2011

Selection and Implementation of Technologies for the Re-Engineering of an Existing Software System

Stan William Naspinski
University of South Florida, stan@naspinski.net

Selection and Implementation of Technologies for the Re-Engineering of an Existing Software System

by

Stan Naspinski

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Jay Ligatti, Ph.D.
Dewey Rundus, Ph.D.
Yicheng Tu, Ph.D.

Date of Approval:
October 14, 2011

Keywords: REST, MVC, Legacy Application, Enporion, FLEX, .Net, Web Services

ACKNOWLEDGEMENTS

I would like to thank my Major Professor Jay Ligatti, for guiding me along and keeping me on track throughout the thesis process. I would also like to thank Professor Dewey Rundus for being a great help in preparing me for this opportunity as well as Professor Yicheng Tu for introducing me to the research assistant opportunity that vectored this thesis project.

I would also like to acknowledge the University of South Florida as a whole for allowing me the opportunity to work with the Enporion company for my case study. This project has truly made my graduate studies much more fruitful on my end as a student.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A major hurdle for any company to cross is the act of re-engineering software if they wish to stay relevant. With the speed that software and technology advances, it would be ignorant for any product to stagnate. With that comes the inherent difficulties of choosing which of the older technologies to keep (if any) and which newer technologies to employ in the re-engineered solution. Once that is covered, the actual implementation presents its own set of challenges to both the decision makers and developers in the process.

This thesis describes a case study, in particular the efforts put forth to re-engineer some specific software. While the software is quite capable, it is becoming more and more outdated every passing year, not to mention more difficult to maintain, upgrade and alter, providing a perfect example to explore.

The focus of this thesis is to discuss what avenues of upgrading and methods of providing comparable or improved services to the end user our team chose and implemented. These include using a relational database with an advanced object-relational mapper in a modern environment to provide a REpresentational State Transfer (REST) web service that will then supply a rich interactive front-end. Taken together, these tools are quite powerful and capable.

CHAPTER 1

INTRODUCTION

Software will inherently become out of date as the world of programming and applications moves faster than almost any other. It then becomes necessary practice for an organization to re-engineer their software on a regular basis, in order to keep their systems and even employees skills relevant. The decision to upgrade is one thing, but the difficulties stemming from this decision are the true challenges. Selection of which technologies to use can be challenging and divisive for a team of planners and engineers. After that comes the implementation, which can bring on challenges of its own, including environmental constraints, open-source software support and an endless list of others. All of this combines to make a quite complicated process.

Choosing which technologies to use is often more than picking the "best" technology available and applying it. It is clear to most in any sort of technology field that at almost no time in a project's life-cycle is there a single true best option that stands out; it is the art of choosing the best fit not only for your project, but the combination of that project with the abilities, experience and talents of the development group that will be implementing said project.

This thesis is based around a case study of Enporion, an E-Procurement company from Tampa, Florida. Enporion currently has a software product that it provides to its customers. This product was developed years ago and has shown the need to be upgraded to a higher level of both service maintainability and back-end technological capability. To do this, they decided to bring in a team of graduate students from the University of South Florida (USF): Matthew Spaulding, Nalin Saigal and myself. Our job was to migrate the old technology and database tables to newer, more powerful and easier to maintain technologies while

also incorporating new and more useful features to the suite. All of this was needed while maintaining 100% uptime for the existing customers with a smooth, gradual transition of the existing services to the newly developed one. We were to help evaluate what was provided in the past and how we could translate that into the most versatile and useful setup we could engineer; we needed to take a good product and attempt to make it greater through a deliberate process. In the end we needed to have produced testable, well-documented code that would be easily maintained and modular in nature. We were given a lot of freedom to use what technologies we had our best experiences with and were able to suggest alternative approaches to development with open discussions between the USF group, the CTO James Garcia, and the local technology manager Chris Stimac.

Not only did there have to be a new, fully-functional system implemented, but the existing users had to be migrated over without any data or service loss. This was to be a gradual migration over a large period of time as each application was independently produced, tested, and finally pushed into production. A large caveat to this was that we were to redesign the user, role and privilege system that already existed and to keep the two systems in sync during the prolonged cut-over. One challenge raised by this was how, during the migration, could application data and interface be switched over without a service interruption to users? Additionally, how do both the legacy and new users and privilege system in-sync during this entire process? Both of these require a good amount of planning and even more testing; this case study is not just a lab experiment, it is an in-use application supporting paying customers the whole way through.

Overall there are a number of challenges to overcome and features to improve upon. There is a lot of development to be done, both in replicating old processes and making new ones. The goal is to make existing processes as useful and well-coded as they already are with the intention of improving upon them, as well as adding new features and abilities where applicable, all in the newer and more advanced technologies we decide upon.

1.1 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 describes the problems that have been faced inside the old system and the shortcomings of the frameworks it is built upon; this provides a good foundation for the remaining text. Chapter 3 is focused on the new methods and frameworks we chose to replace and improve upon the existing architecture, as well as some of the problems we came across while implementing them. Chapter 4 describes the deployment and testing environment through which we automated the entire system. Finally, Chapter 5 discusses related work and alternatives to the choices that were made in this particular case study. Chapter 6 then concludes the thesis.

CHAPTER 2

EXISTING ARCHITECTURE

The existing system is more than capable and has been in use by over 2,500 users for over eight years. It is not that it is a broken system or that it is not fully usable, it is the fact that the CTO had realized that it was limited and the setup they were currently using was not allowing them to expand and improve on the system the way they would like to or the speed at which they would like to as well. The existing system was built upon a proprietary architecture named "SkyWay" - while SkyWay proved to be a capable and valuable system, it did have some limitations and lack of flexibility that clashed with the goals of the company.

2.1 SkyWay

This case study has used software that had developed over the years with a proprietary framework known as SkyWay. SkyWay is a Java/JBoss-based software-building system that abstracts away the programming out of developing what can be complex software. SkyWay is capable and intuitive software. As in modern object relational mappers (ORMs), it is able to extract structure, data-types and relations out of an existing database and produce classes of objects. These objects are then combined, through a very intricate graphical user interface (GUI) with different methods, views and goals to then produce a final product, which is a Java based web application. For example, a user could point SkyWay at an existing database, which would extract information about the tables, say a *users* and a *companies* table. SkyWay then produces Java objects that correspond to an entry in each table. The user could then drag the object onto a workspace. Now that the object is visible

on the workspace, the user could then drag an action such as editing and connect it to the object; following that, they would drag a hypertext markup language (HTML) view as to what is visible while editing. Inside this view, they can customize the HTML markup either via drag-and-drop modules or directly manipulating the markup.

Though this is just a small example, when all of this is said and done and the user chooses to deploy and publish, SkyWay produces `.ear`, `.jar` and `.jsp` files like any other java-based web interface. There is much more SkyWay is capable of, and Enporion had used it to its fullest abilities.

Enporion was able to do this partly because SkyWay had worked with Enporion since very close to the start of SkyWay itself, with the original version 1. Enporion had been in on the ground floor and had become a flagship customer, so they worked closely with SkyWay when developing new features all the way up until the present when they are version 5.1.

Even with the ability to ask for new features directly, problems began arising with the powerful, but ultimately limiting abilities of SkyWay. Though it is a very robust, clean and easy-to-use system, the limitations included the following:

2.1.1 Interoperability

With SkyWay being its own proprietary system, its interoperability was limited at best. As it was essentially the logical communicator for the data (between the database and the interface), it essentially cut off communication from other systems that were not using SkyWay unless they accessed the MSSQL server instance directly, bypassing the logic and rules, which is often a poor plan. Though modern ORMs do work in this same way, they are meant to be interfaced by outside actors. SkyWay did have the ability to interact with certain constructs of Java files, but this was not enough for the goals of the system.

2.1.2 Scalability

SkyWay was not built for a highly scalable environment; it was built for smaller relational databases and interaction where data could be easily visualized. But this simple model

quickly slows down among masses of data and complicated structure — it was never meant to handle software of the size Enporion's had grown. A view of the workspace, where just a single application was built, looked like an indecipherable web of relations, with each line in the web representing some sort of logic and each node representing an action, object, etc. The visualization was nice up close, but overall, it became extremely confusing. And with no way to modularize the applications into smaller divisions, this had become overwhelming. The applications had become too big to efficiently manage in SkyWay.

2.1.3 Modularization And Reuse

It is now a very common and encouraged practice in software development to encapsulate programs into distinct working units able to plug into one another. These *modules* make for the simple reuse of code throughout an application or even between applications. SkyWay did not provide a simple way of modularization, especially between applications. For example, if one application had a need to see a user's privileges, it is safe to say that another application will need that ability as well. It is simple enough to direct SkyWay to do such things, but to accomplish this required repeating what was already done, a sort of "copy-paste" approach to development. This works fine and can be quite fast, but the real problem comes in maintenance. When a change is made, it has to be made in many places; not to mention there is no method in place to make sure all the changes were made in all the right places; this pattern can be seen in Figure 2.1.

