Technical Report

Co-op Evaluation System

Senior Project 2014-2015

Team Members:

Tyler Geery
Maddison Hickson
Casey Klimkowsky
Emma Nelson

Faculty Coach:

Samuel Malachowsky

Project Sponsors:

Jim Bondi (OCECS)
Kim Sowers (ITS)

Table of Contents

1 Project Overview

1.1 Background

The purpose of the Co-op Evaluation System (CES) is to allow students to provide feedback on their most recent co-op, and for employers to provide feedback on a student's performance during their most recent co-op. Additionally, the system is used by faculty to approve or fail a student's co-op, and is also used by OCSCE to gather data on students' co-ops.

1.2 Motivation

The purpose of this project is to re-engineer the Co-op Evaluation System in order to leverage newer web technologies while also improving performance and user interaction. The current system uses outdated, under-documented technology, which makes it difficult to maintain. Furthermore, the random errors that occur do not give users confidence that their information was submitted properly. Significant improvements to the user interface needed to be made, but the existing database structures were used as a reference for modifications.

1.3 Scope

One of our primary goals was that by the conclusion of this project, we will, at a minimum, have supplied OCSCE and ITS with a product that is functionally equivalent to the existing system, with fewer performance, reliability, and usability issues. Time permitting, we hoped to implement a small number of enhancements, as defined by our project sponsors, as well.

However, we knew that our desired results were optimistic. Therefore, we planned to design and implement the system with extensibility and maintainability in mind, so that in the future, other developers may implement any additional features that we were unable to implement during the duration of this project.

Once we realized that it would not be feasible for us to deliver a functionally equivalent system by the end of the academic year, we decided the resulting system would be judged successful if we were able to implement the end-to-end workflow of evaluating a student's co-op, and complete any number of administrative features in the given time frame.

2 Basic Requirements

2.1 User Classes and Characteristics

2.1.1 Student

Students are defined as co-op students from RIT. They primarily use the system to complete a work report evaluating their co-op experience for their most recent co-op block, and therefore use the system once per co-op block.

2.1.2 Employer

Employers are the managers or designated reviewer at the student's workplace. The sole interaction an employer has with the system is to complete an evaluation on a student's co-op performance. Employers use the system one time per student per co-op block.

2.1.3 Evaluator

Evaluators are usually faculty or advisors in the student's department, who review the student's and employer's survey responses to determine the student's grade (S or F), and use the system one time per student being evaluated.

2.1.4 Administrator

Administrators are normally a member of OCSCE and are the user class that uses the system most frequently and for the most reasons. Their main use cases are to create and assign new form and gather statistical data from survey responses for ABET accreditation.

2.2 High-Level System Requirements

2.2.1 Student

The system shall allow students to complete a work report for their most recent co-op block, review their previously submitted work reports, view an employer's evaluation of a co-op block, and see whether a co-op block has been approved or rejected.

2.2.2 Employer

The system shall allow employers to complete an evaluation on a student's co-op performance for each student they manage. Additionally, the system shall allow employers authenticate with the system through a non-Shibboleth login screen.

2.2.3 Evaluator

The system shall allow evaluators to review the student's and employer's survey responses to determine the student's grade (S or F), search for and view any submitted or pending evaluations within their department, and view and edit email notifications for their department.

2.2.4 Administrator

The system shall allow administrators to search for and view any submitted or pending evaluations, generate reports, manage emails (schedule email notifications, edit content of student emails, and edit content of employer emails), create and delete users (OCSCE staff and faculty), manage colleges, manage departments, and manage forms (create new forms, delete or modify existing forms, and assign a form to a department). There are several other smaller tasks that administrators are able to perform as well, such as initializing the next term and updating the status of a form or group of forms.

2.2.2 System Requirements

Form Administration

Student and employer evaluations shall have the following progress statuses: Submitted, Saved, Open, Pending, Manually Completed, and Archived. Student evaluations shall have the following evaluation approval statuses: Submitted, Approved, and Rejected. Refer to the section on Evaluation Status States for more detail.

