Project Evaluation

KSU Student Portal

Version 1.0

Submitted in partial fulfillment of the requirements of the degree of MSE

Javier Ramos Rodríguez

CIS 895 – MSE Project

Kansas State University

Table of contents

1. Introduction

In this document I will summarize my experiences while developing the KSU Student Portal Web Application. Since this project used many new technologies and tool many problems were encountered and not so much documentation was available to solve it. Many decisions had to be made to be able to finish this application.

I will explain my option about the development process that I choose the technologies and tools, the schedule and the effort. Then, I will explicate the main problems encountered during the development and the future work.

2. Process

2.1. Introduction

The underlying process used to develop this project was the Rational Unified Process (RUP). The RUP is an iterative software development process created by the Rational Software Corporation. Also, the RUP is not a single concrete prescriptive process, but rather an adaptable process framework.

The KSU Student Portal used a version of the RUP; some changes were made in the process due to different reasons. First of all, the process was not real iterative like many of the software process available today. It follows the classic waterfall model. It uses rigorous methods to create complex models to correctly specify the application to save time and avoid risk in the last phase of development.

It does have some iteration, and some tasks were performed before some previous activity was done. It also included many other techniques used in modern approach such as prototypes or been architecture first approach.

The project was done in three different phases that provided different artifacts. UML was used to model the system. One key factor of this project is that it used a **model-driven architecture**. The process was driven by the models created using UML. As mentioned in the documentation, in each phase the UML were changing adding more details as the project advanced.

In the requirements phase the idea was to create a use case model to capture the requirements of the system, a prototype was also build to help this task. In phase 2, the idea was to create the main architecture, so an extended use case model was built including all the system behavior. At his phase, the platform was still yet to decide, the objective was to create a detailed model without adding any implementation detail and focusing only on the problem domain. This way, I created a platform independent model that can be implemented in any existing platform without much effort.

Next, I explain what was done in each phase:

1. <u>Phase 1</u>: Requirements
    a. Requirements Analysis
    b. Project Description
    c. Prototype
    d. Use Case Diagram
    e. Use Case Specification
    f. Initial E-R Diagram
    g. Initial Project Plan
    h. SQA Plan

2. Phase 2: Architecture
    a. Use Case Diagram Revision
    b. Sequence & Collaboration Diagrams
    c. Class Diagram
    d. E-R Revision
    e. UML Data Model
    f. Relational Schema
    g. UML Design Revision
    h. Formal Specification
        i. Alloy Model
        ii. User Model
    i. Test Plan
    j. Formal Inspection Checklist

3. Phase 3: Implementation
    a. Implement Relational Schema in Database
    b. Create Entity Beans From Tables
    c. Implement Relations Between the Entity Beans
    d. Write initial EJB-QL Queries
    e. Create Session Beans
    f. Create Model Facade
    g. Publish Methods
    h. Test Methods
    i. Create JSP pages and backing beans
    j. Create additional classes
    k. Improve the view
    l. Create Test Cases
    m. Fix Bugs.
    n. User Manual
    o. Component Design
    p. Test Plan 2.0
    q. Formal Inspection Checklist 2.0
    r. References

2.2. Evaluation

The waterfall model was very useful to learn the basics of software development, it is clearer than the most modern approaches and, in my opinion, a student should start learning this process before going into more modern approaches. But I also faced the problems of this approach. I found some requirements in the implementation phase that I couldn’t find earlier in the process and I had to go back to make the appropriate changes.

Also, when implementing the system I decided to implement most of the model before implementing the view. I took this decision because I wasn’t sure what technology use to create the view. I was doubting between JSF, Struts or Spring. The problem is that I implemented and publish in the model façade many methods that ended up not been used. Sometimes the problem was that the method parameters were not correct. Further some, functionalities such the messaging system were implemented in the model but not in the view due to time restrictions. This decision was positive because it allow me to start implementing without knowing about the view, achieving a completely separation between the model and the view. In case that the technology to generate the view is known before starting the implementation, I would suggest create first the entity beans from tables and then, instead of start creating the session beans, begin creating the view and create the session beans as needed.