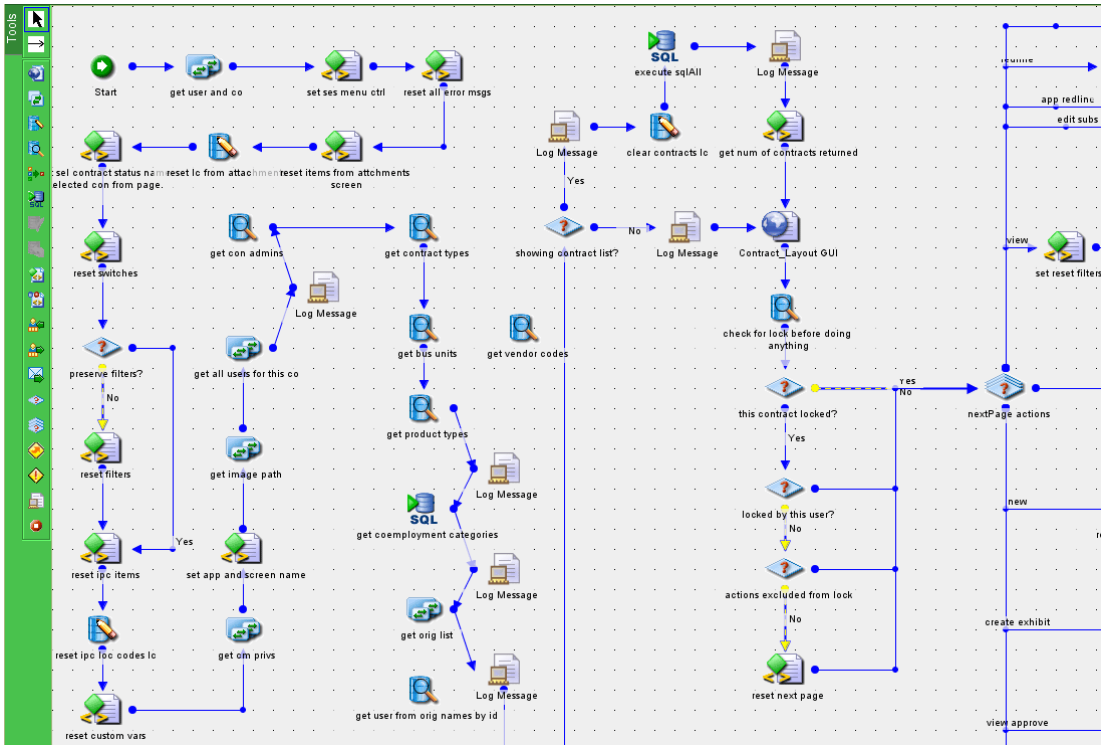

Figure 2.1. Example of SkyWay implementation GUI.

2.1.4 Testability

Not only can testability help locate where problems are occurring, it can pinpoint them before they become a problem for the end user or client. SkyWay itself has no means of automated unit testing. Because our project relies so heavily on automation, this presents an obstacle; this also means that early error detection would require a significant amount of engineering, or simply ignoring any testing solutions (which was the chosen path in the past). Manually testing is inefficient and incomplete, and engineering a solution is extremely time consuming.

2.1.5 Application State Management

The internet is a stateless medium, meaning it is up to programs to persist state information if it needs to be done. SkyWay was able to track states, but not to the desired level. Timeouts were often a problem, caused by a few different reasons, but all the same

result of the state being lost and the program failing to function as intended. One cause was the expiration of java beans within the system. If a user initiated an action, a bean was created, then too much time passed or the memory was needed, the bean expired leaving the user orphaned without a method to get back. Another problem that arose was that when the internal ORM got out of sync with the database, no database updates would be allowed to take place, essentially leaving the user unable to regain their session within the application. It is impossible to make an error-proof system, but error recovery is also very important and viable; with the SkyWay architecture, achieving this would require some in-depth engineering and a large investment in time.

2.1.6 Resources

Though a smaller issue than the ones previously mentioned, SkyWay required additional resources on the servers. Enporion's servers were already running Microsoft IIS as a web serving service, so SkyWay's need to run on JBoss, meant that the server had to run two separate processes to essentially accomplish the goal of serving up web content. While this might not seem a lot to run on powerful servers, under high stress, these extra cycles used could mean poor performance or downtime for users, which is often unacceptable.

2.2 Summary

Despite the shortcomings detailed in the sections above, SkyWay proved to be an incredibly robust tool that helped Enporion grow while creating and maintaining a huge base of customers. The SkyWay framework was capable, but as Enporion's needs changed, so did the viability of staying on this platform that they had become accustomed to.

The growth of Enporion and the changing needs of their customers meant that they had to evolve and provide an ever increasing level of service. The problems with state management were starting to cause the users inconvenience and time lost. Though the framework provided by SkyWay had served them well, any system that moves to abstract away actual programming means that you are going to lose a certain level of control and

customization that is available when your hands are in the coding dirt. With hands on the actual code the other problems such as testability and modularity could also be addressed and corrected, making maintenance and the overall final product a more robust piece of software.

CHAPTER 3

EXPLORING AND IMPLEMENTING NEW SOLUTIONS

There are nearly unlimited avenues of development when given the freedom to choose our own methods. The only limitations imposed on us were that we were to use Adobe Flex as the user interface and we were to be working on Windows 2003 Servers with future upgrades to Windows 2008 Servers. This opened up the possibilities for almost any architecture we could think of, whichever we decided would help us accomplish our goals the best and often the simplest way.

3.1 Operating System

As mentioned above, the operating system provided was that of Microsoft Windows Server 2003 R2 with the plans of moving to Server 2008 in the future. This is a less flexible part of the implementation as the existing servers were also used for other business processes and could not be simply replaced or removed without significant monetary and labor costs to the organization. The given server configurations meant that our server technology would be provided by Windows Server as well through the Internet Information Server (IIS) 6.0 while using the Server 2003 machines and later this would be IIS 7[.5] in the future with 2008. Altogether these are very flexible technologies that the group had a large collective experience with.

3.2 Data

Early in the project, I was designated as the lead for the data management due to my prior experience with database projects both academic, governmental and commercial.

The obvious choice for data is that of a database. There are other options as files and other relational database alternatives, but with the existing data already stored in a SQL database, plus the amount of data stored and the highly structured and relational nature of the data, it seemed logical to keep using some sort of database technology to build our solution. The combination of convenience, available software licences, and the advantages given by the relational structure were all factors in the decision to keep Microsoft SQL 2005 as our data storage solution.

3.2.1 Relational Database Management System

Transactional Structured Query Language (T-SQL), often referred to as simply SQL, is one of the most common methods of data storage and retrieval for computer programs of all sizes and origins. SQL has been around since 1970 and is familiar with most anyone involved in any sort of software development [1]. In order to use SQL, a relational database management system (RDBMS) must be employed, which is the actual implementation of the database itself. Alternatives were discussed, but the combination of the type of data along with the existing system weighed heavily on the decision use with SQL and an RDBMS for data storage.

The data stored in the system was up-to-date, but the structures and fields themselves were not necessarily still useful or relevant. We had discussed that the existing database held a large amount of sparse or even unused data, and those fields could be trimmed without losing any information. With this information, it was clear to see that data transfer, pruning, and organization would have to be addressed regardless of whether or not the platform was changed. Given that the existing data was stored in an instance of MSSQL 2005 it would be most convenient to keep the data in the same storage structure unless other methods provided what we needed along with other advantages to justify the additional labor to move the data across platforms.

The group's skills and nearly all of its past data-storage experience, was with SQL, though this experience was spread across multiple platforms including MSSQL, PostgreSQL,

MySQL and SQLite [2, 3, 4, 5]. These are all capable platforms in their own right, but the platform we were working on, and the future of possible large scale expansion, pointed us towards MSSQL. Moreover, the data was highly relational, and Enporion already owned the software licenses for MSSQL 2005. This platform also had the distinct advantage of being known for the ability to scale effectively and easily. Even when compared to the more expensive Oracle platform, it was found that, while all were capable, SQL Server 2005 was simpler to manage and more cost effective than Oracle, as well better at scaling than all others [6]. MSSQL is a proven and reliable database with which our team had confidence and experience, and as will be discussed in the following section, this provided an advantage with the chosen data-access layer.