Submissions

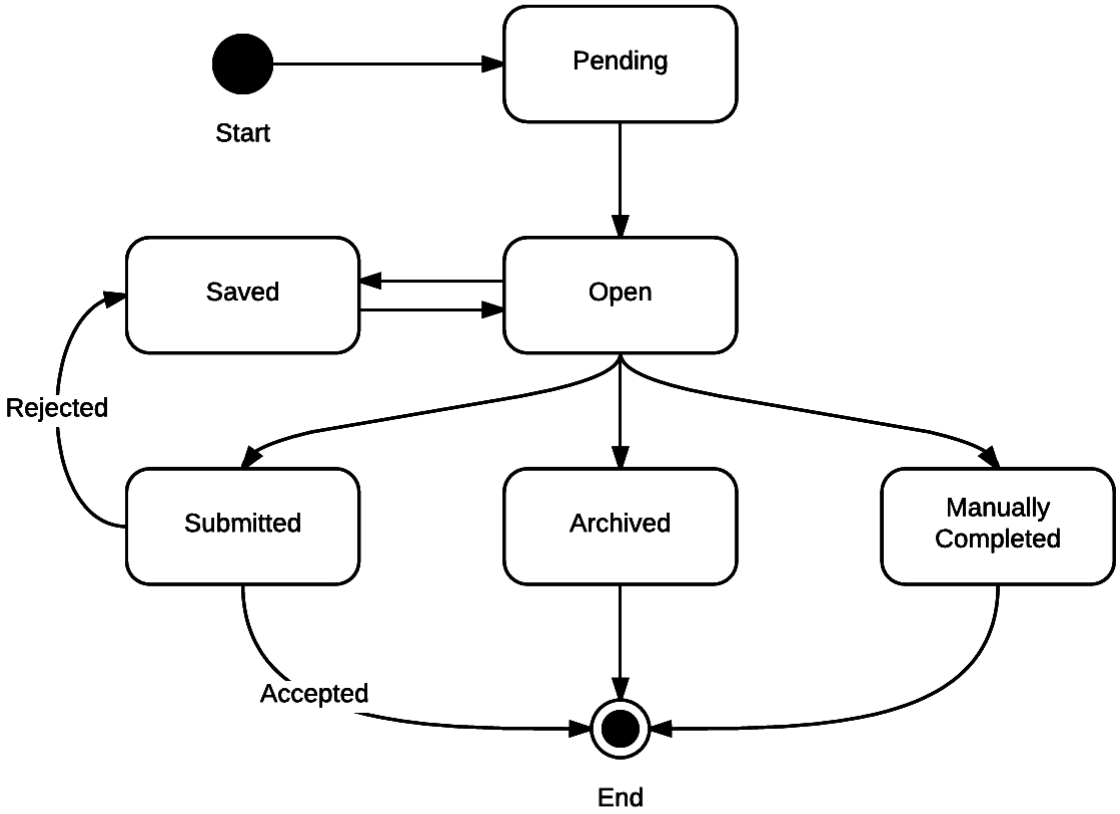
The system shall be able to search forms based on student last name, student first name, student ID, company name, term, department, advisor's RIT Computer Account username, student progress status, evaluation progress status, and employer evaluation status.

Evaluations

The system shall be able to validate an evaluation for completeness. The system shall be able to validate an evaluation for correctness (e.g. email validation) the client side.

Evaluation Status States

The following diagram details the possible states an evaluation may have within the system:



Evaluations, both employer and student, will start out as Pending. In the Pending state, the evaluation appears in the system, but cannot be filled out yet.

Three weeks from the end of the term, all evaluations will be changed from Pending to Open. An evaluation in the Open state can be filled out by the employer or student that it is assigned to. An Open evaluation can be changed to Saved, Submitted, Manually Completed, or Archived.

If the user saved the evaluation, it is changed to the Saved state. In this state, the user can open the evaluation and continue filling out answers. They can continue saving the evaluation until they are finished. The evaluation can be changed to Submitted, Manually Completed, or Archived state from the Saved state.

The Submitted state is reached by submitting the form from either the Open state or the Saved state. At this point, the only change that can be made to the evaluation is by an evaluation approval change. If the evaluation is approved, nothing else will happen. If the evaluation is rejected, the evaluation will go back to the Saved state so that the student may modify their answers and re-submit.

There are two other states that an evaluation can end up in: Manually Completed and Archived. If the evaluation is completed in some way other than the normal process, an evaluation can be changed to the Manually Completed state. If the evaluation will never be completed, and the user does not want to receive messages telling them to fill out the

evaluation, the evaluation can be marked as Archived. The Archived state was known as the Past Pending state in the previous version of the CES.

Reports (Out of Scope)

The system shall generate reports based on a user-defined selection of submissions and statistics. The system shall produce statistics on questions that have numeric answers and qualitative answers (“Comments Only”). The system should provide an easy way to select items to be included or excluded from the report. The system should provide the ability to run reports across multiple year-levels (i.e. all undergraduate students) or by academic year or by form. The system should be able to run reports on qualitative data and should display the qualitative questions on forms as selectable options when choosing the report settings.

Emails (Out of Scope)

The system shall be able to send generated email notifications to students and evaluators manually and automatically. The system shall be able to generate evaluation notifications to all employers and students a configurable number of weeks before the student’s end date. The system shall be able to generate a confirmation email to students and employers. The system shall send a notification email to a student when their evaluation has been rejected. The system shall send an email notification to students confirming their supervisor, start date, and end date. The system shall send an email notification to evaluators when their students submit a work report.

Users

The system shall use the user information captured from existing OCSCE database to provide authorization for employers. The system shall be able to automatically initialize the next term for students and employers based on the RIT academic calendar and update the status of submissions when supplied with a new status (manually or automatically). The system shall use Shibboleth to authorize students, faculty, and OCSCE staff.

2.3 Inputs

2.3.1 User Input

The system accepts input from all four user classes, although the type of input will vary by user type. Students and employers simply input answers to form questions, and evaluators can only accept or reject evaluations. Administrators enter the largest and most diverse input into the system, as they can perform the most functions.

2.3.2 Placement Data

The CES gets its co-op placement information from student input into the Co-op Notification Form. The data includes the student’s name, major, terms of employment, start and end dates, position title, and company information.

2.3.3 Shibboleth

The system uses Shibboleth to authenticate student, administrator, and faculty users (employers use a separate custom-built authentication). From Shibboleth, the system

receives the user's UID, RIT Computer Account, email address, first name, and last name, as well as the type of user.

2.4 Outputs

Note: The following outputs are requirements of the system, but we did not implement them during our time spent working on this project due to time constraints.

2.4.1 Emails

The system sends emails to both students and employers. To students, the system sends a notification email when their work report is now open and can be completed, a confirmation email when their work report has been submitted successfully, and, if the student's co-op evaluation is rejected, an email notifying them of this rejection.

2.4.2 Reports

The system will use a third-party tool for reporting. Although we performed some research into potential tools, we did not decide on a tool. However, whichever tool is chosen, it must support creating the SQL view used to generate a report, and exporting report data in CSV format, as these are the two outputs from reporting that are defined in our SRS.

