

Regis University ePublications at Regis University

All Regis University Theses

Summer 2006

Building a Robust Web Application

Eric Filonowich
Regis University

Follow this and additional works at: <https://epublications.regis.edu/theses>

 Part of the [Computer Sciences Commons](#)

Recommended Citation

Filonowich, Eric, "Building a Robust Web Application" (2006). *All Regis University Theses*. 322.
<https://epublications.regis.edu/theses/322>

This Thesis - Open Access is brought to you for free and open access by ePublications at Regis University. It has been accepted for inclusion in All Regis University Theses by an authorized administrator of ePublications at Regis University. For more information, please contact epublications@regis.edu.

Regis University
School for Professional Studies Graduate Programs
Final Project/Thesis

Disclaimer

Use of the materials available in the Regis University Thesis Collection ("Collection") is limited and restricted to those users who agree to comply with the following terms of use. Regis University reserves the right to deny access to the Collection to any person who violates these terms of use or who seeks to or does alter, avoid or supersede the functional conditions, restrictions and limitations of the Collection.

The site may be used only for lawful purposes. The user is solely responsible for knowing and adhering to any and all applicable laws, rules, and regulations relating or pertaining to use of the Collection.

All content in this Collection is owned by and subject to the exclusive control of Regis University and the authors of the materials. It is available only for research purposes and may not be used in violation of copyright laws or for unlawful purposes. The materials may not be downloaded in whole or in part without permission of the copyright holder or as otherwise authorized in the "fair use" standards of the U.S. copyright laws and regulations.

Table of Contents

Abstract.....	7
Acknowledgement.....	8
Chapter I – Introduction/Executive Summary	9
Problem: “What is Robust?”	9
The Prototype: “Better Tracking and Feedback”	10
Project Goals	11
Project Scope.....	11
Chapter II - The Model.....	12
“Robustness” Philosophy.....	12
The n-Tier Architecture	14
Adaptability.....	14
Extensibility.....	16
Flexibility	17
Scalability.....	17
Understandability.....	18
Model Design Architecture	18
Presentation Tier (Layer)	20
Services “Business Logic” Tier (Layer)	20
Integration Layer	21
Chapter III - Research & Analysis	23
The Project Goals	23
Requirements / Overview of the Application.....	23
Quick Overview of Prototype.....	24
Use Cases	25
Chapter IV - Systems Design.....	33
Database Implementation.....	33
Design HTML Front End Templates.....	34
Design Class and Concept Models	36
Implementing the Robust Architecture.....	40
Adding Efficiency to the Presentation Layer	44
Getting the Data To and From the Presentation Layer	51
Robust Services / Integration Layers	59
Conclusions	68
Appendix – References.....	75

Table of Figures

Figure 1 - Excel Spreadsheet tracking	24
Figure 2 - Use Case Diagram	25
Figure 3 - Full Schema	33
Figure 4 - HTML mock up of log in page	35
Figure 5 - HTML mock up of Join page	35
Figure 6 - Welcome.jsp system page	36
Figure 7 - System Data Class Diagram	37
Figure 8 - Services Classes model	38
Figure 9- Collection object in Java	40
Figure 10 - Trail.java class snippet	41
Figure 11 - Collection class modified with new collection container type	42
Figure 12 - Collection example in C#	43
Figure 13 - JSP/Struts Presentation Layer example	45
Figure 14 - Code snippet from interfaceTop.jsp	46
Figure 15 - ASP.Net Example	47
Figure 16 - Sample Using Tiles Insert (Malani, 2002)	48
Figure 17 - Tiles Template sample (Malani, 2002)	49
Figure 18 - Applying Tiles Template (Malani, 2002)	49
Figure 19 - Index.jsp for Java/Struts implementation	52
Figure 20 - Struts Form Bean Definition	53
Figure 21 - Struts configuration file	54
Figure 22 - Struts Form JavaBean for login (index.jsp)	54
Figure 23 - Struts Login Action (index.jsp action)	55
Figure 24 - Visual Studio .NET HTML view of index.aspx	56
Figure 25 - onclick command to login button in index.aspx	57
Figure 26 - index.aspx.cs Login_ServerClick method	57
Figure 27 - Sample C# Manager class	60
Figure 28 - Factory.java code snippet	62
Figure 29 - Implementing the Factory with ImplMap.txt	63
Figure 30 - Database Service using Interface in Java	64
Figure 31 - IDatabaseService interface in Java	65
Figure 32 - Java implementation of Manager class	66
Figure 33 - Web.config file in C# / ASP.NET implementation of Factory	66
Figure 34 - Factory in C#	67

Abstract

Change is inevitable. Software applications must be prepared for that inevitable moment by following structured robust software design and architecture. Utilizing popular n-tier architectures and robust philosophies in web applications enables developers to implement robust systems that are prepared for the unknown future. This project highlights and demonstrates robust software development techniques in a prototype web application using an n-tier architecture. The examples are designed to provide a robust philosophy that can be applied to similar robust solutions for other development efforts.

Acknowledgement

I would like to thank my family, Danica, Dominick, and Pyper for their support and willingness to put up the seemingly endless hours put into this project. Your patience and love cannot be measured. Thank you.

I would also like to thank Rob Sjodin, Joe Gerber, and all who have offered words of wisdom and comments. Rob, your advice, comments, and aid throughout the process, as well as willingness to help whenever needed are greatly appreciated. Joe, your inspiring comments and support helped push me when I most needed it. To all, thank you sincerely.

Chapter I – Introduction/Executive Summary

Problem: “What is Robust?”

It is a bad plan that admits of no modification.

— Publilius Syrus, First Century BC

Computers and computer software have become synonymous with nearly everything modern man uses to function on a daily basis. It becomes increasingly difficult to imagine something in the modern world that isn't in some manner controlled via a computer and software (or hardware). And as soon as that technology becomes en vogue, it is replaced by a newer, better version, or a completely different piece of technology. In this rapid-paced environment how are software developers supposed to keep up with the changing demands of not only that technology but also each and every one of the users that must adapt to this pace each day?

Arthur Schopenhauer, a German 18th Century philosopher, once said “change alone is eternal, perpetual, immortal.” This is very similar to the definition of dynamic, which is defined as “changeable; fluid; not steady; in motion.” (Wiktionary, 2006) The concept of dealing with change, essentially creating dynamic software, is one of the key components to constructing a robust system. The Linux Information Project (2005) states that a robust system is one that “that performs well not only under ordinary conditions but also under unusual conditions that stress its designers' assumptions.” Such a system needs to be “general code that can accommodate a wide range of situations and thereby avoid having to insert extra code into it just to handle

special cases.” (2005, The Linux Information Project) Essentially, this describes a dynamic, thinking, almost living entity.

If the famous entertainer, Pearl Bailey’s quote, “we must change in order to survive” is any indication, then robust software is essential to survival in the marketplace. Michael Huhns of a University of South Carolina study suggests that “as software developers, we would like the systems we construct to be robust and not crash. But we can’t make them more robust simply by adding more code, as we add more bricks or steel to make a physical structure stronger.” (2002, Huhns, p. 1) Throwing more code into the mix might indeed fix the problem, but will ultimately result in a jumble of fixes and patches of unrelated and poorly functioning code, whereas a quality design must be rather “heavily influenced by a system’s package relationships [by being] loosely coupled and highly cohesive.” (Knoernschild, 2003)

Using a code-independent model based upon robust software design principles, represents an opportunity to analyze the effectiveness of that model and the robust software concept. Implementing a prototype with an effective robust architecture based upon the aforementioned model allows for the study of a foundation for a robust development philosophy.

The Prototype: “Better Tracking and Feedback”

The prototype is a proposed, *TrailTracker*, which offers a fully customizable approach to tracking any type of running or riding activity. It is designed to specifically offer a trailer runner the ability to track only the data they wish to track, and receive the precise feedback they require. This online system can be used to track personal data, and also view other’s data on similar trails. The prototype itself will

only be completed to a point for the purpose of analysis. Following the completion of the project, the system could be potentially finished and used for “real life” exercise tracking and comparison.

Project Goals

This project includes research and design of a code-independent structure and model of the proposed online application for *TrailTracker*, plus analysis of the robust architecture techniques applied. Java development will be built with Java/JSP/Struts application utilizing Eclipse and run on an Apache Tomcat server. Additionally, .NET development will be built with ASP.NET using C# utilizing Microsoft Visual Studio .NET and run on an IIS test server (Windows XP, Service Pack 2). For the purposes of testing and development, the backend database will be MySQL, version 4.0.20a.

Project Scope

The scope of this project will include demonstration of various robust techniques in a prototype application in an effort to begin developing a robust philosophy. The implementation of the “robust” model into multiple languages will allow for additional in-depth analysis of the applied techniques versus potential changes without inherently affecting every aspect of the code base.

Chapter II - The Model

“Robustness” Philosophy

Wikipedia defines *robustness* as “the ability of the software system to maintain function even with the changes in internal structure or external environment.” (Wikipedia, 2006) It further expands that in the computer software world this *robustness* is the “resilience of the system, especially when under stress or when confronted with invalid input.” The popular website continues with this definition by including terminology such as “system integrity,” “clean design,” and “careful coding.” An example is presented where “an operating system is considered robust if it operates correctly when it is starved of memory or storage space, or when confronted with an application that has bugs or is behaving in an illegal fashion - such as trying to access memory or storage belonging to other tasks in a multitasking system.”

Robust, as defined by Merriam Webster Online (www.m-w.com, 2006), is “having or exhibiting strength” or “strongly formed or constructed.” This is similar and supportive of Vance T. Holderfield and Michael N. Huhns’ research in “A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration” where they define *robustness* “as strong and stoutly built, able to withstand the rigors of normal wear and tear.” (2006, p. 2) In essence, defining *robust* is almost as if one were defining perfection, which, although unlikely, is and should be the design intent of any system. The intent of this project is to develop a model by which the system could be developed with the same *robustness*.

The concept of *robustness* relies on a great deal of work prior to developing and designing a system that is able to think and react while maintaining system integrity.

Rob Sjodin from Regis University identifies three key strategies in developing a large scale system to be:

- User-driven Requirements
 - Architecture-centric Design
 - Iterative Processes
- (2005, p. 3)

The user-driven requirements equate out to defining the system's functionality, or "building the right thing." The second bullet item relates to the correct definition of the solution's form, or better said "building the thing right." The final item identifies the incremental approach of "making it happen." (Sjodin, 2005, p. 3)

An architecture-centric design is a primary key to achieving a *robust* product. Granted, without the proper identification of what the system should accomplish, then regardless of correct and *robust* design the system will fail. Likewise, without an iterative process in place to correctly gauge the development process the ultimate product might be completely out-of-line with the intent of those requesting the system to begin with. Still, "building the thing right" is the absolute essence of arriving at a *robust* deliverable that will be "strongly formed [and] constructed."

As simple as it is to state that a system needs to be *robust*, creating one is much more involved. Several "quality factors" are inherently involved in arriving at a *robust* product when development is finally complete. *Robustness* is not only an ideal deliverable as a product, but it is a course of action throughout the process. The following factors need to be thoroughly designed and implemented to achieve the final goal of a *robust* product:

- Adaptability
- Extensibility

- Flexibility
 - Scalability
 - Understandability
- (Sjodin, 2005, p. 5)

The n-Tier Architecture

A system that fits the *robustness* definition must be *dynamic*. A dynamic system must be adaptable, extensible, flexible, scaleable, and ultimately understandable in order to be *robust*. The concept of a multi-tier, or n-tier, architecture is a common structure that “is executed by more than one distinct software agent, [such as] an application that uses middleware to service data requests between a user and a database.” (Wikipedia, 2006) The multi-tier architecture allows for software modularity by separating out the functionality of objects and classes into multi-tiered groups of common use. These groups (or tiers) such as “user interface, functional process logic (‘business rules’), data storage and data access are developed and maintained as independent modules.” (Wikipedia, 2006) The power lies within the fact that each of these modules can use and be used by any number of other modules within and between the different tiers, but can be modified, upgraded, or replaced independently without directly affecting any of the other modules. This allows for new functionality to be implemented without the worry of causing system wide stoppage or down times. This low-coupled approach adheres strongly to the “robust” requirements of this project.

Adaptability

Adaptability is the ability “to make fit (as for a specific or new use or situation) often by modification.” (<http://www.m-w.com/>, 2006) Since change is inevitable, the more dynamic the application is at modifying and changing the better. Code that is

written for a single purpose within a very tight scope might have its uses, but if that code must be completely redone each time requirements change then the lack of adaptability of that code exhibits inefficiency.

Being able to make modifications without affecting every segment of the system code is very powerful. The system itself is *adaptable* because it can change requirements independently without a complete system re-write or system shut down. The removal of the “hard coding” and tight integration amongst the pieces directly relates to high-coupling¹ versus the low-coupled approach of an n-tier architecture.

Mohamad Fayad says in his article, “Aspects of Software Adaptability,”

“In today’s rapidly changing business environment, adaptability is a critical weapon for survival. Businesses must be adaptable in order to meet increasingly narrow market windows. This need for adaptability at the business level has changed the focus in many businesses from efficiency to opportunity, from reducing costs to generating revenue. For example, an efficient but inflexible system might reduce costs, but might also make it impossible for the business to engage in a new revenue-generating opportunity.” (1996, p. 58)

The importance of adaptable applications cannot be overstated. *Adaptability* itself can be presented in many ways within even a single application. Software can be “self-adaptive [where it] modifies its own behavior in response to changes in its operating environment.” (Subramanian, 2002, p. 52) That same software may be adaptable primarily because of the simplicity with which a change for a new requirement may be made. Additionally, the software must be able to adapt to potential need for new technologies such as adding a new middleware web service to

¹ Wikipedia defines coupling as “the degree to which each program module relies on each other module. With low coupling, a change in one module will not require a change in the implementation of another module.” ([http://en.wikipedia.org/wiki/Coupling_\(computer_science\)](http://en.wikipedia.org/wiki/Coupling_(computer_science)))

the Integration layer (see the Integration Layer section) or upgrading the access technology to the “back-end” database. If the entire system is adversely affected by such a modification or addition then that software is not easily adaptable.

Adaptable software development further reinforces the theory of robustness by emphasizing that “it is no longer acceptable if a software system is correct and solves the problem for which it was designed.” (Fayad, 1996, p. 58) The software must almost be able to see into the future and “grow and change to solve slightly different problems over time [corresponding] to the three stages of the evolution of software development: Build the right thing, build the thing right, and support the next thing.” (Fayad, 1996, p. 58)

Extensibility

Extensibility relates very closely to *adaptability*. Where *adaptability* is the ability to change according to necessity in the future, *extensibility* is the “system design principle where the implementation takes into consideration future growth. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension.” (Wikipedia, 2006) “Extensible describes something, such as a program, programming language, or protocol that is designed so that users (or later designers) can extend its capabilities.” (TechTarget.com, 2006) *Extensibility* is a strong factor in determining the *adaptability* of a given system. It is the ability to extend that given system is based upon “the addition of new functionality or through modification of existing functionality [...] while minimizing impact to existing system functions.” (Wikipedia, 2006)

Flexibility

Flexibility is defined by Merriam Webster Online as “characterized by a ready capability to adapt to new, different, or changing requirements.” (2006) The concept of being a *flexible* application easily relates to both *adaptability* and *extensibility*. An application that can “change with the times” could be termed *flexible*. Fayad describes *flexibility* as “easy to change [a] system’s capabilities in kind. For example, taking something that was a graphical system and making it sensory- or sound based.” (1996, p. 59) The *flexibility* must be inherent and not “on-the-fly,” or the changes made become less *adaptable* and risk reducing the application’s *robustness*. Constantly making software more *flexible* by means that do not fit into the *adaptable* mold risks becoming more like “feature creep” than *flexibility*; which is why Fayad expresses concern, stating that “flexibility is often harder than extensibility, especially when on-the-fly changes are desired.” (1996, p. 59)

Scalability

Scalability refers to a systems ability to grow and expand. The term itself is defined as “capable of being scaled.” (Merriamwebster.com, 2006) By taking a closer look at *scaled*, one notices the use of terms such as “adjust” and “surmount.” The *scalability* of a system is measured by its capability “to increase total throughput under an increased load.” (Wikipedia, 2006) A system can expand (or scale) to meet future requirements such as a larger or more efficient database, or perhaps “contract its resource pool to accommodate heavier or lighter loads.” Other possible dimensions of *scalability* beyond the “load scalability” might be “geographic” in nature such as maintaining a powerful system despite large distances between users,

or “administrative scalability” that can share many vastly different task in a single system that is simple and easy to use and manage. (Wikipedia, 2006)

Understandability

Understandability can be seen as the ability for a user of a system to understand and use the system, or from the opposing view as the ability of a developer (or new developer) to understand and modify that system. The key term is “use.” Whereas a user must be able to use the system or they will not use that system, a developer must be able to understand the system in order to maintain and/or modify it when changes are required. Despite the fact that a system might be able to perform many great and wonderful tasks, if it is not usable then the user won’t use it and those great attributes will never been seen or utilized. This is very similar from the perspective of a developer. As is typically the case in most areas, too much of a good thing is exactly that, too much. Building a robust product that results in very complex and difficult to understand code can produce the exact opposite result, and create a very inefficient upgrade effort when the time comes to do so.