3.3 Data Access Layer

The Data Access Layer (DAL) is simply the layer between the database itself and any other programs that wish to interact. This could be achieved with SQL directly, but doing so defeats the purpose of having reusable code, in addition to ignoring available time-saving resources. Moreover, if all interactions are not routed through a common and controlled actor, it may lead to security vulnerabilities such as SQL injection attacks. Reusability of more complex methods and structures is another large reason for a DAL; this way, if other programs and/or APIs want to take advantage of more complex code that has been written, it does not burden them with the development of such code [7].

3.3.1 Language Integrated Query

Language Integrated Query (Linq) is a query approach developed by Microsoft to make querying SQL in .Net coding more intuitive and resemble object-oriented programming. Objects in .Net, as well as most programming frameworks, such as arrays, lists, etc. are well understood. Often times programmers are also well versed in SQL, as they need to communicate with databases. Linq takes the idea that querying skills can be useful outside of databases. For example, imagine a program has an array of 10,000 strings, and

```
var sql = from s in strings
              where s.Length == 5 ||
                  (s.Length >= 2 && s.Substring(1,1).Equals("p")
              select s;
```

Figure 3.1. Linq using SQL-like syntax.

```
var lambda = strings.Where(s => s.Length == 5 ||
                 (s.Length >= 2 && s.Substring(1,1).Equals("p"));
```

Figure 3.2. Linq using lambda syntax.

a programmer would like to get all the objects that either have the second letter "p" or are five letters in length. Any capable programmer could write a loop to achieve this goal, but that is not necessarily an efficient or clean approach. Linq takes the idea that you can query these objects in an almost SQL-like syntax.

In Figure 3.1 see Linq's SQL like syntax for selecting variables from an array variable named "strings" that have a second letter "p" or length of five. While Figure 3.2 shows how Linq accomplishes the same but with its more terse lambda syntax.

Linq's ability to query in a manner that resembles SQL opens the door for new technologies that integrate with SQL but treat them as a component of a direct object-oriented programming system.

3.3.2 Entity Framework 4

Object-relational mappers (ORMs) are a framework that act as an interface between the data (generally a database) and the logical layer of a program; therefore they are a perfect start to a DAL in any application, as they are widely tested and very mature. ORMs often use code to generate the data-relational language output to deal directly with the data source (SQL in our case). ORM solutions have become extremely useful and common throughout the development world, with some common ones having been around for many years, such as Java's Hibernate and Ruby's ActiveRecord [8]. Though it is often

vital to know how SQL works behind the scenes, ORMs can provide huge advantages such as security, clean error detection, in-pipe validation, triggered events outside of the SQL environment, etc. This approach is much faster to develop, edit and test than more the traditional .Net ADO and SQL stored procedures — though essentially they all accomplish the same goal with similar (often the exact same) actual queries.

There are multiple ORMs available for our given platform combination of .Net and SQL. The most common are NHibernate, Linq-to-SQL and Entity Framework, with lesser known ORMs out there such as Dapper [9, 10, 11, 12]. We decided on the Entity Framework 4 (EF4) ORM due to many advantages over the others in our particular situation which will be covered later. It provided everything we needed as well as a vigorous plan forward as this has become a flagship product for Microsoft and receives a large amount of support and updates, whereas Linq-to-SQL support has been deprecated and NHibernate does not offer any enterprise level support if it were ever necessary [13, 10].

EF4 was designed specifically with Linq and MSSQL in mind (though it is also compatible with other database technologies). By representing the database, rows, tables and relations as objects, it allows programmers to deal with the database as an object oriented collection of classes. This allows developers to utilize the huge array of Object-Oriented Programming (OOP) methods and approaches that have been developed and cultivated over the years —this is familiar territory for a large swathe of programmers, as it was for our team. Now, instead of writing SQL directly, we are able to write strongly-typed Linq queries that are often more optimized that the SQL we had the knowledge to write on our own. Possibly more importantly, it allowed us to write queries at an amazing speed in comparison to the older ADO.Net or simple stored procedures (though there is always still the option for using both *with* EF4).

EF4 also provides a built-in defense against SQL injections, which is always important when working with SQL, as it is often the easiest form of attack. Any user with even a rudimentary understanding of SQL can cause significant damage to a system if it is left vulnerable to this sort of attack. The main attraction is the ability to use Linq syntax when

writing queries for a database, then being able to utilize .Net's IQueryable, interface which is extremely versatile and simple to use, especially to coders with past experience as we were supplied with [11]. This framework allows for very fast code development and testing, allowing testing in a uniform and clean environment, which will also be covered further in the thesis.

In the design of the code-base, we decided to separate the DAL into its own class library or as they are known in .Net dynamic link libraries or "dlls" (singular is "dll"). With this design choice, we will be able to include this particular dll, is the namespace `Enporion.DomainModel` which we had produce the dll `Enporion.DomainModel.dll` —for simplicity, the libraries were names in correspondence with the namespaces they represented as a whole, this is common practice and helps prevent confusion, though not required. With this library separation, not only can we use the library with different applications, the option also now exists to distribute the libraries as a sperate entity without disbursing but the actual source code (though code compilers do exist).

3.3.2.1 Methods To Extend EF4 Code

There were three approaches that were explored once Linq-to-Entities was chosen as our ORM. Though the EF4 code that was generated was very capable, there were additional methods, validation and functionality that we wanted to be added. The fact that the code was generated automatically (upon request) after a database change, different methods of extending the functionality were explored.

The first was the most direct; it involved actually modifying the class files themselves, building them into the generated classes. This presented a problem as the EF4 code was generated on-the-fly after the database itself had been structurally changed, therefore over-writing any changes that would have been made; this meant that if this method was chosen some sort of solution such as a build-time script would have to be figured out to continue, which also meant that the advantages over the following methods must be great enough

to warrant the extra labor. This is the simplest method and not a jump from traditional programming in any way.

The next method was to use public partial classes, essentially extending the capabilities and adding methods and properties to the generated classes. The generated classes are built as partials, so they can be added on to with this method later, without having to do anything upon subsequent regenerations of the EF4 classes. This method also covered possible validation of object in the future if the ASP.NET MVC framework were ever to be used in its more native form through attribute validation [14]. This also allows addition of non-native methods and properties into the base class, which can be extremely convenient as well as relatively transparent to the end user of the access layer.

Third was the simplest, with public static extension methods for each generated class. By utilizing the ability to make static extensions which is present in the newest versions of the .NET framework, static methods can be produced to extend the functionality of an existing class. This is a lot like the partial classes, but as they are static can become a bit more cumbersome as the direct referencing of the static namespaces must be present in order for them to be compiled correctly [15]. Static classes also require that everything be a method and do not allow you to declare properties into the classes like the other two mentioned approaches.

All were explored and experiments run as we did not know how the compiler would deal with essentially the same task done three different ways. Each test ran an initialization of a class and a random addition to an integer property on the class itself; this test was run 100,000 times for each recorded time in order to see if there were any divergence, plus this was loads above any processing we would ever encounter. The results were quite interesting as all methods produced what was basically the same results, both in speed and in processor load as you can see in Table 3.1 and Figure 3.3. As you can see, there is almost no difference in the speed and load of the operations, moving the decision more to convenience than performance. The most promising conclusion that we came to is that the compiler dealt with the different methods in the exact same ways. We were curious

Table 3.1. Time in seconds while running the different methods for extending the generated EF4 classes.

seconds	**Direct Class Modification**	**Partial Classes**	**Static Extensions**
Time 1	3.920	3.539	3.532
Time 2	3.717	3.560	3.606
Time 3	3.639	3.590	3.692
Time 4	3.581	3.606	3.572
Time 5	3.3505	3.580	3.615
Time 6	3.662	3.879	3.640
Time 7	3.591	3.719	3.561
Time 8	3.500	3.592	3.628
Time 9	3.846	3.578	3.817
Time 10	3.564	3.533	3.835
Average	**3.6525**	**3.6176**	**3.6498**

if the compiler would have to take more memory to re-initialize each object in the direct and partial classes each time it was called, while the static extension would only be dealt with once, but it seems as if this was not a problem; most likely due to the advances in the .Net compiler. With the partial classes being more flexible than directly editing the classes, and the partial classes having easier maintenance and as a more accepted programming approach (avoiding statics unless necessary), the decision was made to stick with partial classes as our method of extending the generated DAL classes.