The approach that I used can be good choice when you have different team members working on the model and on the view. The server side developers working on the model can publish the methods in the façade that the members that work on the view request. My problem was that I didn’t have any client in the view so I had to “guest” the methods needed and this caused the creating of methods that were not used, increasing the effort and LOC. Once, the implementation was done these methods were removed except for the messaging system that were kept to be able to add the functionality later on.

The model driven approach, in my opinion, was very useful. It has many advantages. It proves that we can *reuse models* saving a lot of time and money. Also, having several layers of abstractions helps clients and developers to understand the system.

The use case model specifies the system explaining “what” the system must do without specifying how it should do it. These requirements specification is very useful in the early stages of development to set the requirements. In phase 2, we included the behavior (“how”) of the system and we introduced a platform independent model that can be implemented using any technology. This approach can be very useful to directly implement prototypes to test different architectures and technologies. It also helps to understand the problem domain without including implementation details that is useful for developers and customers.

The last phase includes an implementation dependent specification where the complete system behavior is explained. This model is helpful for developers, in case that the software is delivered to other developers, this model will be used to understand the technology behavior. The only drawback is that it requires more documentation but the benefits are well worth it, especially for a long term use.

Another issue is that formal specification was not very useful for this project. The application is quite standard in the requirements. The key of this project sis the process, development and technology used and not so much “what” I’m building. This project could be any other web application. The goal was develop a web application using the RUP and the J2EE platform to test new technologies such EJB 3.0 and JSF. So, since I proposed the requirements and they were really standard, formal specification didn’t provide much information. Formal specification is quite useful in many other context were the requirements are more difficult to get and there are more relations between entities.

Building the prototype was useful to help get the requirements and to test the technology. JSF uses a new approach and without the prototype the implementation would have taken too much time. Also, it helped to find more requirements and face problems before the final implementation. Ironically, the message system was implemented in the prototype and not in the final product.

The prototype helped to realize that the original message system design was incorrect and the data model needed to change. It also showed that the implementation was not too easy. Since it was not a critical use case I decided not to implement it.

Finally, not having a client it was a drawback since I had to figure out the requirements and I missed some use case such change password or delete language/course that were added late in the process.

3. Technologies

This project was implemented using really immature technologies. One of the goals of the project was to test these technologies. In this section I'll give my opinion about these technologies.

3.1. Model

To implement the model I used Java 5 as programming language and EJB 3.0 as the main technology of the J2EE platform.

I'm very happy with the new Java 5 language. I tried to use the new features as much as possible to increase performance and reduce the LOC. Java annotations help to improve maintainability by adding Meta data to the java code that can be interpreted really quickly by the compiler. This way the Meta data related to a class in included within the class and not in a separated file like it used to be. Annotations are used in the EJB 3.0 often times, for example, the OR Mapping information is included within the entity bean and is interpreted by the entity manager, this is faster and easier to maintain than having a separate XML file like other OR Mapping tools such Hibernate use.

I also used generic list and collections to pass data between modules this way we keep the representation independent. I pass generic list between modules that can be reused very easily. The model will fetch the objects from the relational database using OR Mapping and save the result in a generic list that is returned to the view. Many components in the view such data tables support generic list and the binding is done automatically from the list to a table. In the cases where generic lists are not supported by the UI components a transformation is done to adapt the list to the client (transfer object pattern). Enhanced loops that support generic types are also used by the algorithms to take full advantage of this new language. For more information check [16].

I have nothing to complain about the EJB 3.0 technology. People that worked with EJB 2.x will find this specification a lot easier to use and with several new interested features. The key feature besides simplifying EJB 2.x is the OR Mapping approach that allows the developer to work in terms of objects simulating a virtual OO database.

The only problem was the immaturity; at the time I started the project the specification was still a draft and there was only one early implementation developed by Oracle. There wasn't that much information or documentation, and the only reliable document to look for help was the specification itself. But this situation also helped me to realize that the best place to look for information are the professional documentation such as specifications and *JavaDocs* instead of internet tutorials that often times are incorrect.

