibility with multiple SQL implementations. JScaffold currently makes no effort to prevent SQL reserved words from being used in the project model, and such errors would not be caught until the final stage.

Roo allows multiple runs on the same project. Using JScaffold, it is only possible to run Roo twice, once for generating the code, and once for compiling it. Roo also enables some “read” features, such as properties get and properties list. Roo however does not allow multiple use of the project command. Since this command is
always the first command in the generated Roo script, Roo will not run on a project directory that has already been generated. Also, JScaffold does not interact directly with Roo because of Roo's heavy and unavoidable use of ANSI escapes to provide a rich console interface. This problem would need to be addressed before any feedback from Roo can be processed by JScaffold.

The developers from the peer review suggested some additional changes to JScaffold in order to make it more effective as a replacement for Roo. First, an interface to either queue or send commands directly to Roo would allow access to Roo features that cannot be provided by the GUI. This would also have the effect of enabling Roo plugins. Similarly, an editor interface for the queued Roo commands could provide additional customization and forward compatibility with any possible Roo versions that may add more features to existing supported commands. Also, since an advanced user will find Roo more efficient due to its keyboard shortcuts, JScaffold needs to make use of keyboard shortcuts more than it does already in order to compete in efficiency.

Roo provides a level of integration with the Spring Tool Suite (STS) not available from JScaffold. This is achieved by adding an additional tab interface to STS. While JScaffold is not designed in a way that allows easy integration with STS, it is possible to encapsulate the GUI components of JScaffold into an Eclipse plugin. The existing Eclipse plugins used to create STS have demonstrated communication between tab components, suggesting that the same form of communication can be achieved between JScaffold’s tab interfaces and the Roo console tab provided by STS.
Finally, as JScaffold is a desktop GUI application, it is missing three major features seen in many other desktop applications. First, JScaffold makes no use of graphics or icons. In contrast, Iron Speed Designer uses icons and other visual clues to make the program easier for users. Second, a comprehensive help system would help users understand JScaffold's functionality better. Although tooltips and other hints exist already, most of the programmers were confused about how to begin work on an application in JScaffold. Lastly, some user-configurable options exist in JScaffold that are currently set using the Java command line. A preferences interface would be more intuitive in this case.
When choosing a framework, programming language, operating system, or architecture, several factors are considered in choosing a framework. This includes features, costs, development time, quality of documentation, stability, and portability. This includes the Spring Framework and its supporting components: Java, Apache Tomcat, and others. In many scenarios, Spring excels at these factors.

This thesis began as an exploration into the Spring Framework to find areas that were either missing or could be improved. Before I learned about Roo, I had already determined that, with the right tool, Spring was an ideal platform for rapid development. This thesis began with an overview of the components for Java EE development. Many of these components are available from Spring and other similar frameworks. The features provided by these components also be seen in other languages supporting a multi-tier or distributed application model. An overview was then written for the Spring Framework, including many of the differences between Java EE and Spring. Both
frameworks provide the necessary foundation of multi-tier application development, each in a unique way. The structure of the two chapters of background information reflect the component-based architecture of both frameworks. The restaurant example application in Spring was written during my research of Spring in order to contribute to my understanding of the framework. The application itself was partially revised in several areas as I discovered more information, but it still an incomplete demo. An RAD tool such as Roo would have allowed for this application to be completed and enhanced with features often seen in web-based applications. After discovering Roo, and learning of the development decision not to include a GUI, I decided that a graphical front-end was the necessary enhancement to make Roo more competitive with other RAD tools. Thus, JScaffold was developed.

While developing JScaffold, I chose to build an interface similar to the application wizard in Iron Speed Designer for two main reasons. First, ISD's commercial success suggests that a more intuitive interface is likely to attract more users to your program. Any programmer who is familiar with the domain of database-driven applications should be able to use JScaffold with only a shallow learning curve, which can be reduced further with additional polishing of the interface. The built-in hints and tooltips already contribute to the intuitiveness of JScaffold's interface. While JScaffold is not nearly as polished as ISD, it supports the same basic features necessary to build applications of this type. Second, the typical program flow for Roo, ISD's application wizard, and JScaffold are similar in design. Each interface follows the same pattern of initializing a project, defining its database and data objects, and then configuring a user interface for the target
application. The main benefit from all three interfaces is that the user is guided by the tool.

The feedback I received from my peers allows me to conclude that JScaffold is an improvement upon Roo, and that it can make Spring appear more attractive to those interested in rapid application development in Java. While there are advantages and disadvantages to this approach, rapid application development is designed to allow flexibility while shortening the development cycle. Roo itself accomplishes this goal, and JScaffold builds upon Roo to accomplish these goals to an even greater extent.
REFERENCES