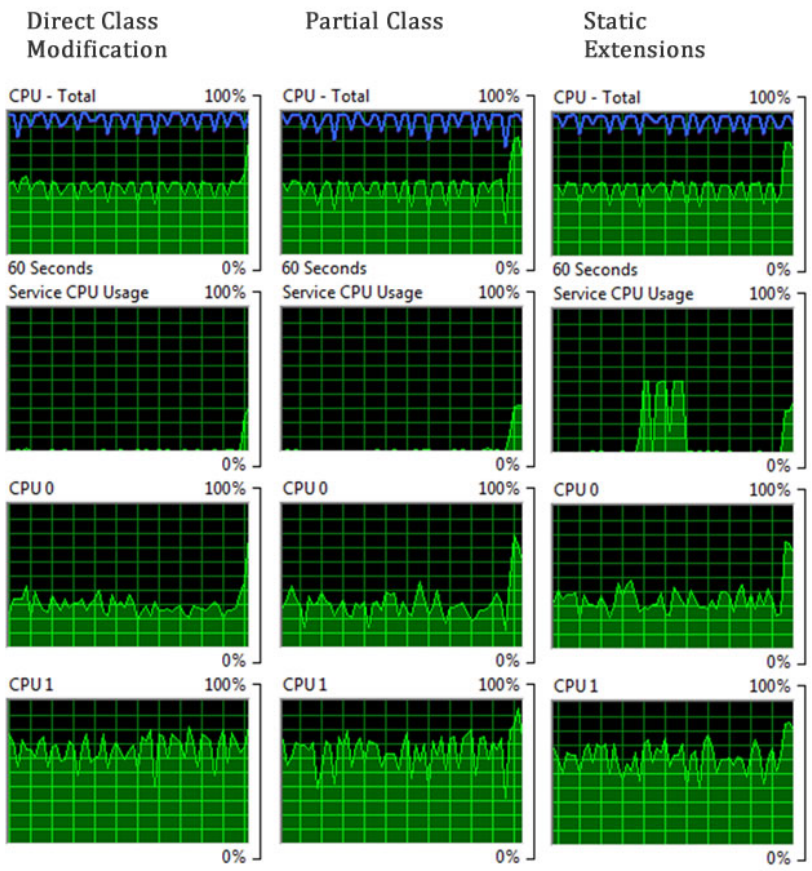

Figure 3.3. CPU load while running the different methods for extending the generated EF4 classes.

3.3.2.2 Additional Data Access

We ran into a somewhat difficult dilemma because of the legacy-support aspect of this case study. The new system we had developed had its own tables for things such as users and companies, which were present in the old system. This was to be a gradual conversion from the old system, so the option to simply copy the data and flip the switch was not there. Updating the legacy system when an update to the new system was not a problem given that full access to the old database itself was available. It was the other direction that produced the problems for the migration. How would the old system update the new system if a change was made that affected both? Considering that the access into the innards of the

legacy code was limited if available at all, it was not possible to go in and modify that code to update the newer database, so there had to be another solution. There was the option of using the database itself to track changes by using only triggers and stored procedures with straight T-SQL commands, but that de-couples our DAL from **Enporion.DomainModel** where it was currently all modularized and splits it into a completely separate part of the system; this breaks the model we were going for, so we delved deeper into the problem. What was found was that .Net Common Runtime Language (CLR) had the ability to run out of the MSSQL shell; therefore, .Net dlls could be executed from inside SQL itself. What this meant was that we would be able to keep the database manipulation code inside our DAL, but still be able to control execution from the database; meaning we would not need to modify the existing code base to get the sync abilities we were after. This turned out to be a limited and not straightforward application, but it was successful.

In order to implement this design, a dll would have to be compiled and transferred to a location available to the instance of MSSQL running the database. Next, trust relationships would have to be set up in order to allow the libraries to be accessed; this was done with either explicit opening of trust, or by digital signatures attached to the libraries, the latter being the one chosen as it was far more secure. Next the libraries were compiled into MSSQL native namespaces. Following that, new stored procedures were constructed that took in regular SQL variables and passed them to the now native functions available. Finally, new SQL triggers were set up to recognize UPDATE statements into the related tables of the database that needed to sync. Now every time a certain condition (updating the *users* table in our example) was met, SQL would automatically trigger the stored procedure with the related parameters, now updating the new system table with information supplied to the legacy table. Now this brings up an entirely new problem on top of being a little convoluted itself. If you realize that earlier it was explained that when the new system updated a table that needed to be synced, it would automatically update the legacy table. Now that the legacy table was armed with a trigger, this means that it will automatically update the newer table with its information; this will always be overwriting data with the same exact

data, but it will be triggering a SQL transaction regardless, which is an unnecessary action. It was deemed that this action, though possibly detrimental to performance was tolerable to keep the databases synced without risking breaking the existing codebase. Though there was a transaction happening, this was strictly communication with MSSQL itself, so the latency would be insignificant. In addition, this was only on what was coined as "low-activity" sections of the program as it was dealing with users and company profiles, which were accessed much less than the actual records that the applications focused on manipulating. Finally, these triggers are only temporary until all applications are migrated to the new system.

The implementation of the CLR within MSSQL is consistently behind that of the standard .Net Framework, meaning at the time of this project it was only able to run a limited set of pre-.Net 3.5 core libraries. One of the major setbacks was the fact that neither of the native ORMs (Linq-to-SQL and Entity Framework) were available to run. This was not a large hurdle as there was plenty of experience in the older .Net ADO, though it did result in mixing of technologies. The fact that this code is written in a very capable, extendable and robust language means that at any time, supplementary abilities could be added (i.e. logging, reports, etc.) without affecting the SQL transactions themselves. Also, considering the code is being managed as a distinct library, if there was ever the need to upgrade or change, it would be simple to pinpoint and modify the behavior, though once again, this code is purely temporary.

3.4 Architecture

The word "architecture" is used in more than one context throughout this thesis, but here it refers to the base design and structure of the backend of the entire project, i.e., the architectural system that ties the database (SQL) to the logic (Domain Model) to the presentation (web services) with reference to the data involved (not the user interface).

3.4.1 MVC

MVC, which is built on the idea of separation of function between the Models, Views and Controllers, is a familiar and proven software model. Each component of MVC can be related to a basic concept:

- Models—the structures on which the program is built
- Controllers—the logic that manipulates the Models
- Views—the output, which is consumed by the user; the user could either be a person or computer

In other words, MVC is a combination of the classes or objects (Models) that are manipulated in the Controllers, and rendered in the Views.

Using MVC, it becomes easier for programmers to see the user interface than it would if they were using an alternative such as legacy ASP that would combine all of these into a single .asp file. As a result, such code would become intermingled with other components of the programmer's software, leading to tangled and scattered code [16]. If markup contains logic and program structure, it is simple to see that the separation is not there; An abundance of if-then statements, conditional logic and any sort of data access in the markup itself is a clear combination of responsibilities. Not to mention, there is also the violation of the "Don't Repeat Yourself" (DRY) concept in programming. Ideas in a system should be represented in just one place [17]. Consider an application in which three .asp pages need a list of Widgets from a backend database. In this case, we need to access data in three different places in code. At a later stage, the application is modified so that the three .asp pages need a list of a subset of all the Widgets (say, all Widgets with wings); using legacy ASP, we would have to make changes to our code in three different .asp files. However, using an MVC architecture for such an application would require code changes in only one file. As a general rule in MVC, if the programmer wishes to change the layout of something in the output, they go to the View; if they wish to change the logic or methods

in anyway, changes would be made in the Controller; while structural changes to the classes are made in the Models. As an *interface* with which to program, this is very clear and simple. Figures 3.4 and 3.5 show how logically these are set out in a few modern MVC architectures.