Since the Oracle implementation of the specification was too immature I had to face several problems. The main one was that the lazy fetching did not work properly and I had to implement it myself by calling the model to fetch the lazy collections. The EJB container should take care of the lazy fetching but it didn't. JDeveloper provides a function that automatically creates the CMP Beans from the relational tables adding the corresponding annotations. It also checks the database relations and set the attributes in the object to maintain the database relations. For example, the blog has a collection of entries (0 or more), this is reflected in the OO world by having an object Blog that has a generic collection on entries as an attributes. By default, the entity manager performs lazy fetching on collections to avoid getting so much data with the object. With single objects it performs eager fetching by default. When I created the entity beans from tables and I had to make several changes. Sometimes I was interested in eager fetching and other times on lazy fetching. For example, when getting the user information I get also the blog information (eager fetching) since the blog contains only one attribute that is very coupled to the user. But I don't get the blog entries, so when getting the user information the attribute blog will have the corresponding value, but in the blog the attribute entries (a list) will be null.

In theory, when referencing the list the entity manager will automatically fetch the data and return the corresponding collection. The problem is that this is only true in the persistence context inside the EJB container, once we leave this module this feature doesn't work. In this project I implemented the layers pattern to separate the view from the model, acting the view as a client for the model. Therefore, the view is not in the EJB container. It wouldn't make sense to have it in the same context. So, the lazy fetching doesn't work outside the container, and in my opinion this is not very useful since most of the times we will have clients requesting data to the session beans and using it outside the persistence context. It took me a while to realize about this problem and found out that the null pointer exceptions were not my fault. To solve I had to implement methods to get the collections from the model when the view needed them.

Another problem that I faced was updating and deleting objects. I had to set the persistence manager configuration to configure the deletion of items. For example, when deleting an event I also delete the filter but when deleting the filter I don't delete the country, state, course or language. All this configuration parameters, including the fetch type, are done with Java 5 annotations and are interpreted by the persistence manager; check the entity beans source code to see this configuration.

Finally, I had also problems with the EJB-QL language, the problem is that different implementations differ in some aspects from other ones just like happen with standard SQL. Some string functions were not supported by Oracle implementation but this didn't suppose a big problem.

3.2. View

The main technology used to implement the view was the Java Server Faces (JSF) along with JSP. Just like EJB 3.0, this technology is really immature and it was too risky to use it in the project.

A prototype was built to test these new technologies. Building it I found many troubles using the JSF that made wait until the last moment to decide which technology use for the view. This is why I delayed implemented the view and I did most of the model before starting the view.

After looking to many tutorials on the internet and I realized that the problem was that JSF were too complex to start using them from scratch, so I decided to read the specification to clearly understand the technology. This way, when I implemented the real project I didn’t have much trouble. It was really easy to validate inputs or use internationalization using this technology. Functions that were getting too complex in the prototype ended up been easier once I understood the technology.

This doesn’t mean that JSF is perfect. Some developers will complain about some decisions made in the specification. Some UI components such data tables work with generic types which is really good because it helps to reuse the items. But other components use their own types such the menus, so I had to implement the transfer object pattern to adapt the lists to the view.

Using backing beans is very useful to add logic to the view without affecting the JSP pages. Although, we should let the model perform the logic in some cases is necessary to perform operations in the view. For example, when editing information we show the current value in the menu and then the rest of the list; we ask the model which is the current value and to get the rest of values. But the view is in charge to check the list and remove the current value from the view to avoid that value appear twice in the list. The approach that I use is overriding the set method of the component in the bean and set the proper value, and then the view will be affected.

4. Tools

The application was built using standard libraries been *JDeveloper* as the main IDE. In general terms I’m really happy with this tool. IDEs have improved considerably in the last years. My first project was done using just a standard text editor and I had to code everything by hand. On my second project I used Eclipse as a main IDE; the productivity increased considerably comparing to my first project. The second project was larger but it took me the same development time to implement it.

JDeveloper adds several new features comparing to the Eclipse version I was using in 2004. JDeveloper allowed me to graphically manage the configuration files instead of doing it by hand. It also configures the application server and takes care of the deployment.

It generates lots of code such creating the entity beans from tables or creating the backing beans. The nicer feature was developing a web application graphically dragging & dropping the UI components into the JSP pages and having JDeveloper taking care of everything.

I wouldn’t recommend JSF without using a graphic interface since it will take a long time to develop the view. At the moment I started implementing the project JDeveloper was the only IDE that supported this feature.