3 Constraints

3.1 Resource Limitations

3.1.1 Time

The available calendar time for the project was the length of two academic semesters, a total of 33 weeks, non-inclusive of breaks in the institute calendar. A limited amount of time was spent by the development team during Intersession and Spring Break, but this time was not slated as official time in our project schedule.

3.1.2 Money

Although we thought that we might need to ask our project sponsor for software licenses (e.g. development software for our own use, or reporting software for our implementation), there ended up being no need. The only monetary investment in the project was the help we received from Anu, a co-op in OCSCE.

3.1.3 Personnel

The personnel for this project was limited to the four development team members. For the second half of the project, we were able to have Anu help us with our development. Ultimately, she primarily helped us write unit tests for our Java code, and she helped to implement some new features as well.

3.2 Design Constraints

The only design constraint we had to work within was when designing the authentication service we had to account for Shibboleth integration. ITS had a very specific way in which Shibboleth needed to be integrated, so that required us to design our authentication filters in a specific way that allowed for the application to leverage the already existing Shibboleth packages. The Shibboleth packages required us to implement our version of specific classes such as the SingleSignOnUtil class.

3.3 Implementation and Technical Constraints

The system had to comply with the development guidelines provided to us by ITS, as defined by the EWA Student Development Guidelines wiki page. At a high level, these guidelines include approved application frameworks, build tools, application server technologies, database standards, and several other technological standards.

Specifically, we were required to use Java on our back-end. ITS had already approved Spring as a framework we could use, but we were required to have Hibernate approved. On the front-end, Sass was already an approved framework as well as AngularJS, which was recommended as our client-side MVC framework. For our build tool, Ant and Maven were our two choices, and we decided to use Maven. Lastly, we were required to use an Oracle database to store our application data. There were several other less substantial guidelines that we were required to work within as well, all of which can be found on the wiki page.

The system was also required to comply with the RIT Web standards document, which defines the standards for RIT-related websites in regard to language, graphics, and navigational architecture. However, we found this document to be fairly outdated, and chose to use RIT websites that had been recently updated as the primary inspiration for our user interface instead.

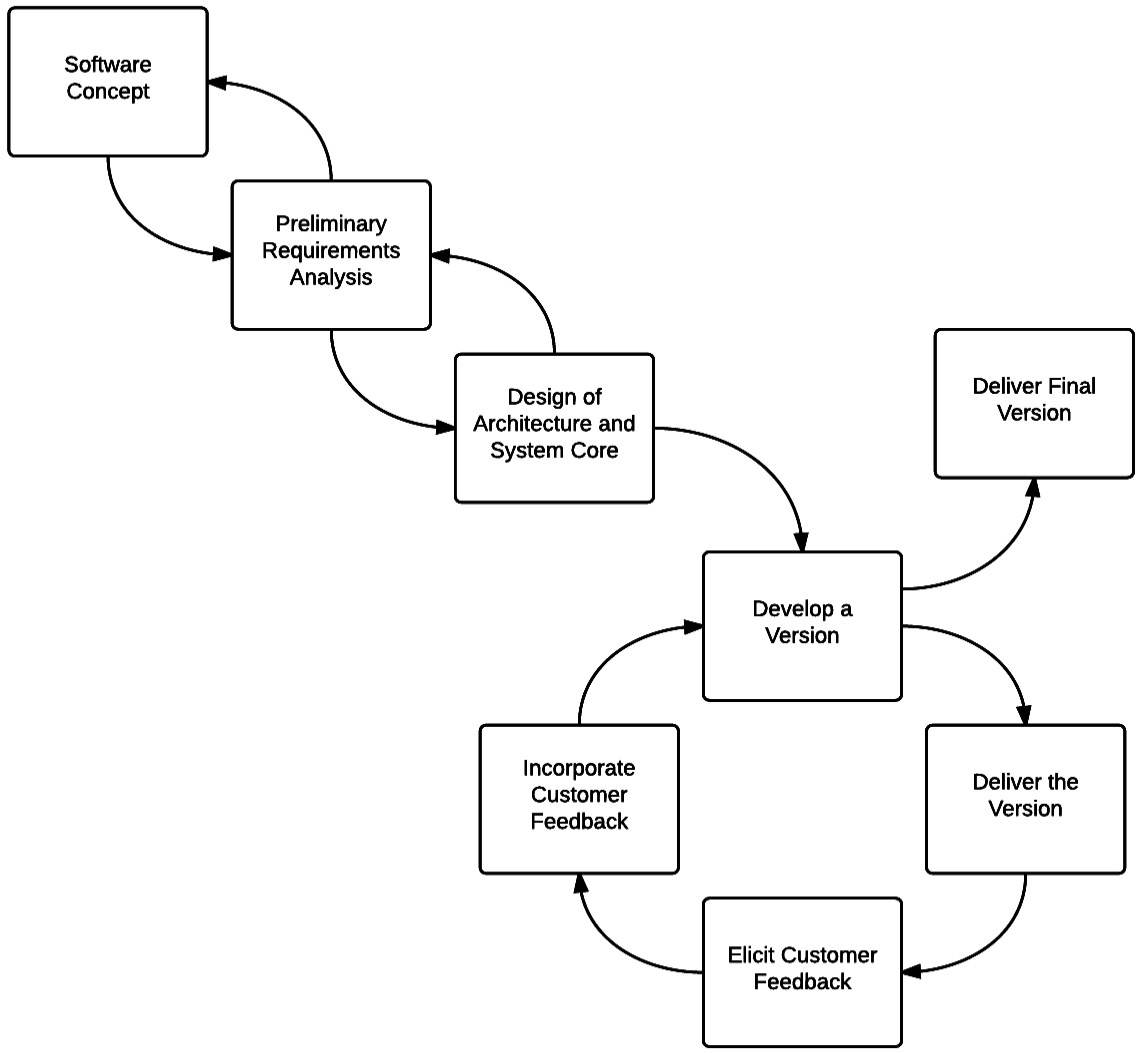
3.4 Delivery and Maintenance Constraints

We were required to deploy our system on a Tomcat server. The requirements for the server environment structure were given to us by ITS. We were also required to deploy identical builds to three separate environments: DEV, TEST, and PROD.