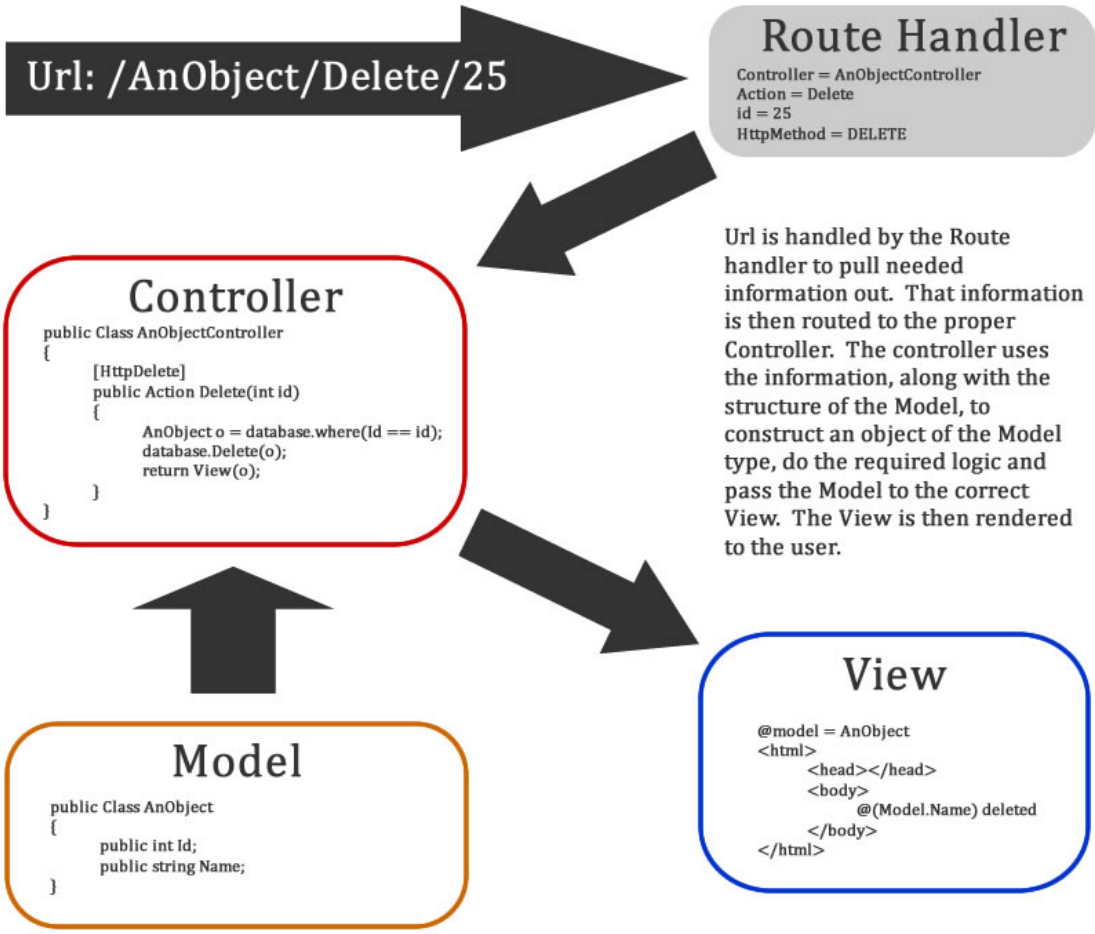

Figure 3.4. Visual representation of the MVC configuration.

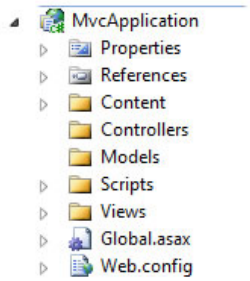

Figure 3.5. Default folder layout in Visual Studio for a .Net MVC 3 Project - note the separation between Models, Views and Controllers as separate folders themselves.

MVC is a logical and well thought out model for software development; it has been used and tested for years and continues to prove its merit. Settling on the MVC architectural backbone proved to have many advantages.

3.4.1.1 Asp.Net MVC

To extract the benefits of using the MVC architecture in a Windows Server environment, we chose Asp.Net MVC 3 (with the Razor rendering engine) as our architecture of choice for the View layer. The Model layer in Asp.NET MVC 3 was intuitive, extendable and was able to make use of the existing EF4 to automatically produce classes as Models themselves; as a result, we were able to use EF4 to generate a large portion of our Model layer automatically with the tools provided by Microsoft. Figure 3.6 shows the construction of a very simple model and Figure 3.7 shows how it is directly referenced in a view.

```
public class Company
{
    public string Name { get; set; }
    public string TaxCode { get; set; }
}
```

Figure 3.6. An example Model to show how the Razor engine renders Views.

```
@model Company
<h1>@Model.Name</h1>
<div>
    <span>Tax Code</span>
    @Model.TaxCode
</div>
```

Figure 3.7. Using the Model supplied in Figure 3.6 the Razor engine uses "@" symbols to indicate code segments; the Razor engine also parses through the code and marks where code starts and stops.

However, using Asp.Net MVC 3 did not come without its own set of problems. As a result of being a relatively new web-presentation technology, there was little information about some of the unique situations that we encountered during development. We discuss some of these problems later in Chapter 4 .

3.5 Intermediate Interface

Our solution was to use MVC to provide a universal API that could be used by any program that followed its input guidelines; that way, even if a client wanted to write their own program to interface with the provided back end they could do so. This causes additional security concerns, and so we had to employ additional means to account for malicious users.

3.5.1 Web Services

Web service supports machine-to-machine communications over a network; though the interface is not only limited to machines, it is designed to output machine processable formats which are often hard to decipher [18]. Although the above definition focuses on the SOAP/WSDL implementation, the idea is the same when dealing with REST.

Web services are prevalent throughout the internet; most websites such as Facebook, Twitter, Amazon etc. utilize web services to provide hooks and entrance points to third-party programs while still securing critical information from malicious users. All user actions (user clicking a button, dragging, moving etc.) can be integrated into a service, and thereby be simulated using programmatic commands if the API is available. Programming APIs are often seen as the part of your application that faces another program. A comparison can be drawn with GUIs as a GUI is the part of your program that faces a user. Similarly to GUIs, it becomes a challenge to keep an API clean and well-documented as well. Without the proper design, APIs can quickly become cluttered and confusing. A Web Service interface is more of a command line interface, often with cryptic or non-existent error messages (many services fail silently on error). It is up to the developers to not only document the interfaces, but provide good user feedback just as in any other sort of interface, though this often seems to be forgotten. A good example of well-documented services can be seen at the Facebook Developers website [19]. Just as a user can get frustrated with poor error-handling in any web-application, it may be argued that it is even more disconcerting when a simple test error is sent back without a description or nothing at all. Because it is generally a machine communicating with Web Services, it is possibly assumed that user-error will be much less common; and while this often is the case, the end users are still prone to human error. Moreover, the programs that consume the services often times use the error messages provided to present to the end user. This problem of silent failure is what happens when a request is sent, fails on the far end and never returns any sort of response or status. It raises several other design questions: does the consumer wait for a certain period of time? How long should they wait? These are questions that should be answered not only in the documentation, but also in some sort of status report; not to mention that silent failure should never be an option.

There are countless other interface concepts that have yet to make it to the mainstream Web Service sphere such as internationalization/localization; as the web is a true interna-

tional medium, it is detrimental to ignore other cultures and languages and such things should be introduced into services as well.

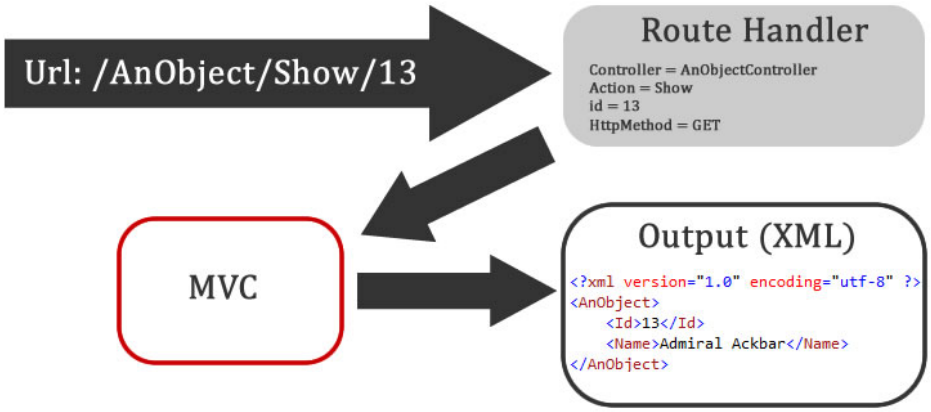

Figure 3.8. From request to Web Service output; note that the incoming request is a GET command, and it is retrieving a record, staying within the RESTful design concept.

Web Services are becoming more and more prevalent as more mash-ups and programs that communicate across the web are popping up at an unprecedented rate. This is an interface that will become only more common in the future of web computing and should be treated as any other interface as far as ease of use. Using Asp.Net MVC we were able to adhere to these guidelines, as shown in Figure 3.8.

3.5.1.1 Localization

Enporion's plan is to expand their market well outside of where it currently is, this includes extending into many other languages. This is becoming increasingly important as the business landscape of the world becomes more and more globally interconnected.

The current application was presented strictly in English. Moreover, this English was hard-coded into the system, making modifications very cumbersome as each program would have to be altered line-by-line. Because the inception of the new development, our team made sure that all user-viewable text was localized. The approaches for this differed in implementation for the user interface and the back-end, but the idea was the same: to centralize an area where all text could be kept and easily translated by someone if the need

arises. Additionally, the translator will be able to translate these files into another language without requiring any knowledge of coding; the interface is represented as a spreadsheet with the layout shown in Figure 3.9.