Model Design Architecture

Software architecture is defined on Wikipedia (2006) as “the external interfaces among the system's software entities, and between the system and its external environment.” This definition is pushed further in regards to *robust software architecture* being defined as “one that exhibits an optimal degree of fault-tolerance, backward compatibility, forward compatibility, extensibility, reliability, maintainability, availability, serviceability, usability, and such other ilities as necessary and/or desirable.” (Wikipedia, 2006) The model design for a *multi-tiered*

application such as the web-based prototype for *TrailTracker* will be a system consisting of several different programmatic levels. Since the prototype will be “an application delivered to users from a web server over a network such as the Internet or an intranet,” a method of delivery must be through a web-based portal. (Wikipedia, 2006) A standard browser, such as Internet Explorer, Firefox, or Netscape, allows for easy access to the Internet and everything it has to offer, including the common language of HTML. Using these browsers as a client (thin client) makes web applications a popular choice because of “the ubiquity of the web browser as a client, sometimes called a thin client, [and the] ability to update and maintain web applications without distributing and installing software on potentially thousands of client computers.” (Wikipedia, 2006)

An n-tier model has several advantages, but can also have disadvantages as well. Advantages of an n-tier model range from modifying Business logic “without making changes to either the user interface or the database,” to business objects being used by multiple interfaces, to isolating “the knowledge required in any given tier to that tier.” (Booth, 2006) Additionally, because a system is divided into multiple layers, many developers can code on same project simultaneously since the boundaries are defined, as are the interfaces. On the flip side, an n-tier model introduces a more complex system design, as well as potentially increasing the “memory footprint of the application.” (Booth, 2006) *N-tier systems* are designed to “share the load,” using already existing modules and systems to create efficient and effective complete systems. This “sharing” relates directly to several concepts of *robustness* such as *adaptability*, *flexibility*, and *extensibility* defined earlier.

Presentation Tier (Layer)

The *Presentation layer* is the first thing the user sees when using your application. It is the ‘front line.’ Bad design on the front-end results in a poor user experience and can ultimately doom a system before the user even begins the process of actually using the system. This tier, “which displays the graphical user interface to the end user,” must be clean, intuitive, and fully functional. (Roman, 2002, p. 475)

The *Presentation tier* for a web-based application is typically created using an HTML based interface, using a scripting language and/or server based language backend to communicate data effectively between the user and the layers that “actually do the work.” The web browser is commonly the portal to the first tier.

Services “Business Logic” Tier (Layer)

The *Services* layer is the initial Business logic layer that “services” the user requests. In other words it delivers, manipulates, and sets up the data for the system; preparing that data for what needs to be done with that data in order to return the necessary data to the user based upon that data received by the Services layer. This “Business logic” layer is an “an engine using some dynamic web content technology (e.g., CGI, PHP, Java Servlets or Active Server Pages)” that allows the “logic” to be acted upon. (Wikipedia, 2006) The boundary of an application is roughly defined by the services it offers through a “set of available operations from the perspective of interfacing client layers.” (Stafford, 2006) This layer “encapsulates the application's Business logic, controlling transactions and coordinating responses in the implementation of its operations.” (Stafford, 2006) The Services layer’s role roughly “boils down to policy-driven message routing and monitoring.” (Wainwright, 2005) The actions of the tier itself are moving and preparing the data from the ‘front-end’ to

the ‘back-end,’ “brokering” that data to the “shape it should take in a standards-based, loosely coupled, services-oriented architecture.” (Wainwright, 2005) This bridge from the user entry point to the “back-end” is designed based upon the rules and policies under which the business operation must run.

Integration Layer

Where the Services layer is the servicing and preparation of the data as it travels and is handled from the front-end to the back-end, the Integration layer is the glue that holds the front-end and back-end together. The integration is the silent in-between that allows different internal systems to operate in a seemingly seamless manner to external observer. Microsoft’s MSDN website identifies the Integration layer as “abstracting one system's internals from other systems allows you to change one system without affecting the other systems.” (msdn.microsoft.com, 2006) By “abstracting” the connection of multiple systems an application is given the “ability to limit the propagation of changes is a key consideration for integration solutions where connections can be plentiful and making changes to applications can be very difficult.” (msdn.microsoft.com, 2006) The Integration layer brings together potentially un-related systems, creating a simulated working relationship, allowing “pluggable modules across a network to create distributed, composite applications,” aiming to “connect together pre-existing, self-sufficient applications.” (Bradley, 2003)

Integration is a fine line. *Adaptability*, as defined above, allows for the re-use of segments of code, or entire systems. Integrating together these pieces is not always a simple prospect, nor is it always the best practice. As Microsoft puts it:

A fully integrated enterprise seems to be any CIO's idea of perfection.

Complex interactions between systems are orchestrated through precisely

modeled business process definitions. Any data format inconsistencies are resolved through the Integration layer. Relevant summary data is presented to executive dashboards with up-to-the-minute accuracy. Such visions are surely enticing, but should every enterprise set out to build such a comprehensive and inherently complicated solution?
(msdn.microsoft.com, 2006)

There comes a point in time where integrating existing systems and applications together becomes more of a hassle than a benefit. How much to integrate is a question that must be asked, and must be answered when “deciding how far to go is [the] important step [of] planning an integration solution.” Despite the benefits that could be ultimately achieved through integration of two systems to avoid inconsistent business practices, the effort and delays may override the benefits.

(msdn.microsoft.com, 2006) Loosely coupling the applications together through solid integration can create a powerful system based upon systems that are not dependent upon one another and can therefore evolve independently. A complete understanding of the *total* system for purposes such as debugging can also become very unmanageable and difficult as well.

Chapter III - Research & Analysis

The Project Goals

The ultimate goal of this project is to demonstrate robust architecture techniques through the development of a prototype web-based application using an n-tier architecture. An idea for this prototype web-based application must exist that will provide enough of a structure to challenge implementation. The application in question, the *TrailTracker* system, will be designed to track an athlete's training for any type of time-based exercise for the purpose of general tracking and/or data comparison. The robust techniques used in the development of this prototype are not all inclusive nor do they represent every possible solution or technology available, but rather will demonstrate a philosophy which can be potentially carried on for other future development projects.

Requirements / Overview of the Application

Despite the fact that the *TrailTracker* system will be the prototype utilized to create the base for which to “see the architecture in action,” the entire project will not need to be completed fully to do so. Attempting to build out a complete, fully functional system is far too time-consuming for the scope of this demonstration and analysis. This examination requires enough of the prototype to be constructed to demonstrate and analyze the robust architecture so therefore not all of the use cases in the next section will be completed; only the number of use cases necessary to accurately demonstrate and analyze the necessary robust development concepts will actually be used.

Quick Overview of Prototype

In 1997, Chikkinlegs Solutions founders, the author, Eric Filonowich and his running “buddy,” Jason Vale, encountered their first “trail running experience” in the mountains just west of Golden, Colorado. That initial trek ignited a passion for the rugged style of running. Although their initial efforts were designed to improve health, fitness, and performance in other sporting interests, both became infatuated with the sheer challenge and exhilaration they enjoyed while trail running.

By 2001, both men had finished in the top third of the infamous Pikes Peak Ascent². A burning desire to improve and eventually master the Pikes Peak Marathon pushed the running duo to a more intense training regimen. Their “pencil and paper” tracking data was immediately translated into an Excel Spreadsheet³, but they quickly discovered the need for a more sophisticated data tracking system.

	O	P	Q	R	S	T	U	V	W	X	Y
1 Total Miles:	12/2/2003	12/4/2003	12/7/2003	12/9/2003	12/11/2003	12/13/2003	12/16/2003	12/19/2003	12/22/2003	12/23/2003	12/28/2003
2			16.03			11.57		10.93			15.00
3 Total Bike Miles:											
4											
5 Total Lunges:											
6											
7 Total Run Time	West Trail House	Lookout	House	House	Hogback	House	House	House	House	House	House
8 Total Bike Time	54:39.0	27:28.8	1:25:20	51:53.4	26:56.9	37:44.0	46:29.7	1:00:50	1:03	51:37.9	27:15.0
9 Avg HR	169	167	167	162	168	153	162	167	166	167	167
10 Lift Time											
11 Avg HR											
12 Calories	857	422	1305	752	418	500	675	923	951	768	418
13 Distance	6.19	3	6.84	5.37	3.54	2.66	4.49	6.44	6.46	5.38	3.54
14 Bike Distance											
15 Speed	6.7		4.8	6.2	7.6	4.2	5.7	6.3	6.1	6.2	7.6
16 Pace	8.5		12.29	9.4	7.49	14.13	10.23	9.27	9.48	9.39	7.6
17 Stair Lunges											
18 Around House											
19 House to 128th Corner (1.05 m)		08:14.2		08:42.4	08:15.5		08:42.4	08:10.9	08:23.6	08:08.0	08:20.0
20 HR		171		164	173		169	170	168	166	167
21 Corner to Lake Turn (.26)		02:04.4		02:09.8	01:59.6		02:08.4	02:06.1	02:03.9	02:02.0	02:01.0
22 HR		169		165	168		169	168	170	167	167
23 Lake Turn around Lake and Back (.85)		06:55.3		07:11.9	06:49.2		07:17.0	06:52.9	06:44.5	06:32.8	06:48.0
24 HR		172		166	167		173	174	174	172	167

Figure 1 - Excel Spreadsheet tracking

² <http://www.pikespeakmarathon.org/>

³ Microsoft Excel (*.xls)

Use Cases

The use case diagram in Figure 2 encompasses the requirements of the web-based application that is intended for the *TrailTracker* system. The diagram itself allows for a quick, high-level overview of the potential uses of the system. The different paths are each individually broken down in greater detail in the tables following the diagram. The use cases and diagram will become the basis for the front-end interface, via the web and HTML, to the *TrailTracker* system.

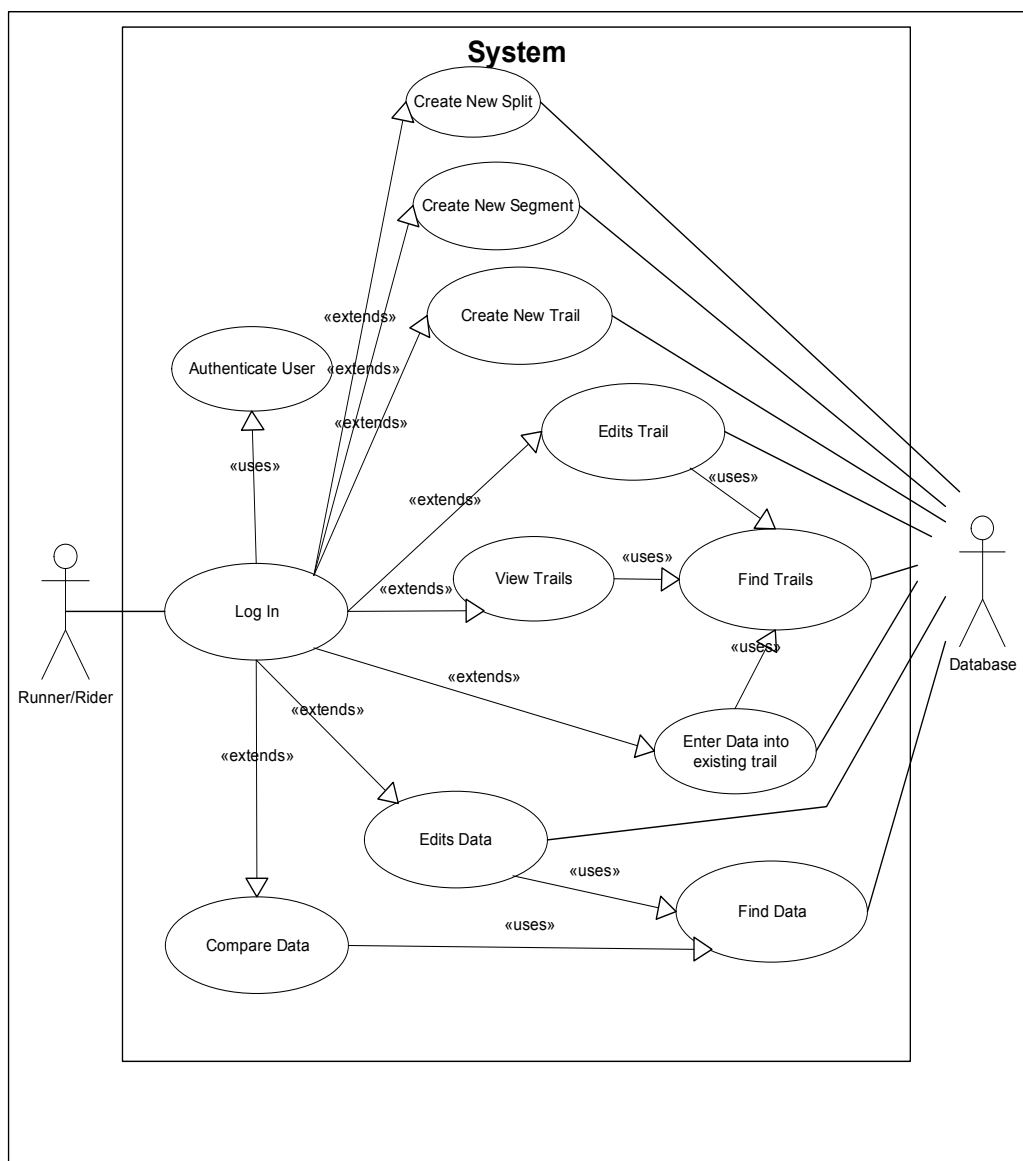


Figure 2 - Use Case Diagram

User logs into system

Use-Case Diagram	TrailTracker System
Use Case Name	User logs into system
Actor(s)	User, Database
Use Case Description	User logs into TrailTracker web application
Initiated by/when	When user arrives at TrailTracker index web page
Terminated by/when	When user successfully enters username and password
Normal Course	<ol style="list-style-type: none"> 1. User arrives at index page of TrailTracker site 2. User enters username 3. User enters password 4. User successfully logs in
Alternate Course(s)	NA
Pre-condition(s)	
Post Condition(s)	User successfully enters system
Assumptions	

User creates new password and username

Use-Case Diagram	TrailTracker System
Use Case Name	User creates new password and username
Actor(s)	User, Database
Use Case Description	User creates new password and username to be able to log into TrailTracker web application
Initiated by/when	When user arrives at TrailTracker index web page
Terminated by/when	When user successfully creates new user information
Normal Course	<ol style="list-style-type: none"> 1. User arrives at index page of TrailTracker site 2. User selects to create new user 3. User enters information 4. User submits information 5. User successfully logs in
Alternate Course(s)	NA
Pre-condition(s)	
Post Condition(s)	User successfully enters system
Assumptions	

User enters data into existing trail

Use-Case Diagram	TrailTracker System
Use Case Name	User enters data into existing trail