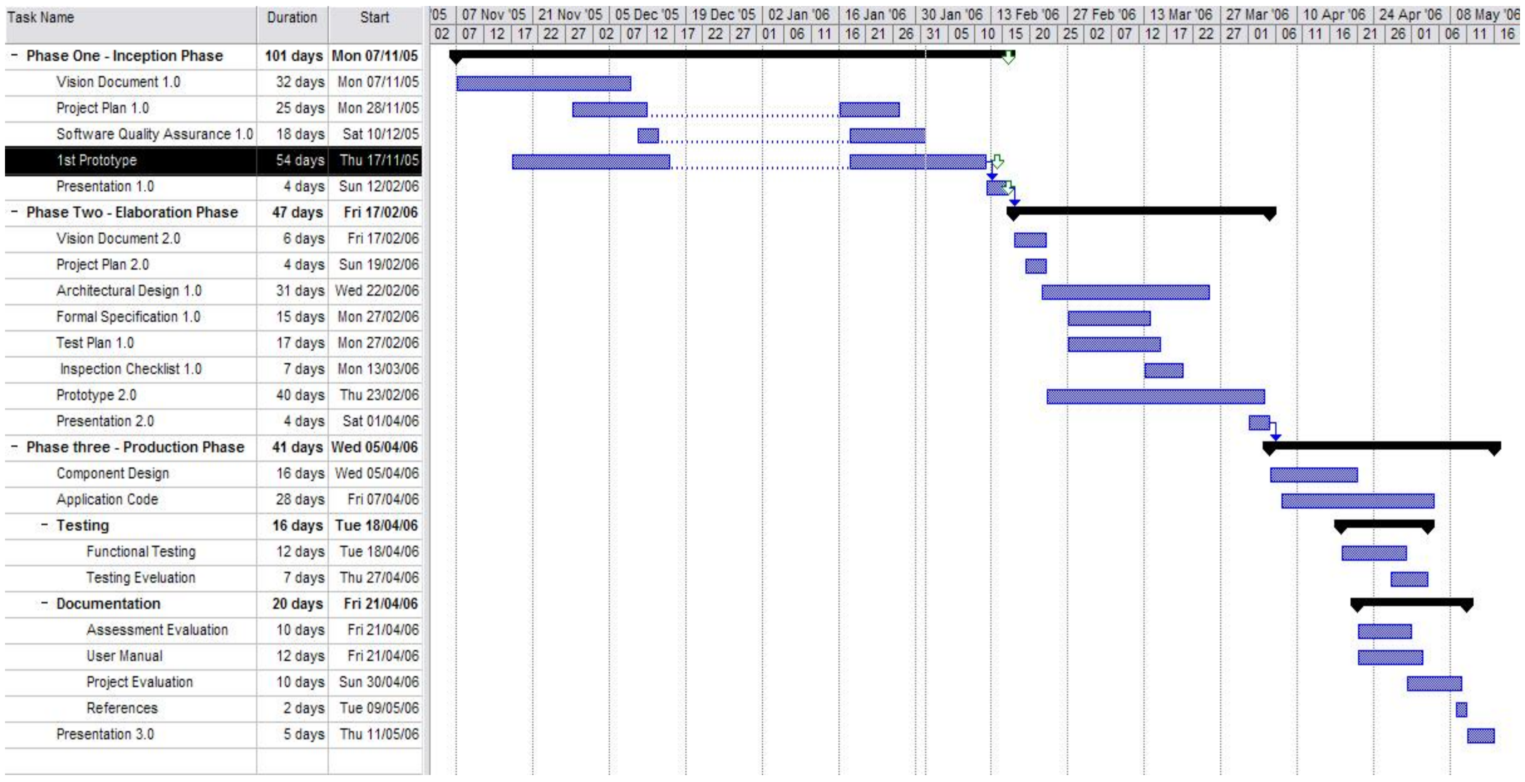
The goal of this paper is not to show all the JDeveloper features. For more information check [2].

The main problem with JDeveloper was that it has some bugs since is still an early version. It crashed many times during the development and I had to reinstall it many times. It also consumes many resources (around 300MB of RAM) and some times it gets too slow. When the number of web pages got too big the page flow editor crashed and I had to configure JSF manually.