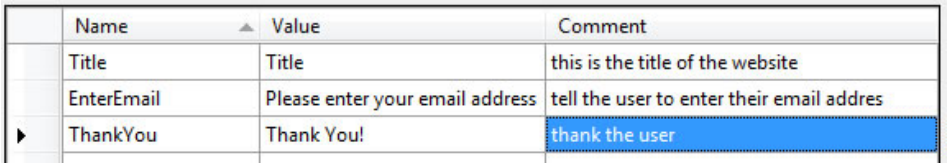

Figure 3.9. Example .resx file in the Visual Studio interface.

This was achieved int the .Net backed by using `.resx` files, which are specialized xml files that are read as resources by the system and applied based on current user or browser settings if available (default if not). For example, if a user has their browser culture set to "de" (German) the server would then search for the `.de.resx` version of the resources. If the German resources are available, it would serve them up; if not if would drop down to the default, which is "en-us" (US English) in our case. With these `.resx` files, resources are referred to by their strongly-typed names; if the example file in Figure 3.9 was named `Resources.resx` (the default) and the coder wished to use the `Title` resource, it is referenced with `Resources.Title`.

3.5.1.2 Security

The prospect of this API being opened to further development, possibly a separate organization, meant that we had to design it to be robust against external access. To do this we came up with a stateless method of tracking internet protocol (ip) addresses of logged-in sessions; this is important as Hypertext Transfer Protocol (HTTP) is not a stateful protocol. These sessions are tied to a specific ip and session identifier. Session owners can only manipulate items within their own organization and within their own privileges granted to them; this structure can be seen in Figure 3.10. It should be noted that there are exceptions implemented to this rule for Enporion's own administration purposes. This is all required to be sent over HTTP over Secure Socket Layer (SSL) which is known as

HTTPS, so the transmissions are not in danger of being hacked if intercepted, because SSL is asymmetrically encrypted and considered secure. It is also extremely modularized and pluggable using the provided `ActionFilterAttribute` base class that can be inherited, extended, and supplied to entire Controllers or sets of Controllers; this provides the ability for the code to be written in one area and applied to Controllers across projects with attribute painting (Figure 3.11) [20].

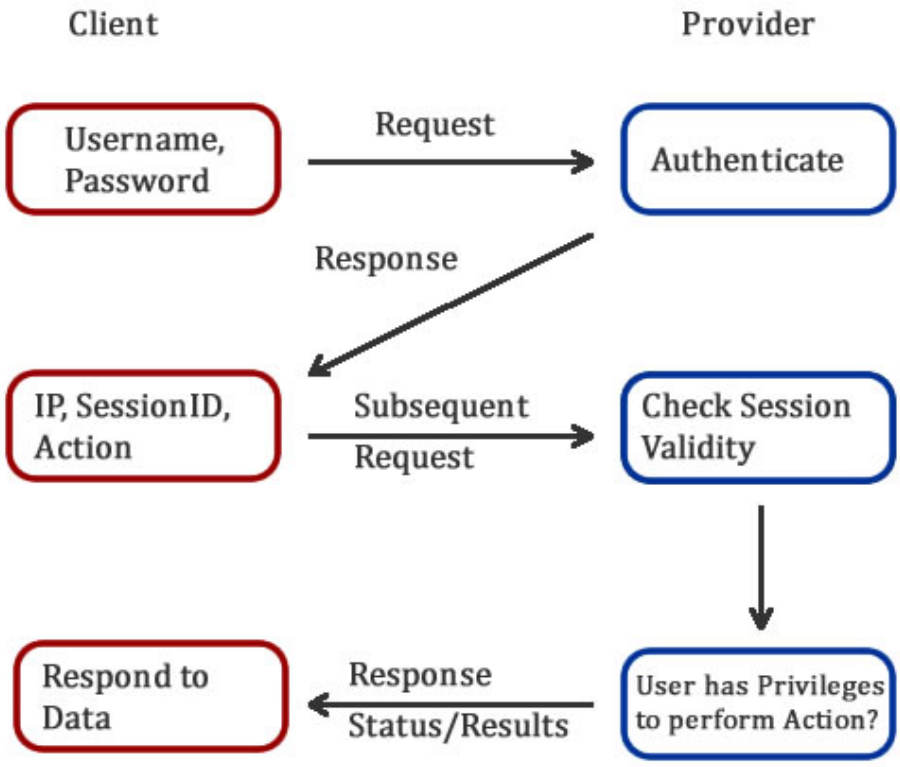

Figure 3.10. The basic model of how authentication, re-authentication and actions are handled by the web service.

```
[PrivilegeRequired(Name=Settings.ADMIN)]
public class AccountController : Controller
{
    //Controller Code Here
}
```

Figure 3.11. This is attribute painting of an entire class; the attribute will be applied across the entire Controller, requiring that the user is an administrator; no matter how many methods, pages or services the Controller serves.

3.5.2 REST

REST stands for REpresentational State Transfer; it was defined by Roy Fielding in his doctoral dissertation in 2000 [21]. REST is the mapping of HTTP actions to Create, Read, Update and Delete operations (CRUD) [22]. HTTP has four main actions: POST is submission of data, which is logically equivalent to Create; GET which is a pure data request, equivalent to Read; PUT, which is submission of data relating to existing data; PUT which is used for updating, and DELETE which is similar in structure to a GET, but is purely for object deletion.

Mentioned above is the fact HTTP supports these, but HTML, which is the implementation we use for HTTP, does not only GET and POST are supported. So in general practice, PUT is represented by a POST, and DELETE is represented by a GET [23]. This is where the synergy between the programming interface comes into play. MVC is used to simulate the unsupported methods. MVC frameworks such as Ruby on Rails will simulate the appearance of the other two methods even though they are not truly implemented. The lack of a complete mapping requires hidden fields or other programming work-arounds to simulate the missing methods; this is shown in Figures 3.5.2 and 3.13.

```
<%= button_to 'delete', obj_to_delete, :method => :delete %>
```

Figure 3.12. Code for a DELETE action in Ruby on Rails.

```
<form method="post" action="/obj_to_delete/1">
    <div>
        <input name="_method" type="hidden" value="delete" />
        <input type="submit" value="delete" />
        <input name="authenticity_token" type="hidden" value="x" />
    </div>
</form>
```

Figure 3.13. HTML markup output from the code, simulating a DELETE.

MVC has the ability to output a specific View based on the request of the user. The external interface can be configured to output different formats such as the common XML or JSON as well as any other format that may be desired. The Controller logic decides which format to render, and that information is passed to the View, which in turn is exposed to the user (as the interface). This provides an extremely modular and flexible design.

REST is an alternative interface to other web services that have also made themselves prevalent such as SOAP/WSDL—but it is the simplicity of REST that is a draw to many developers. The ease of integration with an MVC framework makes REST a great interface, both for programming for and consuming from an exposed API.

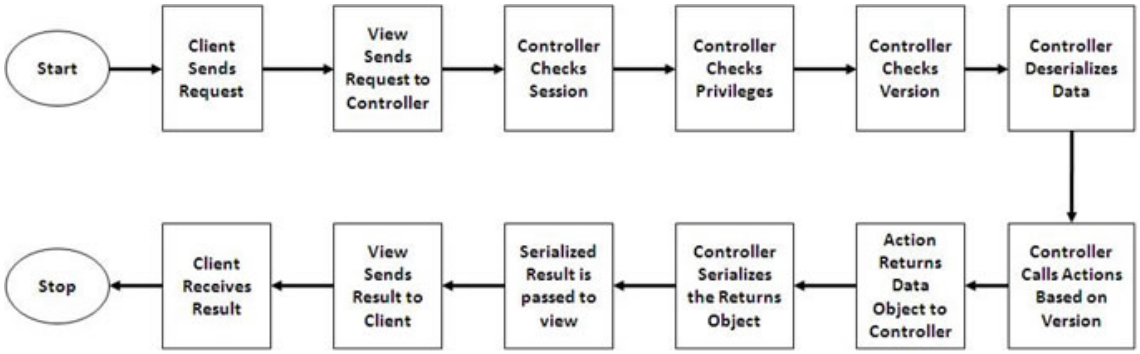

Figure 3.14. Flow diagram used in planning for the development from data store to web service interaction.

3.6 User Interface

The face of most modern consumer-facing programs is the Graphical User Interface (GUI). If the user types, clicks, acts or expects a certain behavior, a program should respond with that behavior or if unable to, an explanation of why it did what it did. There are infinite ways to present a GUI to the user, so this can be a difficult decision to make. In our case study, this was not an item up for discussion, though many other capable options were available.