Actor(s)	User, Database
Use Case Description	User selects an existing trail and enters a date and data for that trail.
Initiated by/when	User selects option to enter data
Terminated by/when	When user is notified that data has been successfully entered.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to enter data 2. User selects trail 3. User enters split information 4. User repeats #3 until all split information filled. 5. User submits data 6. User notified submission successful
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	Data successfully entered into system
Assumptions	

User creates new trail

Use-Case Diagram	TrailTracker System
Use Case Name	User creates new trail
Actor(s)	User, Database
Use Case Description	User selects option to create new trail
Initiated by/when	User selects option to create new trail
Terminated by/when	When user is notified that new trail has been successfully created.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to create new trail 2. User selects name for new trail 3. User adds split/segments information 4. User repeats #3 until all split/segments information filled. 5. User submits new trail 6. User notified submission successful
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	New Trail successfully added to system
Assumptions	

User creates new segment

Use-Case Diagram	TrailTracker System
Use Case Name	User creates new segment
Actor(s)	User, Database
Use Case Description	User selects option to create new segment
Initiated by/when	User selects option to create new segment
Terminated by/when	When user is notified that new segment has been successfully created.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to create new segment 2. User selects name for new segment 3. User adds start split 4. User adds end split 5. User submits new segment 6. User notified submission successful
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	New Segment successfully added to system
Assumptions	

User creates new split

Use-Case Diagram	TrailTracker System
Use Case Name	User creates new split
Actor(s)	User, Database
Use Case Description	User selects option to create new split
Initiated by/when	User selects option to create new split
Terminated by/when	When user is notified that new split has been successfully created.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to create new split 2. User selects name for new split 3. User adds split information 4. User submits new split 5. User notified submission successful
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	New Split successfully added to system
Assumptions	

User edits trail (only one that they created)

Use-Case Diagram	TrailTracker System
Use Case Name	User edits trail that they created
Actor(s)	User, Database
Use Case Description	User selects an existing trail that they created and edits trail information
Initiated by/when	User selects option to edit existing trail
Terminated by/when	When user is notified that trail has been successfully modified.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to edit existing trail 2. User selects trail that they created 3. User selects what they would like to edit <ol style="list-style-type: none"> a. Trail information b. Splits/segments 4. If 3a: <ol style="list-style-type: none"> a. Users modifies trail information 5. If 3b: <ol style="list-style-type: none"> a. User adds or deletes split/segments b. User repeats #4a until all split/segments edited 6. User submits data 7. User notified submission successful
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	Trail edits successfully added to system.
Assumptions	There are trails that user created to edit.

User creates new trail based upon one created by another user

Use-Case Diagram	TrailTracker System
Use Case Name	User creates new trail based upon one created by another user.
Actor(s)	User, Database
Use Case Description	User selects option to create new trail based upon existing trail
Initiated by/when	User selects option to create new trail based upon existing trail
Terminated by/when	When user is notified that new trail has been successfully created.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to create new trail based upon existing trail. 2. User names new trail 3. User continues through Use Case “User edits

	trail (only one that they created)”
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	New Trail successfully added to system.
Assumptions	There are trails created to copy.

User Views Trails

Use-Case Diagram	TrailTracker System
Use Case Name	User view trails
Actor(s)	User, Database
Use Case Description	User selects option to view existing trails.
Initiated by/when	User selects option to view existing trails.
Terminated by/when	Web page displaying trails loads.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to view existing trails. 2. Web page displays all existing trails.
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	Web page displaying all existing trails.
Assumptions	There are trails to display.

User views/compares data

Use-Case Diagram	TrailTracker System
Use Case Name	User
Actor(s)	User, Database
Use Case Description	User selects option to view/compare data.
Initiated by/when	User selects option to view/compare data.
Terminated by/when	When results are displayed.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to view/compare data 2. User selects Comparison Type: <ol style="list-style-type: none"> a. Trail History b. Segment Comparison c. Split Comparison 3. User selects User Base: <ol style="list-style-type: none"> a. User Only b. All Users c. Specific Users 4. If User selects 2a & 3a: <ol style="list-style-type: none"> a. User selects trail.

	<ul style="list-style-type: none"> b. User goes to #13 5. If User selects 2a & 3b: <ul style="list-style-type: none"> a. User selects trail b. User goes to #13 6. If User selects 2a & 3c: <ul style="list-style-type: none"> a. User selects trail b. User selects users to be included in results c. User goes to #13 7. If User selects 2b & 3a: <ul style="list-style-type: none"> a. User selects Segment b. User goes to #13 8. If User selects 2b & 3b: <ul style="list-style-type: none"> a. User selects Segment b. User goes to #13 9. If User selects 2b & 3c: <ul style="list-style-type: none"> a. User selects Segment b. User selects users to be included in results c. User goes to #13 10. If User selects 2c & 3a: <ul style="list-style-type: none"> a. User selects Split b. User goes to #13 11. If User selects 2c & 3b: <ul style="list-style-type: none"> a. User selects Split b. User goes to #13 12. If User selects 2c & 3c: <ul style="list-style-type: none"> a. User selects Split b. User selects users to be included in results c. User goes to #13 13. User selects date range for results 14. User views results
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	Web page displays results
Assumptions	There is data to compare on an existing trail.

User edits data

Use-Case Diagram	TrailTracker System
Use Case Name	User edits data
Actor(s)	User, Database
Use Case Description	User selects a date on a specific trail to edit the data.

Initiated by/when	User selects a date on a specific trail to edit the data.
Terminated by/when	When data is successfully updated on system.
Normal Course	<ol style="list-style-type: none"> 1. User selects option to edit data. 2. User selects trail. 3. User selects date. 4. User edits data for each split/segment 5. User submits 6. User receives feedback that data has been updated.
Alternate Course(s)	NA
Pre-condition(s)	Successful login
Post Condition(s)	User successfully enters system
Assumptions	Assumes there is data to edit, and trails for which there is data to edit.

The use cases above are just the beginning in developing a robust system. By analyzing the users and how each might interact with the system that system begins to come to life. The analysis presents the desired interaction and reaction with the application and begins to identify the potential path of the data as well as exactly what that data might be. Although perhaps viewed as tedious use cases are an excellent medium with which to bring the developer of the system and those wishing to have the system created onto the “same page.” Nothing can truly be considered less robust than developing the wrong system to begin with. The use cases give a high-level overview of how the application will be used by the end users. Hashing out the “way the system must work” is an essential primary step in developing a robust system. Based upon the study of the above use cases the true system design can begin to take shape.

Chapter IV - Systems Design

The system model is the key to this entire project. The better the model itself is designed, the better the n-tier system will operate and demonstrate the concepts of robustness defined earlier. Three key areas comprise the complete system: the database, the HTML interface, and the class model.

Database Implementation

A robust system would not be so without a robust backend database. The process of normalization has been applied to the 3rd normal form as best as possible. A quick description of the normalization and construction of the database is described in this section.

The database behind the *TrailTracker* system must really be split into two areas, based upon the data identified in the previous section. The schema for the database is illustrated in Figure 3.

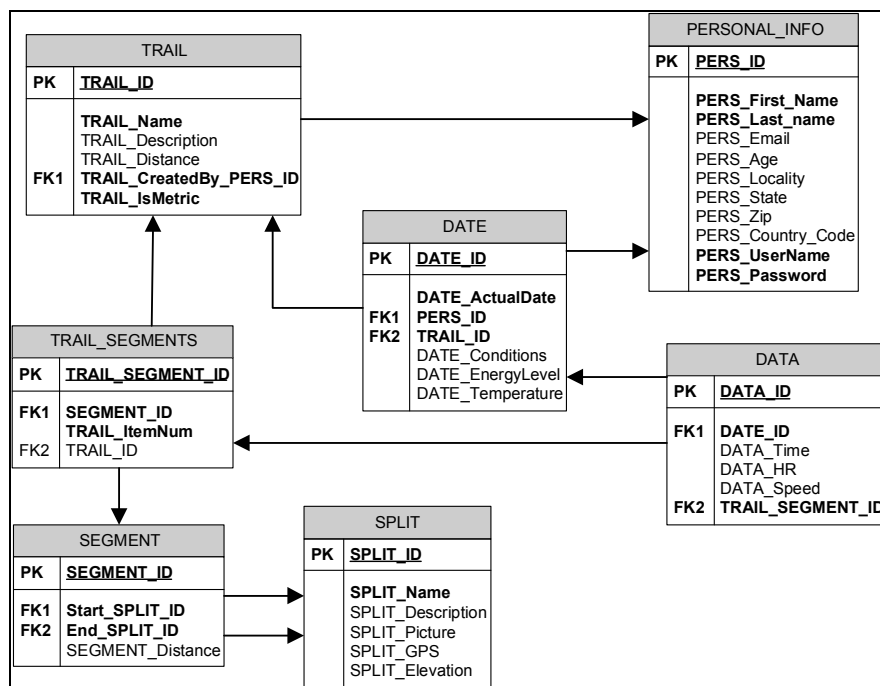


Figure 3 - Full Schema

The first area is the actual trail information, such as trail description and location, splits, and the combination of splits that make up a specific trail. The second is the data related to the actual running or riding of a specific trail on a specific date. The two areas are related because a trail is obviously required in order to run it, yet they must be kept separate in order to reduce duplication of standard trail descriptive information; and ultimately the sharing of data amongst users must be kept at a level that allows for sharing only a trail itself, or allowing for other users to view actual run/ride data as well.

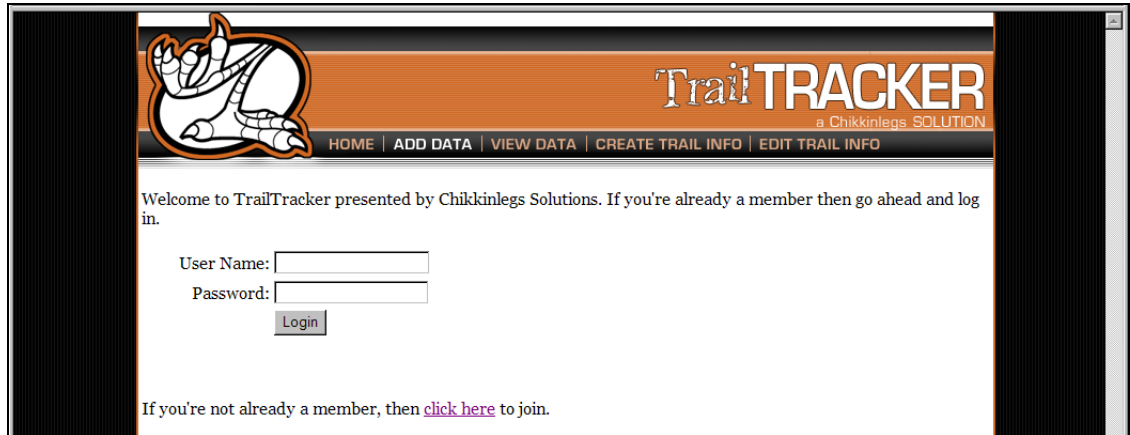
By delineating between SPLITS, SEGMENTS, and TRAILS a user can create new TRAILS based upon already existing SEGMENTS, and likewise new SEGMENTS can be created by using already existing SPLITS. Separating out the granular levels also will allow for better comparisons, as users will be able to not only compare between runs at the trail level, they can then also compare across SEGMENTS or SPLITS even if those SEGMENTS and/or SPLITS reside in completely different TRAILS or SEGMENTS.

The second side of the data within the *TrailTracker* system uses the Trail ID to identify the trail for that specific run/ride. The ID is used to retrieve the TRAIL_SEGMENTS for the specific Trail. For each TRAIL_SEGMENT instance a line item will be added to the DATA table with the appropriate tracking data.

Design HTML Front End Templates

The initial graphic design of the template for the *TrailTracker* website is as seen in Figure 4 below. The screen is designed to fit into a 1024x768 browser window with 800x600 of free space to work with for forms and information. All efforts will

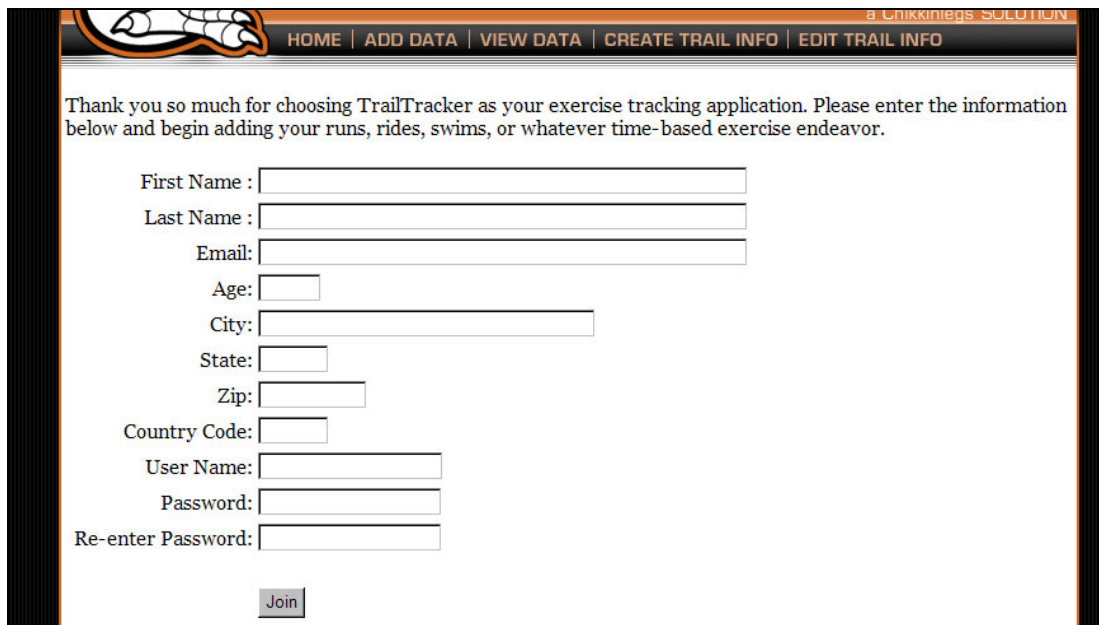
be made to maintain a “scroll bar” free website to avoid unnecessary scrolling, and hopefully maintain better usability. The initial entry point of the *TrailTracker* site/application will be the index page. This front-end will allow the user to either log in, or if not a current user, give the user a link to join.



The image shows a web browser window displaying the login page for TrailTracker. The header features a cartoon chicken leg logo on the left and the text "TrailTRACKER a Chikkinlegs SOLUTION" on the right. Below the header is a navigation menu with links: HOME | ADD DATA | VIEW DATA | CREATE TRAIL INFO | EDIT TRAIL INFO. The main content area contains a welcome message: "Welcome to TrailTracker presented by Chikkinlegs Solutions. If you're already a member then go ahead and log in." Below this is a login form with two input fields: "User Name:" and "Password:". A "Login" button is positioned below the password field. At the bottom of the form, there is a link: "If you're not already a member, then [click here](#) to join."

Figure 4 - HTML mock up of log in page

If a user does not already have a user name and password, the link below the Log In button gives the user a chance to join. After clicking the link, the user is prompted to enter information (see Figure below) to “join” the site, and to choose a login user name and password.



The image shows a web browser window displaying the join page for TrailTracker. The header is identical to the login page, featuring the cartoon chicken leg logo and the text "TrailTRACKER a Chikkinlegs SOLUTION". The navigation menu is also the same: HOME | ADD DATA | VIEW DATA | CREATE TRAIL INFO | EDIT TRAIL INFO. The main content area contains a message: "Thank you so much for choosing TrailTracker as your exercise tracking application. Please enter the information below and begin adding your runs, rides, swims, or whatever time-based exercise endeavor." Below this is a registration form with the following fields: "First Name :", "Last Name :", "Email:", "Age:", "City:", "State:", "Zip:", "Country Code:", "User Name:", "Password:", and "Re-enter Password:". A "Join" button is located at the bottom of the form.