Summarizing, I very glad with JDeveloper. I think is the best IDE available right now, but at this moment I wouldn't recommend using it since is not reliable yet.

5. Schedule

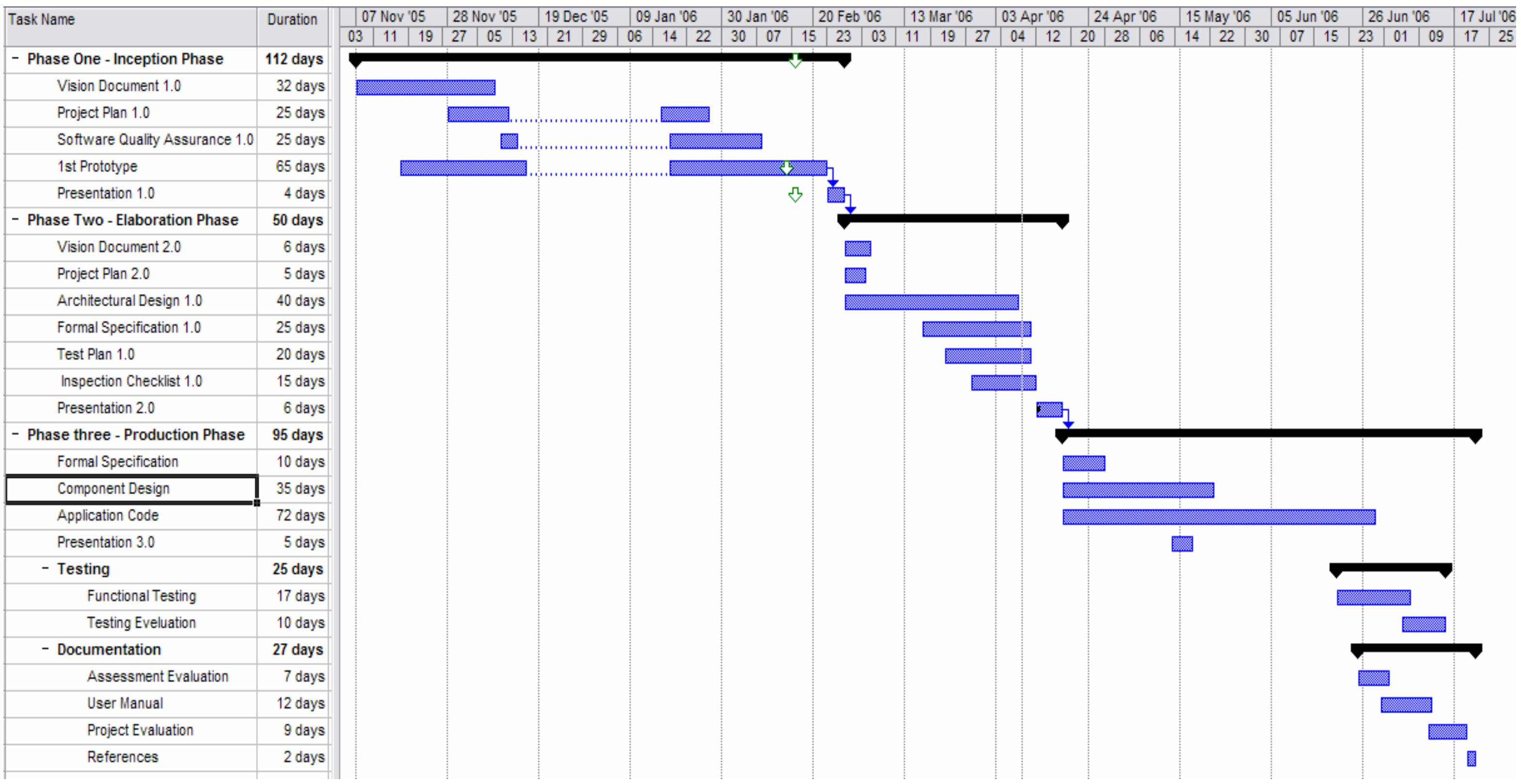
The project started November 2nd 2005 and finished August 11th 2006. It took around 10 months to develop the application. The initial plan is shown in the next figure:



Task Name	Duration	Start
− Phase One - Inception Phase	**101 days**	**Mon 07/11/05**
Vision Document 1.0	32 days	Mon 07/11/05
Project Plan 1.0	25 days	Mon 28/11/05
Software Quality Assurance 1.0	18 days	Sat 10/12/05
1st Prototype	54 days	Thu 17/11/05
Presentation 1.0	4 days	Sun 12/02/06
− Phase Two - Elaboration Phase	**47 days**	**Fri 17/02/06**
Vision Document 2.0	6 days	Fri 17/02/06
Project Plan 2.0	4 days	Sun 19/02/06
Architectural Design 1.0	31 days	Wed 22/02/06
Formal Specification 1.0	15 days	Mon 27/02/06
Test Plan 1.0	17 days	Mon 27/02/06
Inspection Checklist 1.0	7 days	Mon 13/03/06
Prototype 2.0	40 days	Thu 23/02/06
Presentation 2.0	4 days	Sat 01/04/06
− Phase three - Production Phase	**41 days**	**Wed 05/04/06**
Component Design	16 days	Wed 05/04/06
Application Code	28 days	Fri 07/04/06
− Testing	**16 days**	**Tue 18/04/06**
Functional Testing	12 days	Tue 18/04/06
Testing Eveluation	7 days	Thu 27/04/06
− Documentation	**20 days**	**Fri 21/04/06**
Assessment Evaluation	10 days	Fri 21/04/06
User Manual	12 days	Fri 21/04/06
Project Evaluation	10 days	Sun 30/04/06
References	2 days	Tue 09/05/06
Presentation 3.0	5 days	Thu 11/05/06

We can see in the Gantt chart my initial planning. Phase 1 was supposed to be the key phase of development where I was supposed to define all the requirements. This phase was indeed really important and it took me a lot of effort. It is really hard to start a project from scratch and be the only one that put the requirements. Actually, this phase got delayed around 10 days. We can also see how idealistic this plan was. It assumed that the requirements were going to be correct and the architecture (phase 2) and implementation (phase 3) were going to be trivial.

A correction of this initial planning is shown below:



Task Name	Duration
- Phase One - Inception Phase	112 days
Vision Document 1.0	32 days
Project Plan 1.0	25 days
Software Quality Assurance 1.0	25 days
1st Prototype	65 days
Presentation 1.0	4 days
- Phase Two - Elaboration Phase	50 days
Vision Document 2.0	6 days
Project Plan 2.0	5 days
Architectural Design 1.0	40 days
Formal Specification 1.0	25 days
Test Plan 1.0	20 days
Inspection Checklist 1.0	15 days
Presentation 2.0	6 days
- Phase three - Production Phase	95 days
Formal Specification	10 days
Component Design	35 days
Application Code	72 days
Presentation 3.0	5 days
- Testing	25 days
Functional Testing	17 days
Testing Eveluation	10 days
- Documentation	27 days
Assessment Evaluation	7 days
User Manual	12 days
Project Evaluation	9 days
References	2 days

This plan is more realistic. The project in this case was planned to end in the middle-end of July. Basically, I realized that using new technologies and tools was too risky and implementation was going to be harder. I also don't have enough experience to develop software correctly. So, the implementation time almost doubled. It actually took a little bit longer because of the usual final adjustments.

Summarizing, the first plan was too unrealistic because I didn't take into account the fact of suing new technologies and tools. But the planning shows that if you take enough time analyzing the requirements you can avoid many errors in the other phases. You can see how it didn't take me that long to develop the architecture design from the requirements.

The implementation phase would have been easier if I had more experience and if I had used other technologies. For example, if I had used Struts instead of JSF it would have taken less time since I already know that technology. But one of the goals of this project was to test these new technologies.

6. Project Metrics

I used the open source program *SourceMonitor* to get the software metric of this project. In the next table I summarized the results:

	Model	**View Backing**	**View JSP**	**Total View**	**Total Java**	**TOTAL**
LOC	5,681	9,268	1,517	10,785	14,949	**16,466**
Files	40	47	43	90	87	130
Statements	2,545	4,970	30	5,000	7,515	7,545

The model is composed of 40 Java classes that correspond to the model façade and the EJBs. There are 5,681 LOC in this module.