3.6.1 FLEX

Adobe FLEX was to be our chosen GUI framework. Flex is an open source framework for building web and mobile applications with a common codebase among a large community of developers [24]. This framework did prove to supply what we needed, though it was not without some learning. A very powerful ability of this framework is the ability to quickly adapt and change to multiple platforms. With the prospect of having mobile implementations in multiple environments such as phones and tablets, which in turn have varying operating systems, the flexibility of the framework was key. Initially, it seemed that de-coupling the GUI from the MVC framework we were already using was counter-productive; however it did force us to truly modularize each end of the code-base to work

cross-platform. Because the technologies chosen were so different, we had to make sure that the communication went smoothly; additionally, it coerced us into in-depth documentation, as detailed explanation of behavior was vital because cross-programming debuggers are not available to my knowledge.

FLEX is not only able to supply our end users with a rich interactive environment that is visually-appealing, but also supplies solid performance. The fact that the initial download pulls all of the actual application code ensures that there is very little data transfer compared to a traditional web-based application. Once the program is downloaded, only XML, and sometimes simple POST requests are transferred back and forth between the server and client, greatly reducing the bandwidth required to run an application in comparison to the legacy method where full pages and markup were included.

My focus was mainly on the backend of the project: from the database up to the REST services provided via the Asp.Net MVC. Nalin Saigal and Matt Spaulding had their focus on the front-end, which was FLEX, along with providing and consuming XML data, which the REST services required as inputs and divulged as outputs.

3.7 Summary

With the freedom that we were given with this project, the exploration of new technologies had few limitations. The combined expertise of the employees and project members aided in choosing very powerful and capable solutions to the problems encountered. We believe that the backbone of an application is as important as the code itself, and so we paid a lot of attention in choosing the various avenues, while ensuring that these avenues were robust and future-proof.

CHAPTER 4

TESTING AND DEPLOYMENT

Testing and deployment are a constantly changing and evolving processes; this is represented by the common software life-cycle diagram, displayed in Figure 4.1 as an infinite loop. As for this case study, deployment is used in the sense of deploying to a test environment and the methods used to deploy.

Figure 4.1. An example of the software life cycle.

4.1 Deployment

Our deployments are broken down into two steps: version control and automated deployment. The deployment is coupled tightly with the version control and aids us in tracking problems much faster than a traditional deployment method, while the version control allows us to roll-back to or review a previous version if needed.

4.1.1 Version Control - Subversion

Version control is a necessity in any sort of software development cycle, no matter how minute it may seem. The overhead involved with setting up a version control system is small compared to the advantages that it provides by supplying the ability to look back at code changes, possible code branches or even roll back to a certain point at which everything still worked. We chose to use the very common and proven standard version control method of Subversion (also known as SVN). In its simplest form, SVN allows programmers to check in code at any given time with comments to accompany the check in. There are many Subversion clients available to programmers today, for example, Tortoise and AnkhSVN (both of which we used), which allow anyone with proper access to have the ability to review old code, compare differences in versions, roll-back code to a particular version, branch code off in an experimental direction and various other options [25, 26, 27]. We practiced the method of "commit often and early," meaning that we would basically commit any time (often) we wrote any non-trivial amount of code, whether or not it was a complete code segment (early). This helped us avoid the dreaded crash without saving or all-too-common practice of making a change, only to delete it, then realize that a week later you really want that code back —now it was all available.

4.1.2 CruiseControl

CruiseControl is a tool that allows for a continuous build system that can be implemented into a deployment process [28]. It takes the burden off the developer to make sure that synchronization is happening across multiple facets of a project. To further explain this, we can take a look at how our team integrated it into out deployment process.

CruiseControl "watches" the subversion destination directory where our code check-ins arrive; at set time intervals (one minute in our case) it will look to see if any changes were made in the files. If anything was changed, whether in the application code or even the build files, an event will be fired, executing everything within the CruiseControl **config.xml** file that instructs it what to do. In our case, it would compile and test using NAnt, an

automated build tool that we discuss later. If the compilation and tests were successful, CruiseControl publishes to our test server environment. This can be monitored several ways such as via email, checking the built-in intranet site, or even a Windows tray icon, which is what we used because the Windows tray icon is always visible to us, and so it ensures not having to switch to a different window to monitor the compilation results. Moreover, CruiseControl instantly notifies you of problems that may have arisen with your code, whether in compilation or functional testing. This is greatly helpful as often times in development, it is easy to forget one of these steps and deploy a faulty product, or even worse, avoid a constant testing/deployment environment where the team can get too far behind and let the errors pile up in the background without anyone knowing.

4.1.3 NAnt

NAnt is a .Net build tool that behaves much like a make file for a Windows system [29]. Make is a well known build tool used in languages such as C and C++ where programmers can put together a list of tasks to perform each time the `build` command is called from the prompt. Because of Make's popularity among C and C++ programmers, there have been various translations of Make for newer programming languages. NAnt is a result of such a translation of Make for .Net. Additionally, Nant also provides the ability to run testing frameworks inline without user intervention (discussed further in Section 4.2). As long as all relationships and dependencies are laid out clearly in the `build` file, everything will execute, returning a success or failure, and a detailed log of the entire event; and example section of the build file is show in Figure 4.2. In our case, NAnt communicates with CruiseControl, which would then notify us of failure if need be. This is a great system, and also saved much difficultly, but it also presented a few more problems than anticipated.

```
<target name="Enporion.Localization">
    <csc target="library" output="${enporion.localization}"
        keyfile="${signing.keyfile}" rebuild="${force.rebuild}">
        <sources basedir="./Enporion.Localization">
            <include name="./Properties/*.cs" />
            <include name="*.cs" />
        </sources>
        <references>
        </references>
        <resources dynamicprefix="true">
            <include name="./Enporion.Localization/Properties/**.resx" />
        </resources>
    </csc>
</target>
```

Figure 4.2. Build section within NAnt, displaying compiler sections (`csc`), sources, references and localization files (resources).

As previously mentioned, the choice to use the newest .Net 4.0 Framework was not without penalty. As NAnt is an open-source project, it sometimes has the tendency to lag behind a bit in development as it can not necessarily be developed alongside the newest proprietary technologies. What this meant for our team was that a lot of the features of .Net 4.0 were not implemented into NAnt when we wanted to use them.; this proved to be true when implementing the localization and the latest EF4 technologies.

4.2 Testing

Testing is essential in any large development project. Without some sort of testing, it becomes near impossible to locate bugs that are generated without diligent user interaction and direct application use; as mentioned previously, this is very time consuming and inef-

ficient. Considering that we had already decided to automate building and related events, it was logical to automate testing as well.

4.2.1 Unit Testing

Unit testing is the practice of breaking out portions of functionality and test them as smaller sections. Changing code in one location in a program often affects the program in multiple other locations; without directly testing all of these locations, it would be difficult to realize if the developer broke any code. With unit testing if one of these smaller units breaks, these tests can quickly locate the problem.

NAnt has the ability to run tests automatically on a build and notify designated people or systems of the failures. Thus, as soon as new code is uploaded to the source control, it is built, then the tests are run. Upon failure of the tests, NAnt can be instructed to roll back the code, ideally preventing downtime for the users. The testing framework used for the .Net backend was NUnit, and for the front end was FlexUnit [30, 29]. Both frameworks can also be used with stand-alone GUIs that will run the tests reporting the passes and failures as green and red indicators respectively. The GUI is a very intuitive system and easy to evaluate at a glance which is easier to implement than an automated system, but lacks the continuous testing.

4.3 Summary

Deployment and testing are key to the success of this project along with being the only elements that are run daily, and are implemented purely to keep everything synchronized and bugs located as soon as possible. If this system is broken, it can be difficult to recover. For this reason we chose a proven and simple approach that is also highly visible and effective.

CHAPTER 5

RELATED WORK

Technologies that were chosen in our re-engineering were form a pool so large it is sometimes difficult to comprehend. There is no shortage of powerful software setups and different architectural choices to choose from, in fact, there are so many it is often quite difficult. Everything from the data-store all the way to the final user interface, there are choices to be made. The key is to choose the most effective for the given situation; there is no holy grail in this quest, and the choices are constantly changing.

5.1 Alternative Data Storage

Nearly any software needs to store data of some sort. Many times it may just be settings or small amounts of data, alternatively there is often a need to store large caches of data. In addition, the type of data, speed of search and other factors weigh in on the decision of the data storage.

5.1.1 Extensible Markup Language

Extensible markup language is a very capable and easy to understand type of data store; it is employed by many projects quite successfully. Not only is it simple, but almost any programming technology can parse it, and it requires no special back end or interface to use as they are simply text files. XML is highly structured and, as mentioned before, it is the medium of communication chosen for the different components of the project. XML can be implemented with a DBMS and even as what could be considered a relational database, though these are much younger and possibly less robust systems [31]. All this in mind,

the decision to stick with a database in more of a traditional sense seemed to be the most intelligent approach. A system like this would be very interesting to work with as it is a less mature technology and could prove to be quite useful if a project had the time to adopt something like this.