Figure 5 - HTML mock up of Join page

Once a user logs in or creates a new user they will be guided to the *welcome* page. For the sake of this the ultimate goal of this project, the *welcome* will do nothing more than be a simple welcome and display the user's first name and last name as double-check of the database functionality. The simple page can be seen in Figure 6 below.



Figure 6 - Welcome.jsp system page

These pages will build the *Presentation Layer* look and feel by which the system will begin to demonstrate the concepts of robust architecture and development defined in the previous chapter.

Design Class and Concept Models

Based upon the database designs, the essential data will be captured in classes seen in the diagram below (see Figure 7). The classes represented in the diagram nearly mimic the database schematic with the exception of the *Collection* classes.

Although all of the data is encapsulated in the classes, without a collection type container a developer might run across a situation where two (or more) types of containers might be in use and thus represent differing methods of accessing the data within those collections. By “hard coding” in method calls (like *Add* or *Remove*) to a

specific type of container, the code becomes more highly coupled and thus more likely to cause a larger code change if that container is changed.

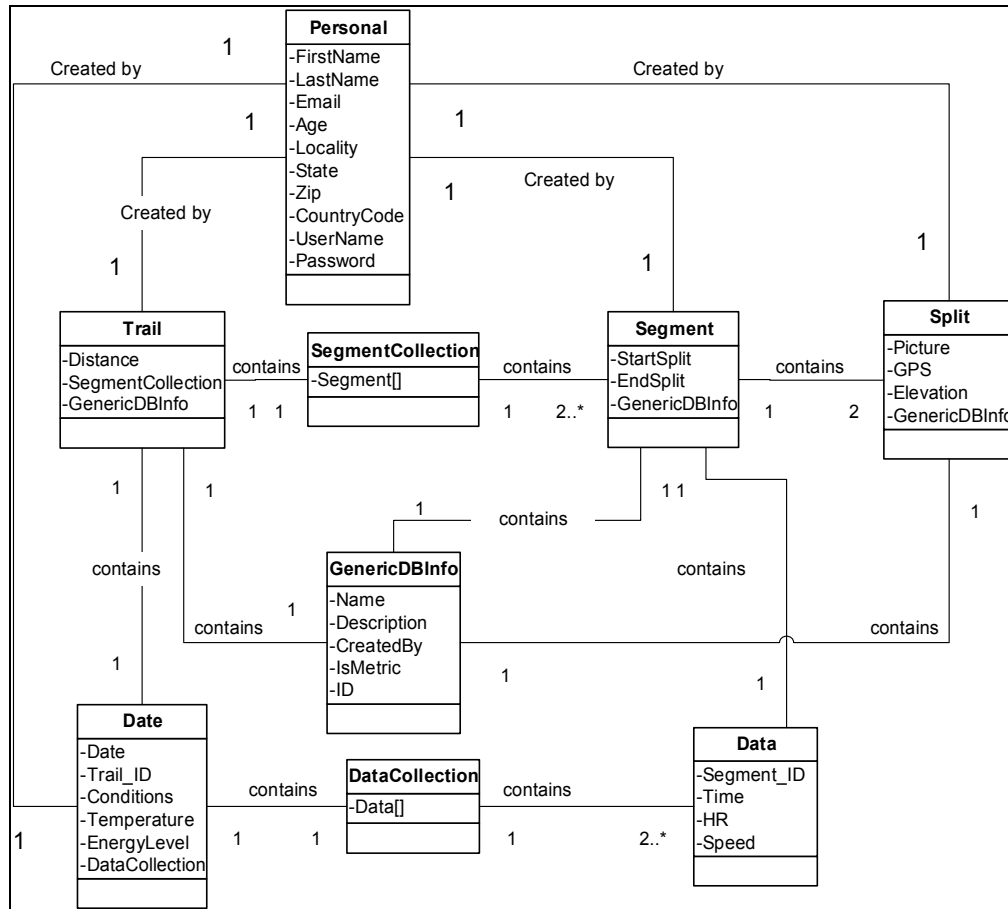


Figure 7 - System Data Class Diagram

By creating an encapsulating collection class as seen in the diagram above, standard accessing methods can be created, and the actual collection class “behind the scenes” becomes irrelevant. This type of *black box* approach allows for the type of container within the collection class to be changed without any apparent modification to the “outside.” An example might be a using an array of Trail objects versus utilizing the more robust collection encapsulation. If, for any number of reasons, the usage of the array is frowned upon or needs to be changed to something like a Java Vector class instead; each location where the code resides will need to be modified. All instances of the array will need to be changed to the Vector object instead, as also

will the access methods and calls. Utilizing the robust model instead, the actual collection type within the collection class can be changed from an Array to a Vector. Only the internal structure of that class would need modified. Externally (outside of the black box), all other objects attempting to access the data would continue doing so in the same manner and notice no difference.

Similarly, the service classes for the system (as seen in the Figure below) utilize the concepts of the robust architecture by separating (or encapsulating) the functionality of each class so as to maintain a low coupling yet high cohesion.

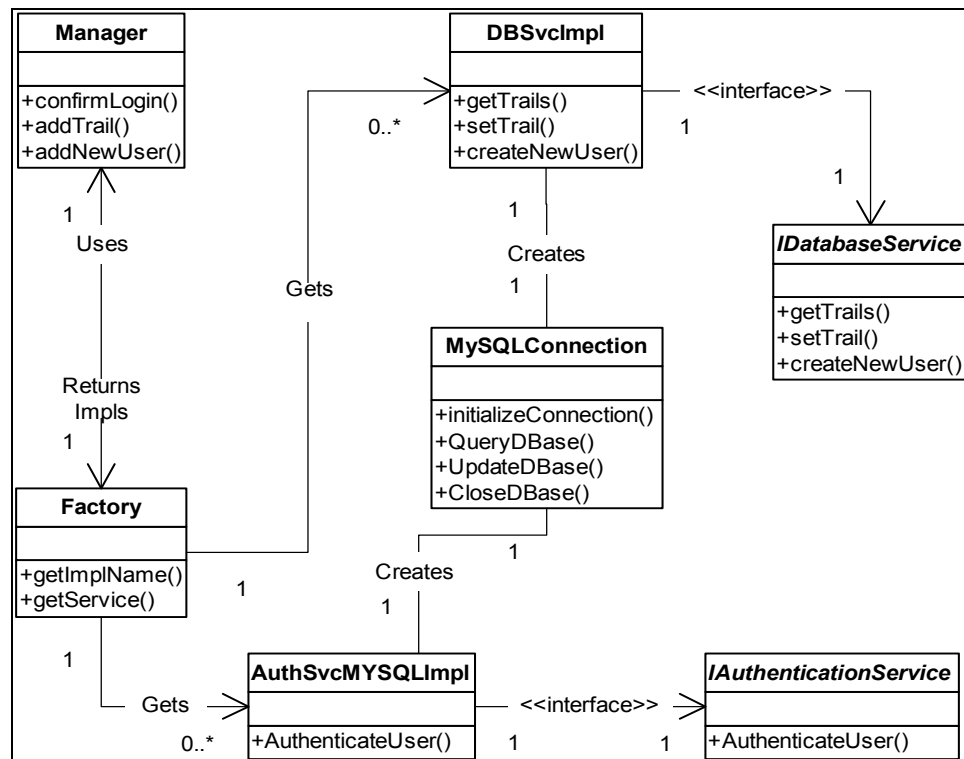


Figure 8 - Services Classes model

Using a Factory pattern, the *Manager* class calls the *Factory* object and is returned the correct database connection object required to communicate with the current database system. The *Manager* class itself does not know any of the details of the type of database being used. The *Factory* uses encapsulated code within itself to generate the connection to whichever database is required and returns that object,

acting as a “factory” of creating database objects. Similar to the data objects above, if the database is swapped out and a different database is being utilized, only the actual database service class will know. The *Factory* will return the type of object needed based upon the type requested. The *Manager* class will expect an *IDatabaseService* interface class and will use the same method calls regardless of if it gets the MySQL implementation class currently in use (see above), or if a new implementation to an Oracle or SQLServer database is returned. The *Manager* class and the *Factory* will not be affected in any manner.

Chapter V - Model Implementation

This section demonstrates the implementation of the robust model developed in the previous chapter through code snippets and the examination of potential changes made to the system. The techniques identified are by no means the only possible solutions, nor do they represent every potential technology or methodology available. They are meant to create a robust philosophy that can be potentially used in future development efforts.

Implementing the Robust Architecture

As mentioned above in Figure 7, tying the type of collection into other code increases the coupling instead of de-coupling acting instead in the exact opposite manner of the robust definition.

```
22
23 public SegmentCollection(Vector segments)
24 {
25     m_vSegs = segments;
26 }
27
28 public void Add(Segment s)
29 {
30     m_vSegs.add(s);
31 }
32
33 public Segment Get(int i)
34 {
35     return (Segment) m_vSegs.get(i);
36 }
37
38 public void Remove(int i)
39 {
40     m_vSegs.remove(i);
41 }
42
43 public void Remove(Segment s)
44 {
45     m_vSegs.remove(s);
46 }
47
48 public void Set(int index, Segment s)
49 {
50     m_vSegs.set(index, s);
51 }
```

Figure 9- Collection object in Java

A better implementation is to encapsulate the collection classes and control the data access, as well as “hide” the collection type. The *Collection* classes are designed to encapsulate the type of collection used to store the *Splits*, *Segments*, or *Trails*, and therefore any type of collection can be used without actually surfacing the type collection to the remainder of the code. This low-coupling approach will enable the implementation to utilize whichever collection type might be more appropriate or more powerful. An example of the *SegmentCollection* class is presented in Figure 9 above.

```
12 public class Trail {
13
14     private int m_iDist;
15     private SegmentCollection m_Seg;
16     private GenericDBInfo gDB;
17
18     public Trail()
19     {
20         gDB = new GenericDBInfo();
21         m_iDist = 0;
22         m_Seg = new SegmentCollection();
23     } //end constructor
24
25     public Trail(int trailID, String trailName, String description, int totalDistance,
26     {
27         gDB = new GenericDBInfo(trailID, trailName, description, owner, useMetric);
28         m_iDist = totalDistance;
29         m_Seg = segments;
30     } //end constructor
31
32
33     public int getTotalDistance()
34     {
35         return m_iDist;
36     }
37
38     public void setTotalDistance(int distance)
39     {
40         m_iDist = distance;
41     }
42
43     public SegmentCollection getSegments()
44     {
45         return m_Seg;
46     }
47 }
```

Figure 10 - Trail.java class snippet

Notice, in Figure 10 above, how the *getSegments()* method in the *Trail* class simply returns the *SegmentCollection* object. This is essentially a custom collection

type that is utilizing an unknown collection container “underneath the hood.” In the code example above in Figure 9 that collection container is a Vector class, but it could just as easily be an array, hash table, or whatever type of collection class is desired and most effective.

```
1 import java.util.ArrayList;;
2
3 public class SegmentCollection {
4
5     //private Vector m_vSegs;
6     private ArrayList m_vSegs;
7
8     public SegmentCollection()
9     {
10         //m_vSegs = new Vector();
11         m_vSegs = new ArrayList();
12     }
13
14     //public SegmentCollection(Vector segments)
15     public SegmentCollection(ArrayList segments)
16     {
17         m_vSegs = segments;
18     }
19
20     public void Add(Segment s)
21     {
22         m_vSegs.add(s);
23     }
24
25     public Segment Get(int i)
26     {
27         return (Segment) m_vSegs.get(i);
28     }
29
30     public void Remove(int i)
31     {
32         m_vSegs.remove(i);
33     }
34
35     public void Remove(Segment s)
36     {
37     }
```

Figure 11 - Collection class modified with new collection container type

Changing and modifying that collection class within the *SegmentCollection* has no effect upon the object accessing and using the *SegmentCollection* class itself. As seen in Figure 11 above, modifying the type of collection object from a Vector class to an ArrayList has no bearing on any of the classes external to the *SegmentCollection* class itself. The “Add” method still appears identical to any object attempting to

access an instance of a *SegmentCollection* object, but underneath the hood the type of collection object could be any type.

Using an example of the same project implemented in C# (see Figure 12 below), the *SegmentCollection* class appears to be identical from an outside class. Each of accessing methods on the collection class are identical to those in Figure 11, but upon closer inspection note the different container object is a *CollectionBase* upon which the entire class is inherited.

```
namespace TrailTracker.valueObjects
{
    /// <summary>
    /// Summary description for SegmentCollection.
    /// </summary>
    public class SegmentCollection : CollectionBase
    {
        public SegmentCollection()
        {
        }

        public void Add(Segment s)
        {
            this.List.Add(s);
        } //end Add

        public void Remove(Segment s)
        {
            this.List.Remove(s);
        } //end Remove

        public Segment Get(int i)
        {
            return (Segment) this.List[i];
        }

        public void Remove(int i)
        {
            this.List.Remove(this.List[i]);
        }

        public void Set(int index, Segment s)
        {
            this.List[index] = s;
        }
    }
}
```

Figure 12 - Collection example in C#

Although the external appearance of the class interface is the same, the internal “workings” are different but completely unknown by the user. This black box approach allows for all of the code surrounding and using these collection classes to

operate independent of the type of collection container within those collection classes. This results in an adaptable and flexible class, because a change whether major or minor, to the type of collection object contained with the collection class requires no modification to external code. This type of low-coupling provides more efficient coding resulting in a more robust implementation.

Adding Efficiency to the Presentation Layer

Robust development doesn't only fall into the bulk of the code in the Service and Integration layers, but can also be practiced in the Presentation layer. One of the primary underlying commonalities of a robust architecture is efficiency. Very similar to the way the collection classes above allowed for the type of collection container to be changed without necessitating changes throughout the code, the Presentation layer code can act in the same manner. The power of the n-tier architecture is in its design to separate out functionality amongst the individual layers. Yet, just by separating out each of the tier's responsibilities doesn't automatically infer that an application is robust. Since the HTML interface is in each page that code is thus duplicated, resulting in duplication of effort and the reverse of efficiency. A better method is required in order to encapsulate the interface design code into a single unit rather than have duplicated code spread across each and every page in the system. As seen in Figure 13 below, the interface HTML code has been removed from the system HTML/JSP code, and is included using the JSP include directive tags.

```

<%@ include file="interfaceTop.jsp" %>
<p><br>
<font face="Georgia, Times New Roman, Times, serif">Welcome to TrailTracker
presented by Chikkinlegs Solutions. If you're already a member then go
ahead and log in.</font></p>
<form action="http://localhost:8080/trailtracker/login.do" method="post" name="login"
<table width="90%" border="0" cellspacing="0" cellpadding="2">
  <tr>
    <td width="17%"><div align="right"><font face="Georgia, Times New Roman, Times
      Name: </font></div></td>
    <td width="83%"><font face="Georgia, Times New Roman, Times, serif">
      <input name="username" type="text" id="username">
    </font></td>
  </tr>
  <tr>
    <td><div align="right"><font face="Georgia, Times New Roman, Times, serif">Pas
      </font></div></td>
    <td><font face="Georgia, Times New Roman, Times, serif">
      <input name="password" type="password" id="password">
    </font></td>
  </tr>
  <tr>
    <td>&nbsp;</td>
    <td><font face="Georgia, Times New Roman, Times, serif">
      <input name="Login" type="submit" id="Login" value="Login">
    </font></td>
  </tr>
</table>
</form>
<p>&nbsp;</p>
<p><font face="Georgia, Times New Roman, Times, serif">If you're not already
a member, then <a href="join.jsp">click here</a> to join.</font></p>
<%@ include file="interfaceBottom.jsp" %>