4 Development Process

4.1 Project Lifecycle

We followed the Evolutionary Delivery process methodology for development of the system, which is depicted in the following diagram:



This process was not approved or mandated by the project sponsor; however, we did have our process approved by our faculty coach.

Our process had regular feedback built into it. At the end of each step in the waterfall part of the methodology, we turned over the resulting artifacts to the sponsors for review and feedback then revised accordingly. At the end of each development cycle, we demoed our progress for our sponsors, customer, and coach. We also used weekly emails to share team status and any updates.

4.2 Project Organization

4.2.1 Roles and Responsibilities

The following table details what roles our team used, who filled which roles, and what those roles entailed:

Name	Role	Responsibility
Emma Nelson	Team Coordinator	Coordinates overall team process and is the end person responsible for all project documentation.
Maddison Hickson	Webmaster Development Coordinator	Maintains the project website and adds new content to the website as necessary. Tracks development progress and is the point person for keeping feature development on schedule.
Casey Klimkowsky	Communications Coordinator	Serves as a communication point between the development team and project sponsors. Records meeting minutes during all meetings, including any major team decisions and action items.
Tyler Geery	Testing Coordinator	Tracks testing progress by comparing the work done to the test plan and ensures that any gaps are addressed.
Anu Sharma	Contractor	Development and testing support for the second half of the project.

4.2.2 Interaction with External Organizations

We worked closely with the ITS Application Development and Operations staff to determine approved technologies for the project. ITS also provided support breaking down technical roadblocks and maintaining our servers and test user accounts. Furthermore, at the conclusion of the project, we performed a formal handoff of the product to the ITS Support staff.

OCSCE is the other organization that we worked with for the duration of the project. The requirements for the final system were given to us by OCSCE, as the goal of this project is to replace their existing Co-op Evaluation System.

5 Project Schedule: Planned and Actual

5.1 Schedule Development

We broke the project into major subsections at a very high-level and assigned them to what we planned to be three week cycles. The key activities we identified included requirements analysis, architecture, system design, and development and testing cycles. We broke development and testing into six cycles, as follows Environment Configuration and Student Dashboard (Cycle 1), Student Role (Cycle 2), Employer Role (Cycle 3), Admin Role Part 1 and Evaluator Role (Cycle 4), Admin Role Part 2 (Cycle 5), and Finalization and Documentation (Cycle 6). What exact features were included in each milestone were determined at the start of each cycle in order to ensure that the scope would fit the schedule set.

5.2 Schedule Refinement and Adaptation

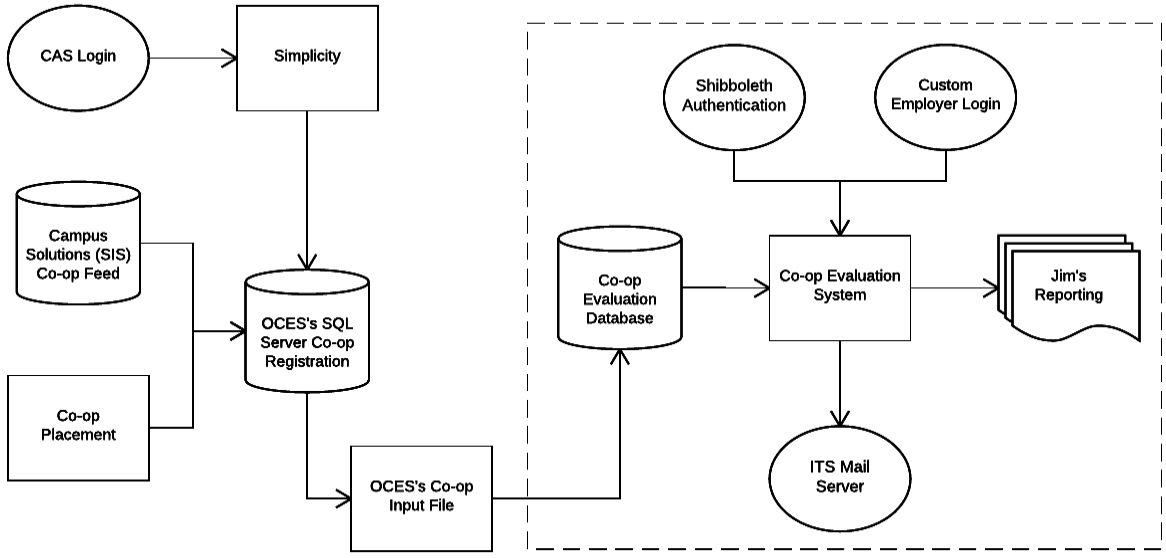
The schedule was continually refined at the start of each stage in the process. Before starting the development work of each stage, the team took time to establish what we wanted to accomplish in the given stage and estimate the time each task was expected to take, thus the exact schedule was not complete until near the end of the project. When we had schedule slips, we adjusted the plan for the next cycle to accommodate adding in the missed requirements. We always adjusted the scope to meet the schedule.