5.1.2 Non-Relational Databases

Non-relation databases (NRD), more commonly known as NoSQL are becoming more and more popular and useful as they are developed further. Although fully capable in most database implementations, non-relational databases specialize in a less structured environment (as one would assume from the name alone). Leavitt claims that their primary advantage over the traditional SQL database is the way they handle unstructured data such as word-processing files, email, multimedia, and social media effectively [32]. Once again, this system showed a lot of promise but did not specialize in the type of data maintenance that we were seeking. Experience with NoSQL systems in the future will be a necessity for a well-rounded computer scientist as it becomes ever more prevalent in development. It has been discussed that this may be a viable option for additional parts of the Enporion system that deal more heavily with documents.

5.1.3 Other Relation Databases Management Systems

MSSQL is far from the only option when choosing a RDBMS, as discussed, some other options were SQLite, PostgreSQL and MySQL [2, 4, 5].

SQLite is a fully contained library that runs on most operating systems. It is very easy to configure and use, basically plug-and-play with minimal configuration. SQLite does have large limitations when it comes to concurrent processing; for example, two SQL INSERT commands cannot be executed simultaneously, which means it was not an option [2].

MySQL and PostgreSQL are both very accomplished DBMSs with large followings and support; each one is open-source with a very large community of users [4, 5]. These two are more traditional and accomplish most regular T-SQL operations just as MSSQL does, often

the only difference being slight syntactical differences. In the past, scalability has been a bit of a concern with the open-source systems, but in recent years, with the use of systems like these in large projects such as Facebook that hold millions of records that are accessed incredibly quickly, it is being shown that these technologies are extremely capable. Mariella Di Giacomo discusses how all of these solutions are scalable and powerful, but does point out that MSSQL does have the ability to be clustered, which could prove to be important in the future with a possibly massive amount of data as with this project [33].

5.2 Source Control

There are countless source control systems out there, from the more loathed in the industry such as Microsoft Visual Source Safe, to the more prominent Team Foundation Server, Mercurial and Git amongst others [34, 35]. Our decision to go with Subversion was based mostly on experience within the group. Recently Git has become increasingly popular; it is similar to Subversion, but decentralizes the source control onto individual contributors, making each one a fully fledged repository. Without relevant experience at the time of the project, and a solid alternative in our grasp, the timeline was somewhat preventative of implementing Git or a similar alternative.

5.3 Summary

The technologies available in software development are a constantly changing and incredibly varied landscape. It takes a large amount of time and knowledge to sift through what is available and choose the best solutions for a project. Even in passing on an available technology, the knowledge gained in exploring it can benefit the developers and possibly the project in the future.

CHAPTER 6

CONCLUSIONS

Re-engineering software is a constant necessity in the competitive world of software. This case study focused on Enporion's recognition of this and their employment of the USF team to assist. Enporion provided us with an extremely open and collaborative environment; this enabled us to pool our knowledge and be very successful in the project. The process was systematically thought out and executed.

We approached the existing SkyWay bases system to deduce its shortfalls; those shortfalls were explored and documented in order to avoid the same mistakes and to also pinpoint possible pitfalls in our development. The experience of the Enporion staff was integral, as these potential problems were only obvious to those with experience in the existing system.

We also investigated many proven, as well as some emerging, technologies to choose the most advantageous combination given the available resources. Once again, a great deal of collaboration and utilization of prior experience and expertise was integral in making these important choices. Our group discussed everything from data storage through web services and front-end interface in order to provide solid options for advancements. A powerful combination of solutions and up-to-date technologies was the result of this diligent process.

Not only were the technologies used in the actual application important, but our building and testing approaches allowed us to stay on track and not get too far ahead of ourselves with application-breaking bugs and build errors. The automation of most all of the build process allowed us to focus on development and let the system catch our mistakes. This not only saved us a lot of time, but made us more aware of the changes that were being made, a valuable approach to any development project.

Along the way, we encountered many useful technologies that did not fall into the scheme of what we were trying to accomplish. This was not time wasted, rather time spent learning alternatives that very well may be better answers in future projects and developments (possibly even in the project at a later date). Staying current on available alternatives is a core component to a competent developer and any team.

Enporion declared this case study as a resounding success. The software developed accomplished the goals set out from the beginning, and even surpassed the level of completion initially set. Re-engineering will continue along the same path using the framework provided by this project.

LIST OF REFERENCES

[1] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377, 1970.

[2] Sqlite, 2011. http://www.sqlite.org/docs.html.

[3] Data development documentation, 2011. http://msdn.microsoft.com/en-us/data/bb419139.

[4] Mysql :: Mysql documentation: Mysql reference manuals, 2011. http://dev.mysql.com/doc/.

[5] Postgresql: Documentation, 2011. http://www.postgresql.org/docs/.

[6] Bryan Thomas. Solutions for highly scalable database applications: An analysis of architectures and technologies, 2005.

[7] F. Sohail, F. Zubairi, N. Sabir, and N. A. Zafar. Designing verifiable and reusable data access layer using formal methods and design patterns. In *Computer Modeling and Simulation, 2009. ICCMS '09. International Conference on*, pages 167–172, 2009.

[8] Ruby on rails documentation, 04/18/2011 2011. http://api.rubyonrails.org/.

[9] Linq to sql: .net language-integrated query for relational data, 2011. http://msdn.microsoft.com/en-us/library/bb425822.aspx.

[10] Nhibernate for .net — hibernate — jboss community, 2011. http://community.jboss.org/wiki/NHibernateForNET.

[11] Ado.net entity framework, 2011. http://msdn.microsoft.com/en-us/library/bb399572.aspx.

[12] dapper-dot-net - simple sql object mapper for sql server - google project hosting, 2011. http://code.google.com.ezproxy.lib.usf.edu/p/dapper-dot-net/.

[13] Update on linq to sql and linq to entities roadmap - ado.net team blog - site home - msdn blogs, 2011. http://blogs.msdn.com/b/adonet/archive/2008/10/29/update-on-linq-to-sql-and-linq-to-entities-roadmap.aspx.

[14] How to: Validate model data using dataannotations attributes, 2011. http://msdn.microsoft.com/en-us/library/ee256141.aspx.

[15] Extension methods (c programming guide), 2011. http://msdn.microsoft.com/en-us/library/bb383977.aspx.

[16] Sergei Kojarski and David H. Lorenz. Domain driven web development with webjinn. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 53–65, New York, NY, USA, 2003. ACM.

[17] A. Hunt and D. Thomas. Oo in one sentence: keep it dry, shy, and tell the other guy. *Software, IEEE*, 21(3):101–103, 2004.

[18] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web services, 11/02/2004 2004.

[19] Mobile apps - facebook developers. http://developers.facebook.com/docs/guides/mobile/.

[20] Actionfilterattribute class (system.web.mvc), 2011. http://msdn.microsoft.com/en-us/library/system.web.mvc.actionfilterattribute.aspx.

[21] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures., 2000.

[22] A. Shaikh, A. Soomro, S. Ali, and N. Memon. The security aspects in web-based architectural design using service oriented architecture. In *Information Visualisation, 2009 13th International Conference*, pages 461–466, 2009.

[23] Jeff Cohen and Brian Eng. *Rails for .Net Developers*. The Pragmatic Bookshelf, Raleigh, NC, 2008.

[24] Open source framework, web application software development — flex - adobe, 2011. http://www.adobe.com/products/flex/.

[25] subversion.tigris.org. http://subversion.tigris.org/.

[26] ankhsvn: Subversion support for visual studio. http://ankhsvn.open.collab.net/.

[27] Tortoisesvn. http://tortoisesvn.net/.

[28] Cruisecontrol home, 2011. http://cruisecontrol.sourceforge.net/.

[29] Nant - a .net build tool. http://nant.sourceforge.net/.

[30] Nunit - home. http://www.nunit.org/.

[31] Hongjun Lu, Jeffrey Xu Yu, Guoren Wang, Shihui Zheng, Haifeng Jiang, Ge Yu, and Aoying Zhou. What makes the differences: benchmarking xml database implementations. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 720–722, 2003.

[32] N. Leavitt. Will nosql databases live up to their promise? *Computer*, 43(2):12–14, 2010.

[33] M. Di Giacomo. Mysql: lessons learned on a digital library. *Software, IEEE*, 22(3):10–13, 2005.

[34] Mercurial scm. http://mercurial.selenic.com/.

[35] Git - fast version control system. http://git-scm.com/.