The *view.backing* package contents the backing beans and the supported classes such *ModelConvert*; all of them are java classes. There are 43 backing beans (same number than JSP pages) and 4 standard Java classes (*ModelConvert* and the three *SessionCheck* classes). There are 9,268 LOC in this package. This large amount of LOC where mostly automated generated by JDeveloper, and they are mostly the UI components and their corresponding getters and setters. I just added the necessary logic to the getters/setters and added few other methods to each bean.

The View JSP column shows the JSP pages. It doesn't count the configuration files or libraries. The program shows that it has 30 statements but actually it doesn't have any, JSP pages just use tags, all the logic is in the backing beans.

The Total View column represents the JSP pages plus the backing beans together. The Total Java column represents the backing beans plus the model, which are all the java classes. Finally, the Total column includes all the LOC including the Java classes and the JSP pages of the whole project.

7. Problems Encountered

In this section I'll summarize the problems encountered during the development of the KSU Student Portal that were explained in the previous sections:

1. Phase 1: Requirements
   a. Several difficulties building prototype. Especially with the JSF.
   b. Difficulties setting the requirements. Some of them were missing and discovered during implementation.
   c. Use case model was changed many times due to the problems with the requirements.

2. Phase 2: Architecture
   a. Use case model was wrong and had to be changed.
   b. Building the sequence diagram became a tedious job.
   c. Initial data model was wrong. I had to make several changes to get the final design. The data relations were not trivial and the creation of the relational schema was a difficult task.

3. Phase 3: Implementation
   a. Problems with the EJB configuration, especially setting the relations and its types (fetch mode, cascade type…).
   b. Filter algorithm.
   c. Controlling the session. I had to create the structure explaining in the component design to solve this problem.
   d. JDeveloper bugs.

8. Future Work

There are several improvements that can be performed in the application that, due to time restrictions, they were not implemented. First of all, a nice feature will be show the results of lists only 10 or 20 at a time letting the user choose how many result he/she wants per page. To perform these ADF table UI components could be used along with the implementation of the page by page iterator pattern.

Another feature that will be easy to implement is the improvement of the password functionalities. It will be good if the system only lets the user try the password 3 or 5 times to avoid hackers use tools to get the user's password. Also, a security question should be asked in the user's profile that will be used to get the password in case the user forgets it.

Of course, implementing the messaging system will be next feature. Also, more features can be easily added to the system since it is lousy coupled. For example some attributes could be added to some entities such adding a title to the user blog (not only the entry) or adding a comment to a link. These changes will be really easy to perform.

The design could be improved a little bit. I could have taken more advantage of an Object Oriented approach and create more hierarchies to reuse code. We also have the trade-off between code reuse and complexity. For example, we could have a general entity class with the attribute ID that can be used to find object by ID independently is those are events or articles. This will create a more complex architecture and since I don't perform many searches by ID's I decided not to use this architecture.

We could also improve the security by adding a stronger encryption algorithm on the server side.

Finally, the view can be improved by a Web designer. Flash elements can be added along with more images. AJAX can be used in some occasions such forms or to reload only parts of the Web page.

9. Conclusions

In my opinion, I think I've learned more than I expected doing this project. I applied a whole software process and I tested new technologies that made me put a lot effort to get the final results.

It is obvious, that the final product is not perfect; although it works fine I would like to achieve a "perfect" design that has all quality attributes. But looking at the final project I realized about all my mistakes and I'm sure my next design will be better and I will continue improving until the day that hopefully I will become a software architect which is my final goal. I think experience is the key to become a successful developer, I know my first design will have flaws but I hope I have time to analyze them and learn from them. I guess is very common that due to time restrictions the design will not be as expected since clients just want the application "working". But I hope I don't commit the mistake of leaving the application as "it is" and have the same mistakes again.

The next step is to enter to work in a company where I can learn the real business processes. I still have many things to learn that I will only learn working in a real project with more experienced workers. I'm eager to learn everything that takes to become a good developer and hopefully one day start my own business.