```

Figure 13 - JSP/Struts Presentation Layer example

The top interface and bottom interface code is no longer duplicated, but rather placed into two separate JSP pages, interfaceTop.jsp and interfaceBottom.jsp, where they can each be imported into each of the system pages using the `<%@ include file=XXXX %>` include directive. Each page, therefore, only contains the HTML code and elements that are necessary for that specific page (in the case of the Figure above, the initial login form). The *interface* code of the Presentation Layer is encapsulated into the two, interfaceTop and interfaceBottom pages, as seen in Figure 14 below. Instead of placing the exact same HTML code for the interface graphics and functionality in each and every page that code resides in a single entity. The table tag `<TD>` at the bottom of Figure 14 is the beginning of the table location where

all system specific operations will be built into the HTML/JSP pages; whereas the closing </TD> tag is located in the *interfaceBottom.jsp* file. This implementation makes modifications to the interface much more efficient and controlled because they are in a single location.

```

<TD>
    <A HREF="index.jsp"
        ONMOUSEOVER="changeImages('Home', 'images/Home-over.gif'); return true"
        ONMOUSEOUT="changeImages('Home', 'images/Home.gif'); return true"
        <IMG NAME="Home" SRC="images/Home.gif" WIDTH=62 HEIGHT=24 BORDER=1>
    </A>
<TD>
    <A HREF="add.jsp"
        ONMOUSEOVER="changeImages('addData', 'images/addData-over.gif'); return true"
        ONMOUSEOUT="changeImages('addData', 'images/addData.gif'); return true"
        <IMG NAME="addData" SRC="images/addData.gif" WIDTH=90 HEIGHT=24 BORDER=1>
    </A>
<TD>
    <A HREF="view.jsp"
        ONMOUSEOVER="changeImages('view', 'images/view-over.gif'); return true"
        ONMOUSEOUT="changeImages('view', 'images/view.gif'); return true"
        <IMG NAME="view" SRC="images/view.gif" WIDTH=98 HEIGHT=24 BORDER=1>
    </A>
<TD>
    <A HREF="create.jsp"
        ONMOUSEOVER="changeImages('createTrail', 'images/createTrail-over.gif'); return true"
        ONMOUSEOUT="changeImages('createTrail', 'images/createTrail.gif'); return true"
        <IMG NAME="createTrail" SRC="images/createTrail.gif" WIDTH=148 HEIGHT=24 BORDER=1>
    </A>
<TD>
    <A HREF="edit.jsp"
        ONMOUSEOVER="changeImages('editTrail', 'images/editTrail-over.gif'); return true"
        ONMOUSEOUT="changeImages('editTrail', 'images/editTrail.gif'); return true"
        <IMG NAME="editTrail" SRC="images/editTrail.gif" WIDTH=128 HEIGHT=24 BORDER=1>
    </A>
</TD>
<TD COLSPAN=2>
    <IMG SRC="images/index_10.gif" WIDTH=105 HEIGHT=24 ALT=""></TD>
</TR>
<TR>
<TD COLSPAN=7>
    <IMG SRC="images/lines_below_toolbar.gif" WIDTH=631 HEIGHT=11 ALT=""></TD>
</TR>
<TR>
<TD width="5" align="left" valign="top" background="images/stage_left.gif">&nbsp; </TD>
<TD height="618" COLSPAN=7 align="left" valign="top" bgcolor="#FFFFFF">

```

Figure 14 - Code snippet from interfaceTop.jsp

Breaking up the interface in this manner allows for the interface and actual system code to be independent of one another. A simple modification to a graphic, link, button, or whatever change required in the interface can be done quickly in a single file and instantly reflected throughout the system without the need for modifying every file. This type of robust architecture can easily be applied to any type of server side scripting. The example in Figure 15 demonstrates the same functionality except in C# and ASP.NET.

```
<%@ Page CodeBehind="index.aspx.cs" Language="c#" AutoEventWireup="false" Inherits="TrailTracker.in
<!--#include file="interfaceTop.aspx"-->
<p><br>
  <font face="Georgia, Times New Roman, Times, serif">Hey Welcome to TrailTracker
  presented by Chikkinlegs Solutions. If you're already a member then go ahead
  and log in.</font></p>
<form runat="server" name="loginForm" id="loginForm">
  <table width="90%" border="0" cellspacing="0" cellpadding="2">
    <tr>
      <td width="17%" height="28"><div align="right"><font face="Georgia, Times New Roman, Ti
        Name: </font>
      </div>
    </td>
      <td width="83%" height="28"><font face="Georgia, Times New Roman, Times, serif"> <input
        </font>
    </td>
    </tr>
    <tr>
      <td><div align="right"><font face="Georgia, Times New Roman, Times, serif">Password: </
        </div>
      <td><font face="Georgia, Times New Roman, Times, serif"> <input name="password" type="p
        </font>
    </td>
    </tr>
    <tr>
      <td>&nbsp;</td>
      <td><font face="Georgia, Times New Roman, Times, serif"> <input name="Login" type="subm
        </font>
    </td>
    </tr>
  </table>
</form>
<p>&nbsp;</p>
<p><font face="Georgia, Times New Roman, Times, serif">If you're not already a member,
  then <a href="join.aspx">click here</a> to join.</font></p>
<!--#include file="interfaceBottom.aspx"-->
```

Figure 15 - ASP.Net Example

Although the method used in the Presentation Layer above are effective, there are also other options that a developer can take advantage of that are perhaps even more effective. Prakish Malani writes in his [JavaWorld](#) article, “UI design with Tiles and Struts,” that the example above using the include directive aids in the robust development, because of the “need to change common view components once.”

(2002)

[This] solution greatly eliminates HTML and JSP code repetition, significantly improving application maintainability. It increases the page number a bit, but drastically reduces the tight coupling between common view components and other pages. On the complexity scale, this solution is simple and readily implemented on many real-world applications.

However, it has one major drawback: if you change how and where you organize the view components (i.e., by changing the component layout), then you would need to update every page -- resulting in an expensive and prohibitive change.” (Malani, 2002)

If, for any reason, the table structure in the template is modified, then the solution becomes more tightly coupled, thus requiring modifications to nearly every file. An even better solution might be through the use of Tiles technology. One option suggests utilizing the Tiles *insert* method with the Tiles tag library, as seen in the Malani’s example in Figure 16 below.

```
<html>
<body>

<!-- include header -->
<jsp:include page="/header.jsp" />

a's body...
<p>

<!-- include footer -->
<jsp:include page="/footer.jsp" />

</body>
</html>
```

Figure 16 - Sample Using Tiles Insert (Malani, 2002)

The example above presents a very similar solution to the samples earlier with the exception of using Tiles. By further expanding on Tiles, the JSP solution earlier could expand to using similar techniques as seen in Figure 16 using the `<jsp:include>` tag to contain the body code in a separate JSP page. This solution allows for “reuse of the bodies in other places, eliminating the need for repetition and duplication, [and thus further diminishing] the coupling between common view components and other application components.” (2002, Malani) The power of Tiles can be seen in their use of templates to control nearly the same exact concept as presented in this project.

By defining a template for pages as seen in the figure below, the structure of the pages is defined through the use of placeholders.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
<html>
<body>
    <!-- include header -->
    <tiles:insert attribute="header"/>
    <!-- include body -->
    <tiles:insert attribute="body"/>
    <!-- include footer -->
    <tiles:insert attribute="footer"/>
</body>
</html>
```

Figure 17 - Tiles Template sample (Malani, 2002)

Malani ties the Tiles template into the Presentation layer in Figure 18 below using the “put” tag into the template defined in Figure 17 above. This could potentially push the robust design of the prototype for this project to an even higher level because “it encapsulates the layout scheme or mechanism, drastically reducing the coupling between common view components and other content bodies.” (Malani, 2002) The problem that immediately comes to mind is the higher level of complexity involved in generating and understanding the Tiles implementation as seen below versus the relatively simplistic original version.

```
<%@ taglib uri="/WEB-INF/tiles.tld" prefix="tiles" %>
<tiles:insert page="/layout.jsp" flush="true">
    <tiles:put name="header" value="/header.jsp"/>
    <tiles:put name="body" value="/aBody.jsp"/>
    <tiles:put name="footer" value="/footer.jsp"/>
</tiles:insert>
```

Figure 18 - Applying Tiles Template (Malani, 2002)

Even though the above examples are in the *Presentation Layer* and do not reflect the “meat” of the coding for the system, applying the *Robustness* concepts from Chapter II leads to more efficient development and control. Any of the solutions allow graphic or navigational modifications to be made quickly in a single location, instantly being reflected throughout the entire system. When compared to modifying the potentially large number of individual files and risking errors the potential for introducing bugs into the system because of a change increases dramatically. Adding even more robustness to the design with Tiles brings up a question of complexity versus implementation. The benefit must constantly be weighed against the understandability in order to maintain an efficient balance. Simply pushing the envelope of robust development without any type of analysis of gains can quickly shift the effectiveness of the solution away from the ultimate goal of creating efficiency. Additionally, simply by implementing an n-tier architecture does not instantly create a robust system. In their study of architecture-based software development, Nenad Medvidovic, David S. Rosenblum, and Richard N. Taylor determine that although “software architectures provide a promising basis for supporting software evolution[,] improved evolvability cannot be achieved simply by focusing solely on architectures,” (1999, p. 52) They conclude that just a “new programming language cannot by itself solve the problems of software engineering, [because it] is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures [...] as tools that also must be supported with specific techniques to achieve desired properties.” (1999, p. 52) The techniques presented in the prototype alone provide a simple solution, but the potential non-robust behavior of breaking the

HTML tables out amongst the top and bottom interface and the content presents a concern. Implementing the Tiles samples would apply robust techniques that would allow the Presentation Tier to operate in an efficient manner that could be quickly updated and modified, but would also add more complexity to the code. If the solution is too complex, all of the robust techniques and the n-tier architecture are nothing more than fancy, complicated, unworkable solutions that do more harm than good.

Getting the Data To and From the Presentation Layer

Remembering what the primary purpose of the Presentation Layer is immediately brings to light two distinct sides of the system. The Presentation Layer is the visual “front” of the system itself, the face of the application. This is the graphical user interface to the data and how that data is manipulated. When it comes to creating an inviting and satisfying web environment the “look and feel” play a very important part in the user experience. Typically, this is the job of graphic designers and not programmers. Likewise, the application coding and system architecture is the place of programmers and not graphic designers; hence, the two distinct sides of the system.

Referring back to the definition of robust development and the n-tier architecture, in order to maintain adaptable, extensible, and flexible code that is scaleable the lowest degree of coupling is generally ideal. When it comes to the Presentation Layer, low coupling is essential in maintaining the “two sides” of the system. Using server side scripting languages like ASP, ASP.NET, JSP, PHP or most others, it is quite simple to integrate “code” into the HTML page itself. Using JSP, simply typing the scripting braces “<% . . . %>” and Java code directly into the HTML code is a

simple “all in one” approach. This, unfortunately, violates the earlier arrived upon definition of a robust architecture. Granted, in a single-man type shop this merging of HTML, graphics, and scripting code isn’t as inefficient, but as the system grows and pages begin to build upon pages, that system and the code interspersed throughout the HTML code becomes unwieldy and unorganized, not to mention very difficult to understand.

Fortunately, many programming languages have developed simple and easier solutions to the highly coupled code problems described above. Implementing a Model View Controller Pattern in Java Struts, or Java Server Faces (JSF), or in ASP.NET allows for very robust development solutions. A Struts implementation on the Tomcat Apache server is relatively painless. Each field in the HTML form is named, for example, the *User Name* field is named *username*, or the *Password* field is named *password*. Adding a simple modification to the form action which calls a Struts action (notice the *.do extension in the form action in Figure 19 below) is all that is needed to tie the form to Struts.

```
ahead and log in.</font></p>
<form action="http://localhost:8080/trailtracker/login.do" method="post" name="login"
  <table width="90%" border="0" cellspacing="0" cellpadding="2">
    <tr>
      <td width="17%"><div align="right"><font face="Georgia, Times New Roman, Times, serif">
        Name: </font></div></td>
      <td width="83%"><font face="Georgia, Times New Roman, Times, serif">
        <input name="username" type="text" id="username">
      </font></td>
    </tr>
    <tr>
      <td><div align="right"><font face="Georgia, Times New Roman, Times, serif">Pa
        </font></div></td>
      <td><font face="Georgia, Times New Roman, Times, serif">
        <input name="password" type="password" id="password">
      </font></td>
    </tr>
  </table>
</form>
```

Figure 19 - Index.jsp for Java/Struts implementation

For each input form that will be used in the system, an entry must be made into the *struts-config.xml* file to map the form submission to a specific Struts action. The

initial entry is seen in Figure 20 below. The *Form Bean Definition* creates a map of a Java bean based upon the page's form data, in the case of the *join.asp* page the `<form-bean>` below labeled "loginForm" maps directly to a Struts bean in the *struts* namespace called *LoginForm*.

```
<!-- ===== Form Bean Definitions -->

<form-beans>
  <form-bean
    name="loginForm"
    type="struts.LoginForm"/>

  <form-bean
    name="joinForm"
    type="struts.JoinForm"/>
</form-beans>
```

Figure 20 - Struts Form Bean Definition

The *name* attribute in the `<form-bean>` maps to an action item in the *Action Mapping Definitions* in the same *struts-config.xml* file (see in Figure 21). The action ties to the submit form tag seen in Figure 20 above. The *action* attribute calls the type "LoginAction" which maps to the *path* attribute "/login" in the *Action Mapping Definitions*, which is determined based upon the "login.do" form action when the HTML form is submitted.

Based upon the Struts configuration file (seen below) the JavaBean is tied to the *action* class for that JavaBean. In the case of the *index.jsp* page, the *loginForm* is tied to the type "struts.LoginAction" which is the *action* for that bean.

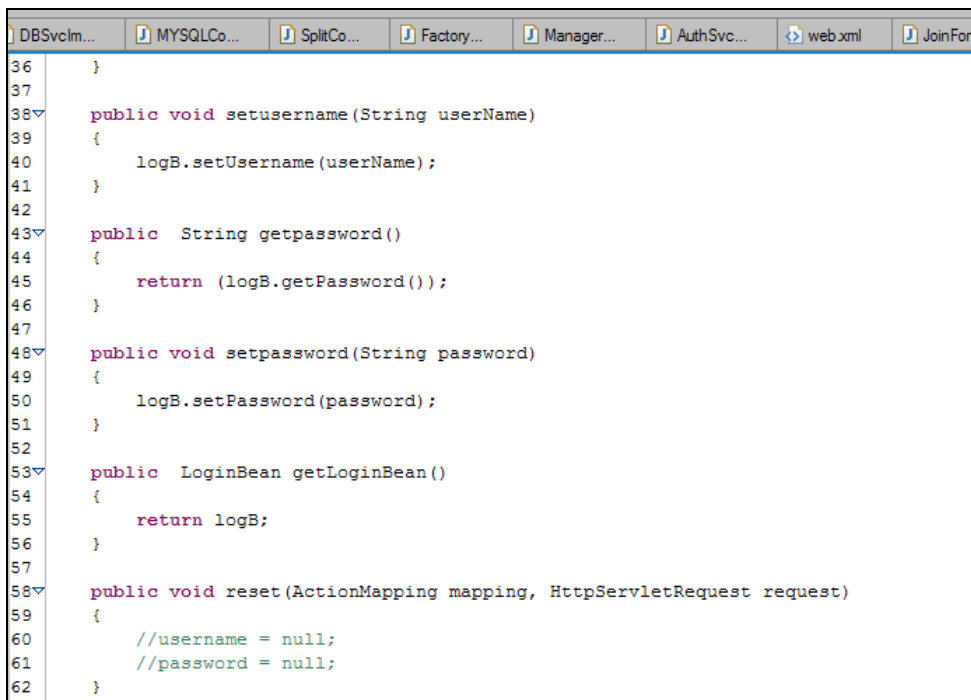
```

<action
  path="/login"
  type="struts.LoginAction"
  name="loginForm"
  scope="request"
  validate="true"
  input="/validation.jsp">
  <forward name="display" path="/display.jsp"/>
  <forward name="edit" path="/edit.jsp"/>
  <forward name="welcome" path="/welcome.jsp"/>
  <forward name="failure" path="/error.jsp"/>
</action>

```

Figure 21 - Struts configuration file

The *LoginAction* class extends the Struts *Action* class. Upon submitting the form and calling the appropriate “*.do” Struts action on the server, the form’s data is automatically added to the *LoginForm* bean based upon the “sets” and “gets” in the bean itself (see Figure 22 below). Struts automatically maps the *LoginForm* bean to the submitting page’s form based upon the *action* and *bean definitions* in the *struts-config.xml*. The code snippet below shows the names of the form fields pre-pended with a ‘get’ and a ‘set.’ This data is added to the *LoginBean* object.