5.3 Planned vs. Actual Schedules

The actual schedule ran over compared to the planned at various points. Design ran over into Cycle 1 because we could not finish the mock-ups before Christmas as planned since they were much more involved than expected. Cycle 2 ran over into Cycle 3; however, as their feature sets were very similar, we caught up by the scheduled completion date for Cycle 3. The Admin Role parts of Cycle 4 ran into Cycle 5, but again we were able to catch up in time. In this case, it was a problem of prioritization of features and dependent tasks being more complex than estimated. In the end, we met the final deadline with more-or-less everything we had planned to deliver.

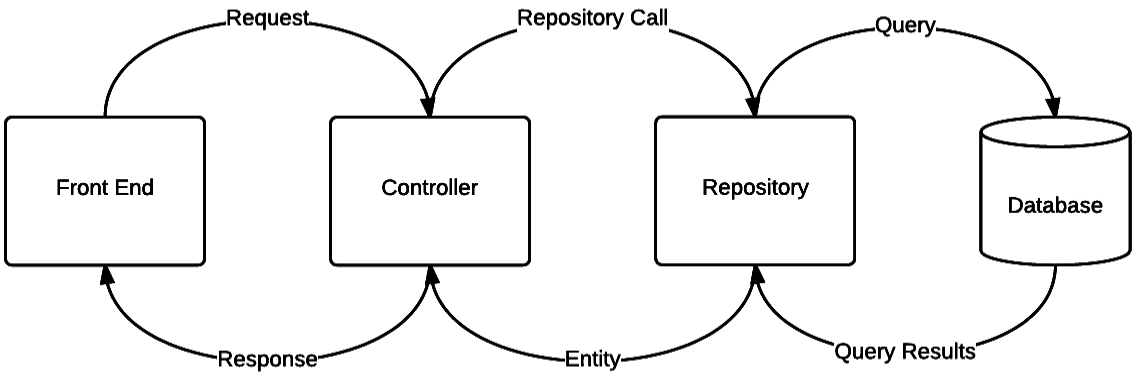
6 System Design

The below diagram shows the basic flow of data into and out of the system at a high level. Our system and direct interfaces are represented inside of the dotted line, with the outside entities depicting how data is created and imported into our system. In this diagram, the “Co-op Evaluation Database” represents the relational database used to store system information. The box labeled “Co-op Evaluation System” represents the core functionality of the system, which is broken down as a series of components. More information will be provided in the below sections.



6.1 Data Design

For our database design, a lot of inspiration was pulled from a previous Senior Project attempt. Our database contains five major pieces of information: user data, college data, evaluation data, form data, and email data. Slight refactoring was needed after creating our initial design in order to add missing fields that were necessary. Also, the addition of createDate, createBy, modDate, and modBy fields were required by ITS in order to help with monitoring of the database.

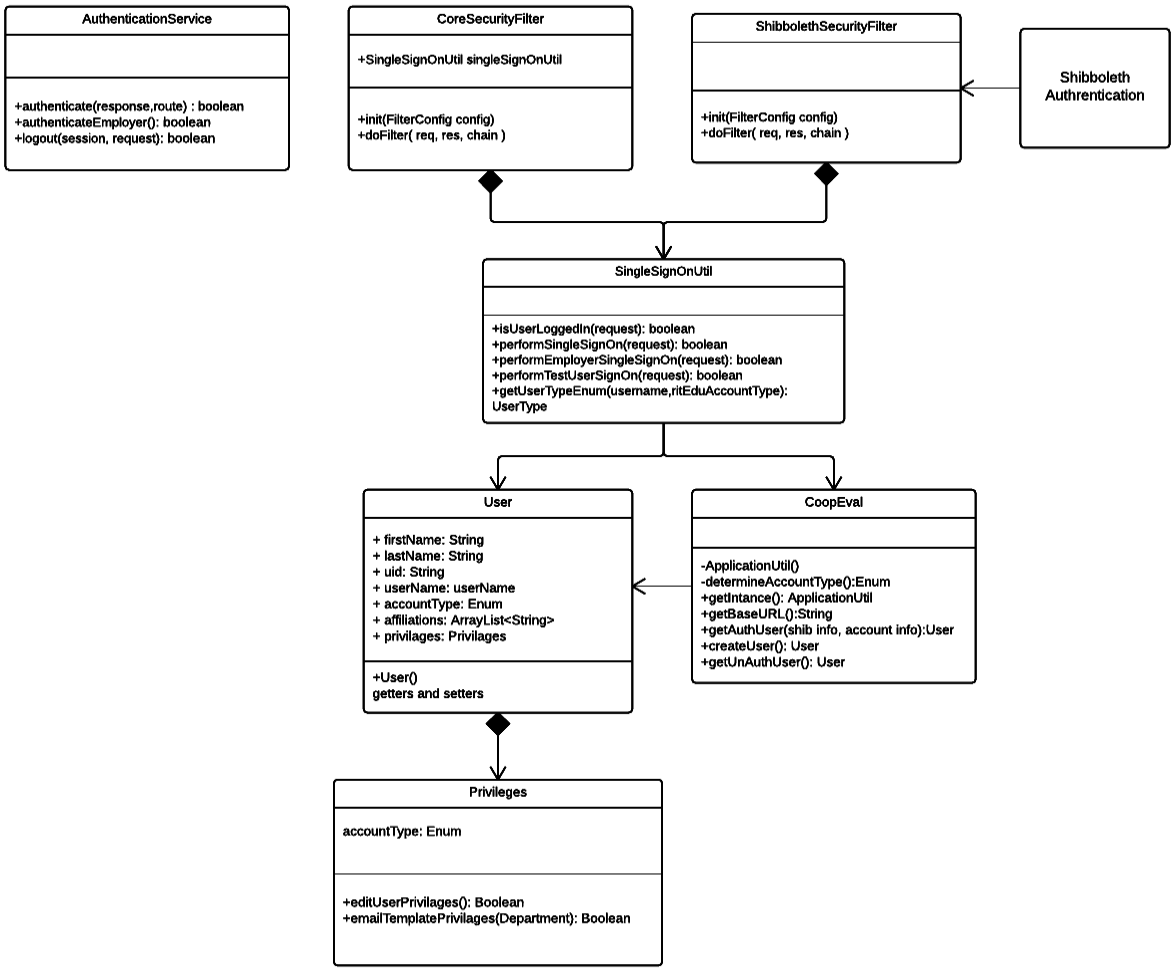


In order to get data in our application, we used Spring repositories to execute queries. When executing queries, these Spring repositories take the information from the database and map that data to JPA entities. These entities are then returned to the controllers to be used by the rest of the application.

6.2 Component Design

6.2.1 Authentication Service

Using Shibboleth and the system's EmployerUser table, this service will identify if a user is within the system and, if so, log them in. At the same time, it will also identify what the user's privileges are and act accordingly (e.g. if the user is a student, administrative content will not be displayed).



User

Our system will take information given to us by Shibboleth or our employer authentication system and create a specific User object that can be accessed by the entire system. The system will use this information to specify an enumeration to that user, which will define what that user's privileges are. This User object is stored in the session.

Privileges

As stated above, our system will use an enum to associate each user type to each individual user. This enum will define the user type; for example, "1" could represent an employer, "2" could represent an administrator of the system, and so on. The User object contains a Privileges class, which returns true or throw an unauthorized error for user-specific privileges that are dependent on the user type (or enum).

AuthenticationService

The AuthenticationService contains endpoints that perform login for Shibboleth or employers. The login methods do nothing themselves but are a way for the filters to check for the specific login URI's and perform the login functionality.

CoreSecurityFilter

The CoreSecurityFilter is hit on every request to any HTML or controller endpoint. If the request is attempting to hit AuthenticationService's employer login endpoint, the filter will call SingleSignOnUtils method for logging in employer users. It also makes sure the request isn't whitelisted. For any other request the session is checked to see if the a the user has been authenticated. If the user has been authenticated then the filter lets the request go through, if the user is not authenticated an unauthorized error is send back.