```

36     }
37
38     public void setUsername(String userName)
39     {
40         logB.setUsername(userName);
41     }
42
43     public String getPassword()
44     {
45         return (logB.getPassword());
46     }
47
48     public void setPassword(String password)
49     {
50         logB.setPassword(password);
51     }
52
53     public LoginBean getLoginBean()
54     {
55         return logB;
56     }
57
58     public void reset(ActionMapping mapping, HttpServletRequest request)
59     {
60         //username = null;
61         //password = null;
62     }

```

Figure 22 - Struts Form JavaBean for login (index.jsp)

The *execute* method is then call on the *LoginAction* class where the login data is extracted from the *LoginForm* bean (see line 42 in the Figure below). Through this *execute* the code gets into the “meat” of the system.

```
29 public final class LoginAction extends Action {
30
31     public ActionForward execute (ActionMapping mapping,
32                                   ActionForm form,
33                                   HttpServletRequest request,
34                                   HttpServletResponse response)
35     throws IOException, ServletException {
36
37         try
38         {
39             //create login rules object
40             Manager theChecker = new Manager();
41             //get bean from strut form
42             LoginBean lb = ((LoginForm)form).getLoginBean();
43             //get student vector from rule confirm
44             Vector trails = (Vector)theChecker.confirmLogin(lb);
45
46             //student array object will be null if login fails
47             if(trails != null)
48             {
49                 //place student collection in session
50                 request.getSession().setAttribute("Trails", trails);
51                 request.getSession().setAttribute("Login", lb);
52             }
53         }
54     }
55 }
```

Figure 23 - Struts Login Action (index.jsp action)

The JSP/Struts implementation of the *index* page (index.jsp) is simple. The only two tags that are specific to JSP are the directive tags at the top and bottom. None of the actual form elements are affected, other than the *action* URL pointing to the “*.do” location to initialize the Struts. The separation from the actual Java code is relatively seamless and simple when distinguishing between perhaps a pure graphic designer and a java code writer. The data is automatically added to an existing JavaBean through the ActionForm interface.

As indicated in the Adding Efficiency to the Presentation Layer section earlier, additional technologies can easily be used to push the robust design of a system. Again, Tiles allows for even more efficient operation but potentially at the cost of simplicity. Struts and Tiles can be tied together in a manner utilizing the template samples above. Using the combined power of Struts in addition to the increased

robustness offered by Tiles could create an even more robust system design in the prototype.

An ASP.NET implementation in C# is also quite simple and likewise offers low-coupling, although it is more operation system specific. Whereas Struts allows for data beans created that match the form's data items' names, the .NET utilizes the server side controls to get the data by mapping the controls to an object on the server. This is simply done by right clicking the element in Visual Studio in the HTML visual view and making it server-based (see below).

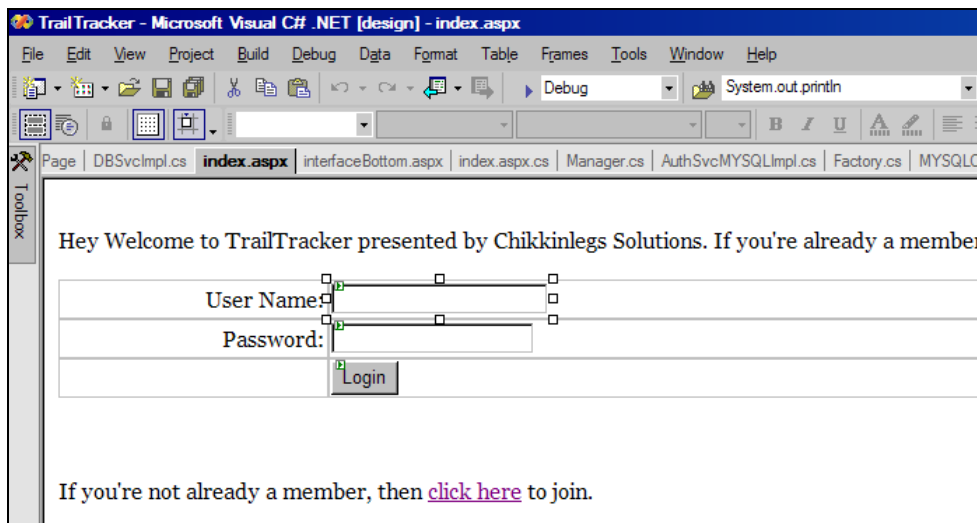


Figure 24 - Visual Studio .NET HTML view of index.aspx

Submitting the data is different than the automatic nature offered in Struts. By making the page control server-based (notice the green arrow in the upper left of the text boxes below indicating the control is server-based), the data is made accessible to the code when the form is submitted. Adding the *onclick* command to the *login* button on the form in the figure above (see figure below for HTML code) and adding the *runat="server"* code make the form ready to submit the data to a method in the *index.aspx.cs* code class.

```
<input name="password" type="password" id="password" runat="server">

<input name="Login" type="submit" id="Login" value="Login" runat="server" onclick="ServerTransfer">
```

Figure 25 - onclick command to login button in index.aspx

Whereas the data in the Struts implementation ends up in the JavaBean and then enters the Struts Action code, the C# works similar yet without having to write a bean class. The *Login_ServerClick()* method acts in exactly the same way as the Struts Action implementation. When each HTML form element is assigned as “server-side” in Visual Studio, their data is made available. In the Figure below, notice the line “lb.Username = this.username.Value;” where the data from the form item containing the user name is made available and ultimately placed into the LoginBean class for later use.

```
private void Login_ServerClick(object sender, System.EventArgs e)
{
    try
    {
        //create login rules object
        Manager theChecker = new Manager();
        //get bean from strut form
        LoginBean lb = new LoginBean();
        lb.Username = this.username.Value;
        lb.Password = this.password.Value;

        //get student vector from rule confirm
        TrailCollection trails = (TrailCollection)theChecker.confirmLogin(lb);

        //student array object will be null if login fails
        if(trails != null)
        {
            //place student collection in session
            Session["Trails"] = trails;
            Session["Login"] = lb;

            Server.Transfer("welcome.aspx");
        }else{
            //build site object as error if login fails
            Server.Transfer("error.aspx");
        } //end if

    } catch (Exception ex){
        System.Diagnostics.Debug.WriteLine("LoginAction : Exception caught: " + ex.Message);
    }
}
```

Figure 26 - index.aspx.cs Login_ServerClick method

Other than the few modifications, the HTML code was void of any C# code, thus successfully adhering to low-coupling. Each of the form's items is submitted and can be easily seen and utilized when creating essentially a "bean" object of data. Their respective data can then simply be extracted into the system and used from there. Both systems (either the C# example of Java/Struts example) offer similarly low-coupled solutions for the web application Presentation Tier.

Struts creates a relatively simple implementation by using the ".do" call to the server for easily getting the data from the submitting page and into the actual object structure. The implementation is very unobtrusive to the HTML environment. No complex coding is required. In the case of keeping a low-coupled environment, a graphic designer without any coding experience (other than HTML) could easily build the necessary pages and then point the form's submission to the Struts location. The page itself could easily be developed in an external HTML IDE such as Macromedia Dreamweaver or Microsoft FrontPage. The page itself could easily be modified in Eclipse as well.

The ASP.NET server side calls to automatically add the submitted objects to the server for easy retrieval is very simple as well.. Much of the code generation can be done with an easy click of a button. Tying the code and the form together is very painless.

Both samples demonstrate adaptability by keeping the ASP.NET or JSP coding separate from the HTML interface to the system. This type of separation creates a low-coupled environment that promotes and extensible and flexible system that can be modified independently of the "other half." The use of the Model View Controller pattern with the n-tier architecture maintains the necessary separation and both examples provide the implementation to keep that separation. Additional

technologies such as Tiles' tight implementation with Struts could add further robustness to the system. Overall, keeping this low coupling keeps the development more efficient because of its separation. Again, the focus on robust design must be balanced against added complexity that could interfere with the efficiency of further development efforts due to higher levels of confusion.

Robust Services / Integration Layers

When it comes to the Services Tier it is even more apparent that robust architecture is very important when maintaining efficient updates for the inevitable change. Two areas immediately come to mind when dealing with and updating the code in the system. First, when (not if) changes are required, the less code that must be affected the better; each class that is affected must be recompiled and the more classes that must be changed the more potential for errors and the introduction of bugs into the system. Second, if the code can be written in a dynamic manner as to completely avoid being recompiled at all, why not?

In the previous section the Presentation Tier was examined and examples showed how a robust architecture enabled the efficient and low-coupled system to be implemented. Yet, once the data itself was pulled from the front-end interface and either sat in JavaBean in Struts or in a similar object in the C#/ASP.NET system that code must be managed or the risk of duplicating efforts across each page of the interface is a dire reality. In the Struts *Action* implementation for the Login Form (see Figure 23 above), the data has been moved from the HTML form and into a JavaBean and now the system must act upon that data. A common step might be to begin using all of the individual system classes and objects directly in this *Integration* layer. Unfortunately, this type of approach can quickly lead to duplicated efforts

across each of the forms where data is moved from that respective page. Any type of change that will be required in the future would ultimately demand changes to each and every one of those Struts Action implementations. This would instantly reduce the robustness of the system because of the inefficiency of operating on so many places within the system in order to complete a single change.

Using a Facade type pattern, which is defined as enabling a system to “use a complex system more easily, either to use just a subset of the system or use the system in a particular way,” a manager type class provides a robust solution. (2002, Shalloway & Trott, p. 89) This manager class, similar to the C# example seen in Figure below, aptly called *Manager* provides the “door” or interface to the Business logic within the system.

```
public class Manager
{
    public Manager ()
    {
        //
        // TODO: Add constructor logic here
        //
    }

    public Object confirmLogin(LoginBean login)
    {
        Factory theFactory = Factory.newInstance();
        IAuthenticationService service = null;

        try
        {
            System.Diagnostics.Debug.WriteLine("*****MANAGER");
            //use authentication service to check login
            service = (IAuthenticationService)theFactory.getService(IAuthenticationService_field.NAME);
            System.Diagnostics.Debug.WriteLine("*****MANAGER - service: " + IAuthenticationService_field.NAME);
            if(service.authenticateUser(login))
            {
                IDatabaseService dbData = null;
                //get database service
                dbData = (IDatabaseService)theFactory.getService(IDatabaseService_field.NAME);
                //get all trails legal for use with this login
                return(dbData.getTrails(login));
            }
            else
            {
                return null;
            }
        }
        catch (Exception e)
        {
            System.Diagnostics.Debug.WriteLine("confirmLogin exception: " + e.Message);
            return null;
        }
    }
}
```

Figure 27 - Sample C# Manager class

Instead of performing all of the code operations within the Struts Action in Java or the `Server_Click` method in C# to determine the user information based upon the information retrieved from the login form, the example in Figure 26 earlier demonstrates the use of the `confirmLogin()` method seen in Figure 27 above. Rather than implementing specific Business logic code directly in the Integration layer in multiple locations, the responsibility now falls upon the Manager class to be the entry point to that Business logic. This further delineates the tiers and emphasizes the requirements defined earlier for robust architecture.

Once inside of the Business Logic, additional efficiency based implementations can be designed to further provide a robust architecture. Actual service code could be implemented into each of the *Manager* class' methods such as database connectivity or web service initialization, but again this type of approach would result in duplicated code and require multiple code modifications if that database connectivity were to change from a database such as MySQL to SQL Server, or from a database connection to perhaps a web service. This is where robust architecture can become an almost living, breathing, thinking system. In earlier analysis and examples in this project, the object oriented architecture and the power of encapsulating data within those objects played a significant role. In the collection based classes earlier, the type of collection container was encapsulated, but the class itself was always a specific type. When it comes to answering the second question posed above regarding avoiding the compilation of code at all in the event of a change, a type of robust architecture must be applied that allows for the code to essentially adapt "on the fly."

Using an Abstract Factory pattern methodology the Manager class such as the C# example shown in Figure 27 above utilizes a factory to generate the necessary objects when they're needed, dynamically. This pattern also de-couples the database form

the implementation which is “is relevant from the perspective of portability of the application to other kinds of databases. For example, it is conceivable the application may be required to work with relational databases from different vendors.” (Selvaraj, 1997, p. 14) Instead of hard-coding in the type of service required directly into the Manager and thus requiring a recompilation of code after any of type of change to the service, the Factory (see a Java example in Figure 28 below) provides an additional level of abstraction by creating the correct type of service necessary based upon a simple description.

```
public class Factory {
    private Factory() {};
    private static Factory factory = new Factory();
    public static Factory newInstance() {return factory;}

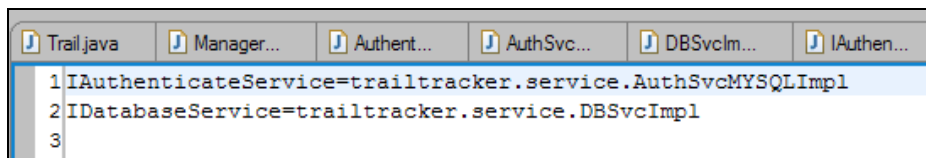
    private String getImplName(String serviceName) throws Exception {
        try {
            java.util.Properties props = new java.util.Properties();
            java.io.FileInputStream fis = new java.io.FileInputStream("C:/JavaClass/JavaEE1/trailtracker/ImplMap.txt");
            System.out.println("*****getImplName - " + fis.toString());
            props.load(fis);
            fis.close();
            return props.getProperty(serviceName);
        } catch (Exception e) {
            throw e;
        }
    }

    public Object getService(String serviceName) throws FactoryServiceException {
        Class classObj;
        try {
            System.out.println("*****Factory getService - serviceName: " + serviceName);
            classObj = Class.forName(getImplName(serviceName));
            return classObj.newInstance();
        } catch (Exception e) {
            throw new FactoryServiceException("Unable To Load Implementation");
        }
    }
}
```

Figure 28 - Factory.java code snippet

The implementation of the *Factory* class provides a dynamic, morphing class that can generate the service necessary for the task required. As seen in the Figure above, the Java coded class is actually quite simple, but further analysis provides insight into the power of this approach. The class itself provides only two methods, each with only a few lines of code. Beginning with the only public method, *getService()*, the method requires only a single string object containing the service required as a parameter. In a less robust manner, a switch case type statement could be added in

this method and depending upon the service name string passed in that type of object could be returned. Unfortunately, this would not only require additional coding to the Factory class for each new service that is added, but also create a higher coupling between the class that is calling the Factory class and the Factory itself. Instead, by using the *getImplName()* method and passing in the service name a mapping of that service name to a namespace and class is returned. Examining the *implMap.txt* file in Figure 29 below, the value-equal pairs represent the mappings of the service name to the namespace and class for that service.



```
1 IAuthenticateService=trailtracker.service.AuthSvcMySQLImpl
2 IDatabaseService=trailtracker.service.DBServiceImpl
3
```

Figure 29 - Implementing the Factory with ImplMap.txt

Using the namespace description returned from the *getImpl()* method, the Factory uses the Java *Class* object method *forName()* and creates an instance of the type of object represented by that namespace. Using an example from Figure 29, if the service name “IDatabaseService” was passed into the method, then “trailtracker.service.DBServiceImpl” would be returned and an instance of a *DBServiceImpl* object would be created and returned.

Although this type of implementation does accomplish one of the goals defined earlier by not requiring the class to be recompiled or new code added when a new service is created there is still potential for higher coupling than necessary if further steps aren’t taken. A new service can easily be added by simply adding a new value-equal pair to the text file, but further examination provides even deeper robust development potential. By examining both the type of class returned from the Factory and the type of class expected by the calling class (in this case the *Manager*

class), additional robust behavior becomes notable. Continuing with the example above, the *DBSvcImpl* class (see in Figure 30 below) implements the *IDatabaseService* interface (see in Figure 31 below). The *DBSvcImpl* class is an excellent example of robust architecture in itself apart from its involvement with the Factory. Through examination earlier in the design process the *IDatabaseService* interface creates the “skeleton” by which all database services must abide, but without actual regard to the exact database that will be implemented underneath. This allows each database service class implemented off of the interface to individually determine the type of database without any external knowledge of the exact database implementation outside of the service class.

```
1 public class DBSvcImpl implements IDatabaseService {
2
3     public Object getTrails(LoginBean login) throws Exception
4     {
5         TrailCollection trails = new TrailCollection();
6         MySQLConnection db = new MySQLConnection();
7
8         try {
9             db.initializeConnection();
10            ResultSet result;
11            result = db.QueryDBase("SELECT * FROM trail where TRAIL_CreatedBy
12                                   login.getID() + """);
13
14            while (result.next())
15            {
16
17                Trail t = new Trail();
18                GenericDBInfo g = new GenericDBInfo();
19                int temp;
20                g.setID(result.getInt("TRAIL_ID"));
21                g.setDescription(result.getString("TRAIL_Description"));
22                temp = result.getInt("TRAIL_IsMetric");
23                g.setIsMetric(temp);
24                g.setName(result.getString("TRAIL_Name"));
25                g.setOwner(result.getInt("TRAIL_CreatedBy_PERS_ID"));
26                t.setDBInfo(g);
27                t.setTotalDistance(result.getInt("TRAIL_Distance"));
28                trails.Add(t);
29            }
30            db.CloseDBase();
31            db = null;
32            return(trails);
33        }
34    }
35 }
```

Figure 30 - Database Service using Interface in Java

Using the interface implementation enables the different levels of the Service tier to remain low coupled while creating adaptable and flexible code designed with changes in mind. The *DBSvcImpl* class encapsulates the database implementation behind the system allowing for any changes to the database or the type of database used to happen without affecting the rest of the system.

```
11 package trailtracker.service;
12
13 import trailtracker.valueObjects.LoginBean;
14
15 public interface IDatabaseService extends IBaseService{
16     public static final String NAME = "IDatabaseService";
17     public Object getTrails(LoginBean login) throws Exception;
18     public void setTrail(Object o) throws Exception;
19     public boolean createNewUser(LoginBean login) throws Exception;
20 }
```

Figure 31 - IDatabaseService interface in Java

By analyzing the “front-end” of the class calling the actual Factory the full power of the Factory pattern and abstraction further demonstrates the robust architecture design. Through the use of abstraction and the interface in Figure 31 above, the code in the Manager class below exhibits a high level of flexibility, adaptability, as well as scalability.

When the Business Logic tier *Manager* class utilizes the *Factory* object, the initial reaction might have been to utilize the actual *DBSvcImpl* class because at the time the developer might have known that that specific was the implementation necessary to access the database. By taking a closer look at the value-equal pairs in the text file in Figure 29 it would be readily apparent that the *IDatabaseService* service name would generate a *DBSvcImpl* object. Unfortunately, this type of “here and now” thinking would couple the *Manager* code tighter with the *Factory* itself and force less efficient code development on more classes when a change becomes necessary.

```

public class Manager {
    public Object confirmLogin(LoginBean login)
    {
        Factory theFactory = Factory.newInstance();
        IAuthenticateService service = null;
        try
        {
            System.out.println("*****MANAGER");
            //use authentication service to check login
            service = (IAuthenticateService)theFactory.getService(IAuthenticateService.NAME);
            System.out.println("*****MANAGER - service: " + IAuthenticateService.NAME);
            if(service.authenticateUser(login))
            {
                IDatabaseService dbData = null;
                //get database service
                dbData = (IDatabaseService)theFactory.getService(IDatabaseService.NAME);
                //get all trails legal for use with this login
                return(dbData.getTrails(login));
            }else{
                return null;
            }
        }
    }
}

```

Figure 32 - Java implementation of Manager class

Instead, by casting the returned object from the *Factory* as the interface class (*IDatabaseService*), the abstraction provides a very robust implementation. Rather than higher coupling with the *Factory* and type of database service, casting the returned class as the interface maintains a very low coupling because the actual object type is ultimately be unknown; any of the method calls made to the *IDatabaseService* object in turn call the inherited class implementation that is actually returned by the *Factory*.

```

<configuration>
  <!-- Added for TrailTracker functionality -->
  <appSettings>
    <add key="IAuthenticateService" value="TrailTracker.service.AuthSvcMYSQ" />
    <add key="IDatabaseService" value="TrailTracker.service.DBSvcImpl" />
  </appSettings>
  <system.web>
    <!-- Add your application configuration -->

```

Figure 33 - Web.config file in C# / ASP.NET implementation of Factory

The exact same type of *Factory* pattern implementation could be generated with C# .NET as well. The code snippet in Figure 34 below demonstrates the same *Factory* class written in C#. The only real difference is the object method within the

getService() method, *System.Activator.CreateInstance()*, that generates the returned object. Additionally, instead of accessing a text file, the C# implementation utilizes the Configuration Settings XML file as seen in Figure 33 above, which returns a value based upon a key.

```
private static Factory factory = new Factory();
public static Factory newInstance() {return factory;}

private string getImplName(String serviceName)
{
    try
    {
        string serviceValue;
        serviceValue = System.Configuration.ConfigurationSettings.AppSettings(serviceName);
        System.Diagnostics.Debug.WriteLine("*****getImplName - " + serviceValue);
        return serviceValue;
    }
    catch (Exception e)
    {
        System.Diagnostics.Debug.WriteLine("*****getImplName Error- " + e.Message);
        throw e;
    }
}