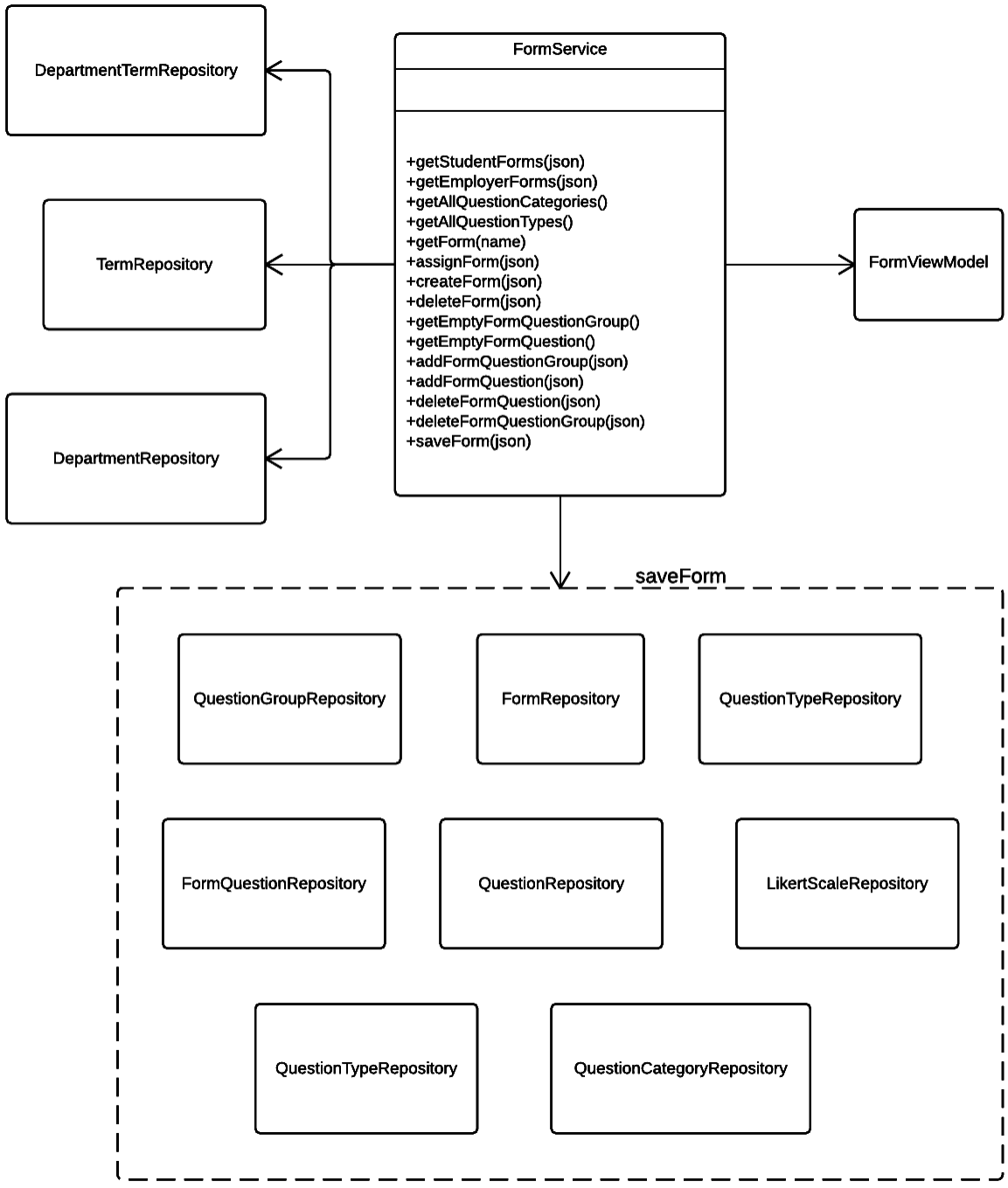
ShibbolethSecurityFilter

The ShibbolethSecurityFilter is only called when the Shibboleth login endpoint is called in AuthenticationService. This filter takes in the Shibboleth header information and verifies that the User has access to the system.

SingleSignOnUtil

This class is in charge of either taken in the Shibboleth header information and turning it into user that is stored in session or taking the login credentials of an employer and authenticating the user. The SingleSignOnUtil uses the department and administrator user tables to verify whether a Shibboleth user has access to the system and also uses the employer table to check if the employer's login credentials are correct. The SingleSignOnUtil can also check if someone is logged into the system by checking if a user is stored in session. SingleSignOnUtil is also in charge of figuring out which UserType enum to assign a specific user.

6.2.2 Form Service



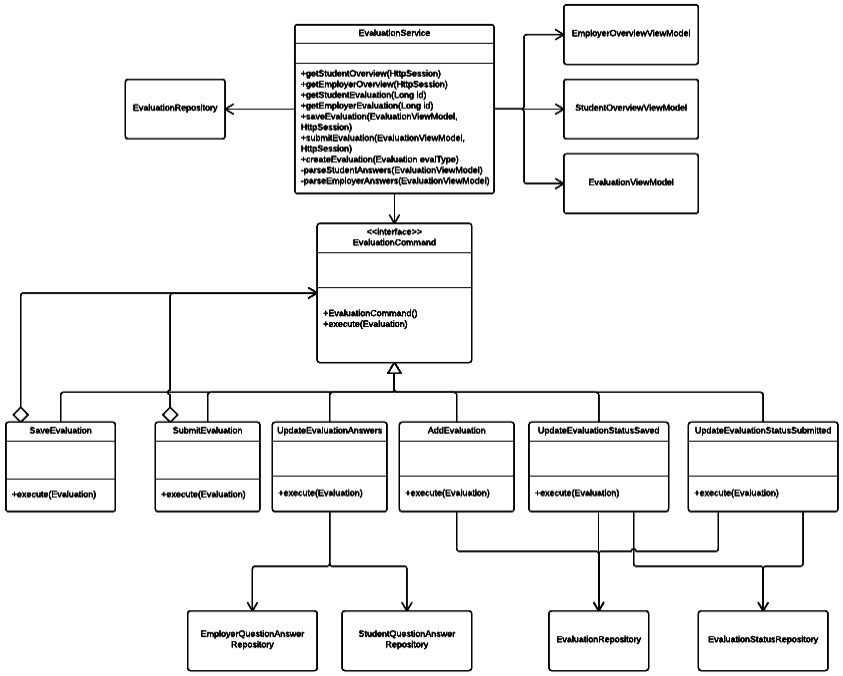
FormService

This controller is in charge of taking in requests for actions related to FormService. The two major components of FormService are editing forms and assigning forms to a department. For form editing the approach that was taken that instead of persisting every time a change was

made the methods actually alter the FormViewModel that is stored in the session. Once the save is called the FormViewModel is compared with the form in the database in order to use the repositories to persist the necessary changes.

6.2.3 Evaluation Service

EvaluationService is responsible for handling all evaluations that are being tracked by the system. An evaluation is either completed or approved. For this service, we made use of the Command Pattern.



EvaluationCommand

This class contains an execute method, which handles an Evaluation object and passes it to a designated class that knows what to do with it.

AddEvaluation

A “receiver” class, which handles adding an Evaluation to the system by persisting it to the database.

UpdateEvaluationAnswers

A “receiver” class, which handles updating question answers in the system by persisting it to the database.

UpdateEvaluationStatusSaved

A “receiver” class, which handles updating an Evaluation object’s status to saved in the system by persisting the changes to the database.

UpdateEvaluationStatusSubmitted

A “receiver” class, which handles updating an Evaluation object’s status to submitted in the system by persisting the changes to the database.

7 Process and Product Metrics

The development team tracked three metrics to aid in measuring the quality of the project: requirements volatility (churn), SUS, backlog management index, and code coverage as well as a number of process metrics around time and effort of the team.

7.1 Time Tracking Metrics

7.1.1 Description and Rationale

The Activity Tracker helped the team track multiple metrics on time and effort. These metrics included total hours per person, average weekly hours per person, total hours by the team, and average weekly hours by the team. The percent completion of estimated time and actual time was also tracked for each week. Tracking the type of task along with the time allowed us to generate statistics on how much time and what percentage of time was spent on each type of activity: Preparation, Coordination, Requirements, Documentation, Implementation, and Testing.

The rationale for tracking all these measures and metrics was that it would allow the team to see who has contributed what to each task, week, and semester. It helped us hold each other accountable for the time we were expected to put in each week.