public object getService(string serviceName)
{
    object classObj;
    try
    {
        System.Diagnostics.Debug.WriteLine("*****Factory getService - serviceName: " + serviceName);
        classObj = System.Activator.CreateInstance(null, getImplName(serviceName));
        return classObj;
    }
    catch (Exception e)
    {

```

Figure 34 – Factory in C#

Regardless of the language, the suggestions for robust architecture demonstrated above provide potential for a low-coupled code base, which lends itself to many of the attributes of robustness. Using encapsulation and abstraction, the code becomes adaptable, extensible, and flexible while affecting only minor portions of the system instead of requiring mass change. The use of the Factory pattern and dynamic factory implementation allows for scalability by enabling new services to be added, changed, or deleted without any code changes or recompilation of code. It does, also, add to the overall complexity of the code base which could serve as a potential hang up when modifications are required in the future. The fact that the operation itself relies on

external files and the higher level of abstraction make the code less intuitive at first glance. As with all solutions, balancing the added complexity versus the benefit determines the efficiency of that solution. Ultimately, the factory pattern solution's "organization of code into such loosely coupled sub-systems provide[s] great flexibility when it comes to maintenance and evolution of software." (Selvaraj, 1997, p. 16)

Conclusions

Change begets change. Nothing propagates so fast.

-- Charles Dickens 1812-1870, English novelist

In the very beginning stages of this project the database was constructed and the organization of that data was determined. The data and organization was examined and analyzed to create the most efficient setup of tables and relationships. This process of "evaluating and correcting table structures to minimize data redundancies [and] therefore helping to eliminate data anomalies" is referred to as *normalization*. (Rob & Coronel, 2004, p. 184) Although the process of normalizing the small database for this project didn't prove difficult primarily because of its scale, it did provide an interesting conclusion when demonstrating robust architecture in the prototype web application.

The ultimate goal of normalizing a database is to reduce data redundancies and data anomalies. Data that is duplicated in many different locations in a database can become the bane of the system if, upon changing that data, some of the information is modified while other information is left untouched. Immediately, the data becomes

invalid because the supposedly consistent data is now different where it should have been the same in each location. Progressing through the different normal form stages of normalization incrementally breaks down the data redundancy and potential data anomalies until the conditions no longer exist.

As with the database and normalizing the data within that database, the bane of coding is the inevitable change. Coding within a vacuum without any regard to the future, changes, or planning can be a simple task. Create the code. Test it against the current situation. Leave it. Forever. But is that truly realistic? Changes are bound to happen to any type of system for any number of reasons. The fact that changes will happen isn't ultimately the problem. The fact that the system hasn't been designed to handle those changes elegantly nor efficiently is.

In the world of database normalization, "the higher the normal form, the more joins are required to produce a specified output and the more slowly the database system responds to end-user demands." (Rob & Coronel, 2004, P. 184) The system's efficiency might actually suffer because of higher level of normalization. But, ultimately it does come down to efficiency, just as it does in the world of robust software architecture. As the system is being designed, special attention must be paid to the relationships between of the system's entities just as the relationships between data is analyzed through normalization.

This project did not try to highlight every possible solution nor every technology available that might aid in creating a robust architecture. The goal was to analyze robust solutions for the prototype in an effort to define a philosophy that could be applied to other development efforts. In the n-tier architecture, identifying the tiers and their responsibilities and relationships is key point in the development of a robust system. The Adding Efficiency to the Presentation Layer section above provides an

example of separating the backend system code of the Service and Integration tiers from the Presentation Layer itself. Although it might seem easier and quicker to “keep everything in one place” by putting the code directly into the HTML, this type of approach can just as quickly become very difficult to manage. The code and service methods, as well as the Business Logic of the system itself becomes very highly coupled to the Presentation and therefore the responsibilities of that layer become intermixed with the other tiers. Any type of change to the system could potentially result in a large scale effort touching multiple files and locations on multiple tiers. Clearly identifying the tiers and their responsibilities creates a more efficient environment that reduces the duplication of efforts. Keeping the Presentation Layer as lowly couple to the Business Logic as possible allows for either to be changed without affecting the other. Although it may be more difficult to plan and build out a system in this manner in the early stages, the efficiencies gained in the later stages when the changes happen is well worth the efforts early on.

There is a caveat to the robust approach. Although normalizing a database to its highest form might be the ideal state for eliminating data anomalies and redundancy, an appropriate balance between data organization and the level of normalization must be examined to keep the most efficient system possible. If the database itself becomes inefficient then the efforts of normalization have failed to achieve the desired results. Similarly, robust architecture and development must “walk the fine line” between creating a code base that is too complex and difficult to not only understand but maintain versus utilizing the concepts described in this paper to improve efficiency. One advantage of having code all in a single location and highly coupled is that everything is laid out right in front of the developer. Granted, it might be a large block of code and difficult to interpret, but at least it’s all there. Nothing is

hidden. All of the techniques described in this project encapsulate, hide, and separate out these blocks into more manageable and efficient chunks, but in the process it can be more cumbersome to quickly identify what is actually happening in the code because it jumps around. Pushing any of the robust techniques without truly realizing any benefit, or worse, creating more complexity and less efficiency is not robust development or architecture. Just as database normalization can go too far and ultimately create a less efficient system, so too can robust design and architecture just for the sake of robust design without any consideration for complexity and usability. Each of the different steps in the Tiles example added a new layer of complexity, all while creating a more “robust” system in the Presentation Layer, but the code to implement the Tiles technology becomes spread out amongst several different files and buried in separate XML files. Someone unfamiliar with the technology may have a difficult time interpreting the solution and be inefficient in making changes. The same situation goes for the Factory pattern solution. As the abstraction of the code solution grows so does the difficulty in understanding precisely what that component is actually doing. Again, forcing developers to step through code line by line and trace through the component in order to even begin to understand what might be happening does not imply a robust environment. Rather, if the robust code solution becomes too complex for its own good then perhaps taking a step back and applying a slightly higher coupling in order to maintain understandability would be the better option.

Any type of repeated code is a candidate for applying a robust methodology. Object oriented development and encapsulation of code within those objects is very popular for many reasons, but the ability to maintain a single unit of code and not have to repeat that same code throughout the system is clearly at the top of the list.

As seen in the examples above, the database code is encapsulated behind many layers of objects within the Service tier. Instead of implementing the database connectivity directly into the Integration and Presentation tiers, all of that knowledge and “power” is built into the *Manager* class in the Service Tier. The Business Logic for the system is controlled in this single location. Within that Business Logic, the Manager object manages the rules of the system, but again, even the Manager object doesn’t implement the exact database connectivity. It makes a call to the Factory object, which does its “black box” magic and returns a database connection which allows that Manager to now get the necessary data. The type of database or service that is accessing that data is clearly hidden. The Business logic within the Manager and Service tier is also clearly hidden from the Presentation and Integration tiers. This type of robust separation allows for very efficient change. By maintaining a low-coupled environment within the system, any type of change or modification to any one part of the system has very little, if no, effect on other components in the system. Making the change once, instead of multiple times, significantly reduces the chance for new errors and bugs to be introduced into the system and creates a very efficient and simple effort in making that change.

Using the robust concepts introduced throughout the project demonstrates an adaptable and extensible system. Change such as the type of database backend required or the type of collection objects used to hold the data or expanding the services within the system all could be handled without a dramatic effect on the system itself. Changing from a MySQL implementation for the database to a larger scale Oracle or SQL Server system would be hidden from the majority of the application. A new Oracle database service class could be implemented using the interface provided and then a simple change to the text or XML file (depending on

the implementation above in C# or Java) and the application would be using the new Oracle database rather than the MySQL backend. The effect on the other tiers and layers would be undetectable because nothing from their perspective would change. The backend operation of the database, type and/or functionality, is completely hidden via the encapsulation into the use of the factory and the interface.

Since the changes to the system in the future are nearly impossible to foretell, the flexibility of that given system is very important. If a change requires major modification to nearly every aspect of the system then the flexibility of that system must be questioned. Again, focusing on the Factory pattern example above, a wide variety of flexibility is provided through the usage of maintaining the actual class implementation names outside of the code (in text or XML) and through the usage of the interface. Being flexible is being able to change quickly and efficiently. If that change requires a different service, then that class can be generated, added to the system, and then the text or XML file modified without affecting any other code in the system. If the database system needs to be updated or changed, only a small piece of the code must be changed, but the bulk of the system will continue operation without any modification. By breaking the system into distinct segments within the tiers and layers to maintain low-coupling, the architecture provides the development team with the ability to make the changes necessary in a very quick and efficient manner.

In terms of scalability, the system's ability to add functionality such as new service, a new backend database, a different type of collection for better memory management, or a new page into the Presentation layer without major sweeping changes to the code base is a key important in a robust architecture. Systems will grow and change. As was presented at the very introduction of this project, change is

inevitable. If so, then preparation is the key. This project and the prototype designed were not developed nor presented as the “end all, be all” of robust software design and architecture, but rather presented in a manner to open the door to more efficient development in the future. This is not a general solution to all poor software design, merely the beginning of the philosophy and understanding required when attempting to develop a robust and efficient system and architecture. One key point to the philosophy is that robust architecture is essential to that preparation. Preparing and spending the extra effort during the initial design will pay off in the future when that change happens. If change does in fact “beget change,” and propagates quickly, then the extra preparation through robust architecture development will be well worth the effort. The key to the robust architecture is being able to handle the changes and growth efficiently.

Appendix – References

- Advice on Management. Retrieved May 29, 2006 from the World Wide Web: http://www.adviceonmanagement.com/advice_change.html
- Ambler, Scott W. (2006, March) Agile Modeling and eXtreme Programming (XP). Retrieved May 13, 2006 from the World Wide Web: <http://www.agilemodeling.com/essays/agileModelingXP.htm>
- Ambler, Scott W. (2006, April) UML 2 Class Diagrams. Retrieved May 13, 2006 from the World Wide Web: <http://www.agilemodeling.com/artifacts/classDiagram.htm>
- Anuganti, Venu, & Reggie Burnett. Exploring MySQL in the Microsoft .NET Environment. MySQL:Developer Zone. Retrieved May 13, 2006 from the World Wide Web: <http://dev.mysql.com/tech-resources/articles/dotnet/>
- Bahar, Abdul (Rajib). (2004, January) MySQL Schema in C#. The Code Project. January 19, 2004. Retrieved May 13, 2006 from the World Wide Web: <http://www.codeproject.com/cs/database/AbdMySqlSchema.asp>
- Booth, Jim. Building Middle Tier Objects in Visual Foxpro. Retrieved May 13, 2006 from the World Wide Web: <http://www.jamesbooth.com/n-tier.htm>
- Bradley, Ronan. (2003, October) Not so simple after all. Loosely Coupled Monthly Digest. Retrieved May 13, 2006 from the World Wide Web: <http://www.looselycoupled.com/opinion/2003/bradl-esb-infr1013.html>
- cardinals33 (alias) Using MySQL with .NET – Introduction. Developer Fusion. Retrieved May 13, 2006 from the World Wide Web: <http://www.developerfusion.co.uk/show/4635/>
- Chartier, Robert. (2006) Application Architecture: An N-Tier Approach - Part 1. 15 Seconds. Retrieved May 13, 2006 from the World Wide Web: <http://www.15seconds.com/issue/011023.htm>
- CreateObject Equivalent. .NET 247. Retrieved May 13, 2006 from the World Wide Web: <http://www.dotnet247.com/247reference/messages/33/166584.aspx>
- Davis, Malcom. (2001, February) Struts, an open-source MVC implementation: Manage complexity in large Web sites with this servlets and JSP framework. Retrieved May 13, 2006 from the World Wide Web: <http://www-128.ibm.com/developerworks/library/j-struts/?dwzone=java>
- Defining Adaptability. Merriam-Webster Online. Retrieved May 13, 2006 from the World Wide Web: <http://www.m-w.com/dictionary/adapted>

Defining Availability. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Availability>

Defining Business Logic. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/Business_logic

Defining Computer Software. Wikipedia. Retrieved May 29, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Software>.

Defining Coupling (Computer Science). Wikipedia. Retrieved May 20, 2006 from the World Wide Web: [http://en.wikipedia.org/wiki/Coupling_\(computer_science\)](http://en.wikipedia.org/wiki/Coupling_(computer_science))

Defining Dynamic. Wiktionary. Retrieved May 29, 2006 from the World Wide Web: <http://en.wiktionary.org/wiki/dynamic>.

Defining Extensibility. SearchWebServices.com. Retrieved May 13, 2006 from the World Wide Web: http://searchwebservicestarget.com/sDefinition/0,,sid26_gci283975,00.html

Defining Extensibility. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Extensibility>

Define Extreme Programming. Retrieved May 13, 2006 from the World Wide Web: http://www.webopedia.com/TERM/E/Extreme_Programming.html

Defining Flexibility. Merriam-Webster Online. Retrieved May 13, 2006 from the World Wide Web: <http://www.m-w.com/dictionary/flexibility>

Defining Middleware. Wikipedia. Retrieved June 18, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Middleware>.

Defining Model View Controller Pattern. Wikipedia. Retrieved May 29, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Model-view-controller>.

Defining Multi-tier Architecture. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/Multitier_architecture

Defining Reusability. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Reusability>

Defining Robustness. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Robustness>

Defining Robustness. Merriam-Webster Online. Retrieved May 13, 2006 from the World Wide Web: <http://www.m-w.com/dictionary/robustness>

Defining Scalability. Merriam-Webster Online. Retrieved May 13, 2006 from the World Wide Web: <http://www.merriamwebster.com/cgi-bin/dictionary?va=scalable>

- Defining Scalability. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Scalability>
- Defining Software Architecture. Wikipedia. Retrieved May 29, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/Software_Architecture.
- Defining Software Development Life Cycle (SDLC). Wikipedia. Retrieved May 13, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/Software_development_life_cycle
- Defining Three Tier Computing. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/Three-tier_%28computing%29
- Defining Understandability. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: <http://en.wikipedia.org/wiki/Understandability>
- Defining Web Application. Wikipedia. Retrieved May 13, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/Web_application
- Deitel, H.M., & P.J. Deitel. (2003) Java: How to Program. 5th Edition. Upper Saddle River: Prentice Hall.
- Eclipse Plugin Central. Retrieved May 2, 2006 from the World Wide Web: <http://eclipseplugincentral.com/PNphpBB2+file-viewtopic-t-2011.html>
- Farley, Jim. (2001, August) Microsoft .NET vs. J2EE: How Do They Stack Up? Retrieved November, 2005 from the World Wide Web: http://java.oreilly.com/news/farley_0800.html
- Fayad , Mohamed, & Marshall P. Cline. (1996, October) Aspects of Software Adaptability. University of Nevada, and Paradigm Shift, Inc. Vol. 39, No. 10 COMMUNICATIONS OF THE ACM.
- Fox, Daniel L. (2003, December) Using common domain logic patterns in your .NET applications. Builder.com. Retrieved May 13, 2006 from the World Wide Web: http://builderau.com.au/architect/sdi/soa/Using_common_domain_logic_patterns_in_your_NET_applications/0,39024602,20281590,00.htm
- Fowler, Martin. (2005, December) “The New Methodology.” Retrieved May 13, 2006 from the World Wide Web: <http://www.martinfowler.com/articles/newMethodology.html>
- Hinchcliffe, Dion. Is ASP.NET not measuring up to JSP’s MVC 2 architecture? Retrieved May 13, 2006 from the World Wide Web: <http://hinchcliffe.org/archive/2005/01/23/158.aspx>

- Holderfield, Vance T. & Michael N. Huhns. A Foundational Analysis of Software Robustness Using Redundant Agent Collaboration. Department of Computer Science and Engineering, University of South Carolina. Columbia, SC 29208 USA. Retrieved May 13, 2006 from the World Wide Web: <http://www.old.netobjectdays.org/pdf/02/papers/ws-ages/0906.pdf>
- Huhns, Michael N., & Vance T. Holderfield. (2002) Robust Software. Retrieved May 29, 2006 from the World Wide Web: <http://www.cse.sc.edu/research/cit/publications/papers/V6N2.pdf>
- Huhns, Michael N., & Vance T Holderfield, & Rosa Laura Zavala Gutierrez. Roust Software Via Agent-Based Redundancy. Retrieved May 29, 2006 from the World Wide Web: <http://www.cs.cmu.edu/~garlan/17811/Readings/ichucednha.pdf>
- Homer, Alex, & Dave Sussman, & Rob Howard, & Brian Francis, & Karli Watson, & Richard Anderson. (2004) Professional ASP.NET 1.1. Indianapolis: Wiley.
- How to Use the ODBC .NET Managed Provider in Visual C# .NET and Connection Strings. (2004, July) Retrieved May 13, 2006 from the World Wide Web: <http://support.microsoft.com/kb/310988/en-us>
- IIS: Notes on Server-Side Includes (SSI) syntax. (2005) Microsoft Help and Support. Retrieved May 13, 2006 from the World Wide Web: <http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q203064>
- Integrating Layer. (2004, May) Retrieved May 13, 2006 from the World Wide Web: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/intpatt-ch03.asp>
- Java Blueprints: Model-View-Controller. Retrieved June 8, 2006 from the World Wide Web: <http://java.sun.com/blueprints/patterns/MVC-detailed.html>
- Java Connectivity: Java to C# Mapping. Retrieved May 13, 2006 from the World Wide Web: <http://j-integra.intrinsyc.com/support/espresso/doc/JavaConn/mapping.html>
- Junhua, Nancy. (2004, April) Developing Struts with Easy Struts for Eclipse. Retrieved May 13, 2006 from the World Wide Web: <http://www-128.ibm.com/developerworks/library/os-ecstruts/>
- Kay, Russel. (2003, May) QuickStudy: System Development Life Cycle. ComputerWorld. Retrieved May 13, 2006 from the World Wide Web: <http://www.computerworld.com/developmenttopics/development/story/0,10801,71151,00.html>
- Knoernschild, Kirk. (2003) Object-oriented design metrics ensure robust software. Retrieved May 29, 2006 from the World Wide Web: <http://builder.com.com/5100-6386-5035294.html>

- Leadership Now. Retrieved May 29, 2006 from the World Wide Web:
<http://www.leadershipnow.com/changequotes.html>
- Learn About Java Technology. Java.com. Retrieved May 13, 2006 from the World Wide Web: <http://java.com/en/about/>
- Learn IT: Software Development. SearchWinIT.com. Retrieved May 13, 2006 from the World Wide Web: http://searchwin2000.techtarget.com/sDefinition/0,,sid1_gci936454,00.html
- Lewis, John, & William Loftus. (2001) Java Software Solutions: Foundations of Program Design. 2nd Edition. Reading: Addison-Wesley.
- Malani, Prakash. (January 4, 2002). UI design with Tiles and Struts: Several solutions for organizing your HTML and JSP view components. JavaWorld. Retrieve June 5, 2006 from the World Wide Web: <http://www.javaworld.com/javaworld/jw-01-2002/jw-0104-tilestrut.html>
- Mannadiar, Sabith.V. (2004, February) Connecting to MySQL database from your .NET applications. The Code Project. Retrieved May 13, 2006 from the World Wide Web: <http://www.codeproject.com/cs/database/ConnectMySQL.asp>
- Martin, Jeff. (2003, June) Using CollectionBase and DictionaryBase. The Code Project. Retrieved May 13, 2006 from the World Wide Web: <http://www.codeproject.com/csharp/collection1.asp>
- Medvidovic, Nenad, & David S. Rosenblum, & Richard N. Taylor. (1999) A language and environment for architecture-based software development and evolution. Paper presented at International Conference on Software Engineering, Los Angeles, California. Retrieved June 7, 2006 from the ACM Digital Library.
- Mitchell, Scott. Accessing Common Code, Constants, and Functions in an ASP.NET Project. 4 Guys from Rolla.com. Retrieved May 13, 2006 from the World Wide Web: <http://aspnet.4guysfromrolla.com/articles/122403-1.aspx>
- Mortensen, M. Keith, & Rob McGovern, & Charles Liptaak. (2003, December) ASP.NET and Struts: Web Application Architectures. MSDN. Retrieved May 13, 2006 from the World Wide Web: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspp/html/ASPNet-ASPNet-J2EE-Struts.asp>
- Murphy, Thomas. (2002, September) Battle of the platforms: Java versus .Net. ZDNet. Retrieved May 13, 2006 from the World Wide Web: <http://techupdate.zdnet.com/techupdate/stories/main/0,14179,2881567-2,00.html>>
- MySQL Connector/ODBC 3.51 Downloads. Retrieved May 13, 2006 from the World Wide Web: <http://dev.mysql.com/downloads/connector/odbc/3.51.html>.
- MySQL Reference Manual. MySQL.com. Retrieved May 13, 2006 from the World Wide Web: <http://dev.mysql.com/doc/refman/5.0/en/odbc-net-op-c-sharp-cp.html>

Nielsen, Jakob. (2003, August) Usability 101: Introduction to Usability. Retrieved May 13, 2006 from the World Wide Web: <http://www.useit.com/alertbox/20030825.html>>

Note: Interfaces cannot contain fields in the .NET Framework. MSDN. Retrieved May 13, 2006 from the World Wide Web: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_jlca/html/vberr1045interfacescannotcontainfieldsinnet.asp

Olson, Daniel. Using XML Based Configuration File in Windows Form Applications. C# Corner. Retrieved May 13, 2006 from the World Wide Web: <http://www.c-sharpcorner.com/Code/2002/April/XMLConfigInWinForms.asp>

Particle (alias) (2003, June) Using MySQL with Visual Studio. Planet Source Code. Retrieved May 13, 2006 from the World Wide Web: <http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=1288&lngWId=10>

Quote World. Retrieved May 29, 2006 from the World Wide Web: <http://www.quoteworld.org/quotes/885>

Ramasamy, Senthil. Submitting Web Form data from one ASP.NET page to another. Stardeveloper.com. Retrieved May 13, 2006 from the World Wide Web: <http://www.stardeveloper.com/articles/display.html?article=2003061901&page=1>

Rob, Peter, & Carlos Coronel. (2004) Database Systems: Design, Implementation, & Management. 6th Edition. Boston: Thomson Learning.

Roman, Ed, & Scott W. Ambler, & Tyler Jewell. (2002) Mastering Enterprise JavaBeans. 2nd Edition. New York: Wiley Computer Publishing.

Robinson, Simon, & K. Scott Allen, & Ollie Cornes, & Jay Glynn, & Zach Greenvoss, & Burton Harvey, & Christain Nagel, & Morgan Skinner, & Karli Watson. (2003) Professional C#. 2nd Edition. Indianapolis: Wiley Publishing, Inc.

Runtime object creation using strings (classname) with dotnet. .NET 247. Retrieved May 13, 2006 from the World Wide Web: <http://www.dotnet247.com/247reference/msgs/1/5414.aspx>

Selvaraj, Asokan R., & Debasish Ghosh. (June 1997) Implementation of a database factory. ACM SIGPLAN Notices, 32, 14 – 18.

Shalloway, Alan, & James R. Trott. (2002) Design Patterns Explained: A New Perspective on Object Oriented Design. Boston: Addison-Wesley.

Sheil, Humphrey, & Michael Monteiro. (2002, June) Rumble in the jungle: J2EE versus .Net, Part 1. How do J2EE and Microsoft's .Net compare in enterprise

- environments? JavaWorld. Retrieved May 13, 2006 from the World Wide Web: <http://www.javaworld.com/javaworld/jw-06-2002/jw-0628-j2eevsnet.html>
- Shen, Derek Yang. (July 19, 2004). Put JSF to Work. JavaWorld. Retrieved June 18, 2006 from the World Wide Web: <http://www.javaworld.com/javaworld/jw-07-2004/jw-0719-jsf.html>
- Sjodin, Robert. (2005, September) The Roles of Architecture, Patterns & Frameworks in Agile System Development. Regis University.
- Stafford, Randy. Services Layer. Retrieved May 13, 2006 from the World Wide Web: <http://www.martinfowler.com/eaCatalog/serviceLayer.html>
- Stall, Tim. Understanding Interfaces and Their Usefulness. 4 Guys from Rolla.com. Retrieved May 13, 2006 from the World Wide Web: <http://aspnet.4guysfromrolla.com/articles/110304-1.aspx>
- Struts. Learntechnology.net. Retrieved May 13, 2006 from the World Wide Web: <http://www.learntechnology.net/struts-lesson-1.do>
- Subramanian, Nary, & Lawrence Chung. (2002) Software Architecture Adaptability: An NFR Approach. Applied Technology Division, Anritsu Company and Dept. of Computer Science, University of Texas, Dallas. Copyright ACM 2002 1-58
- The Linux Information Project. (2005) Robust Definition. Retrieved May 29, 2006 from the World Wide Web: <http://www.bellevuelinux.org/robust.html>
- ucblockhead (alias). (2002, June) Language Comparison, C#, C++ and Java (Technology). Retrieved May 13, 2006 from the World Wide Web: <http://www.kuro5hin.org/story/2002/6/25/122237/078>
- Ullman, Chris, & John Kauffman, & Chris Hart, & David Sussman, & Daniel Maharry. (2004) Beginning ASP.NET 1.1 with Visual C#.NET 2003. Indianapolis: Wiley.
- Usability in Website and Software Design. Usability First. Retrieved May 13, 2006 from the World Wide Web: <http://www.usabilityfirst.com/>
- Wainwright, Phil. (2005, October) Loosely Coupled weblog. Retrieved October, 2005 from the World Wide Web: <http://www.looselycoupled.com/blog/lc00aa00121.html>
- Watson, Karli, & David Espinosz, & Zach Greenvoss, & Jacob Hammer Perderson, & Christian Nagel, & Jon D. Reid, & Matthew Reynolds, & Morgan Skinner, & Eric White. (2003) Beginning Visual C#. Indianapolis: Wiley Publishing, Inc.
- Welcome to Apache Struts. The Apache Software Foundation. Retrieved May 13, 2006 from the World Wide Web: <http://struts.apache.org/>

What's the equivalent of this in C# (Java). .NET 247. Retrieved May 13, 2006 from the World Wide Web: <http://www.dotnet247.com/247reference/msgs/11/56932.aspx>

Whitten, Jeffrey L., & Lonnie D. Bentley, & Kevin C. Dittman. (2000) Systems Analysis and Design Methods. 5th Edition. McGraw Hill: Boston.