7.1.2 Results

The team put in over 1,300 hours total. Every member averaged over eight hours a week, and as a team, we averaged about 39 hours a week. Only about 30% of project time was spent on implementation; over half was on more process-based tasks, such as requirements elicitation, documentation, and working sessions. We feel like our hours show our dedication to the project, and the breakdown of where the time was spent really shows our emphasis on good software engineering practices. The goal of this project was not just a working system, it was to have a well-documented, working system. Our focus on documentation supported this goal.

The team obviously put in the number of hours required by the department and then some. We all worked really hard; however, when we looked at individual team member's numbers, there was a fairly big difference in total time spent. There was a difference of about 70 hours between the highest and lowest totals. This demonstrated while a lot of effort was put forth, it was not equal among all members. It would have been preferred for the range in time spent to be smaller, making it more of an equal effort.

7.2 Requirements Churn

7.2.1 Description and Rationale

Tracking requirements volatility gave the development team insight into how much the requirements changed over time, starting from the time of completion of the initial version of

the requirement documents. This is particularly important when it came to recognizing when, how often, and how many requirements were added or dramatically changed.

We recognized that we had a project with a large, complex scope that needed to be carefully managed, and we needed a way to track the changes occurring. We only tracked it the first semester because we stopped allowing for scope changes once development started.

7.2.2 Results

There were 19 requirements changes, and our churn rate was 2.7 over the seven weeks between completion of the initial Software Requirements Specification and the end of the fall semester. We stopped allowing for changes after that point, so we stopped tracking the metric. Nineteen requirements ends up being about 20% of our itemized requirements, thus we acknowledge that our churn rate was quite high. It forced us to keep in mind potential changes as we went through the design phases.

The development team did a good job keeping changes in mind and designed for it; however, we did not plan for it in the initial schedule development. We agreed to a scope that kept growing and changing, making it impossible to deliver what we originally promised in the early fall. We had to pair down the scope, in the end, in order to deliver a working system on time.

7.3 Software Usability Scale (SUS)

7.3.1 Description and Rationale

The SUS is a set of 10 Likert questions that is a quick reliable way to measure usability of the product and each user role. A score of 68 or above is considered better than average.

The team wanted a metric that would demonstrate how well we were doing in the usability department. It was a core factor in our design, so it was important to measure it in some meaningful way.

7.3.2 Results

We achieved a score of 86 for Student, 78 for Employer, 87 for Department, and 81 for Administrator. Our average score was an 82. These results demonstrate that we met, and exceeded, our usability goals as we were significantly above the threshold of 68 in every area.

These metrics show the team that mocking up the screens for the whole system was worth the time and effort, as we were able to test our design before writing a single line of code and make adjustments very early to account for user needs. We had a good idea of what the pain points were and reasonable ideas on how to fix them. Our usability testers agreed to such an extent that we surpassed our goals.

7.4 Backlog Management Index (BMI)

7.4.1 Description and Rationale

The backlog management index is the number of bugs introduced (reported) compared to the number of bugs resolved. This metric was actively tracked starting at the beginning of the development phase through the very end of the project.

Similar to requirements churn, this metric gave the development team insight into how much our project was changing and was there to throw a warning flag if we started slacking on good software engineering development practices. We knew that if our BMI was below a certain point, we needed to step away from feature development in order to correct some bugs as we wanted to stay above a certain threshold.

7.4.2 Results

Of the 36 bugs reported, we resolved 34, giving us a 0.94 backlog management index. Either we were really on top of fixing bugs or we were not very good at reporting them accurately. Probably was a mix of both.

The metric, regardless of the interpretation, indicates that we had a reasonable level of quality control in place. We were aware of bugs in our system and made an effort to report them. At the end of each cycle, we always spent a day or two knocking out some bugs before demoing the system to our sponsors. We probably could have had a better process in place for reporting bugs more accurately and made more of an effort to do so regularly.

7.5 Code Coverage

7.5.1 Description and Rationale

Code coverage is the percentage of lines of product code covered by unit tests. In our case, we measured this only over the business logic in both the front-end JavaScript and back-end Java. This scope reduction was done since a large portion of our source is entities and other data containers on which unit testing is not useful.

The idea behind this metric was to ensure that we were generating a reasonable number of tests. It also forced us to run our test suite at least once a week, turning up any regressions in source code or tests on a regular basis.

7.5.2 Results

We managed 69% coverage of our reduced-scope Java and 15% coverage of all JavaScript. The high number of JUnit tests was the result of having an OCSCE intern on our team who focused a lot of her time on testing. JavaScript was a lot lower because the core JavaScript developers focused on feature-development, and the intern did not know the language.

This metric clearly shows that we probably should have spent more time writing unit tests; however, it came down to a matter of priorities. We had limited time and resources so we had to choose between more functionality and more tests. Functionality won the debate every time since the system was much more complex than we originally estimated.

8 Product State at Time of Delivery

8.1 Current State

At the conclusion of the project, we were able to complete 36 of our 87 requirements, or 41.38%. Of our in-scope requirements, 3 requirements were left in progress, and three requirements had not been started. We currently have 2 bugs open on GitHub Issues.

The entire employer and student roles have been completed, with the exception of form validation. Although we were able to complete numerous administrative features, the administrator user role is the one user role that is far from complete. The evaluator role is still missing the email features that it shares with the administrator role.

Throughout the entire system, we display some error messages when the user performs an invalid action or an error occurs within the system. However, more specific messages need to be implemented to account for all possible errors, and also to give more detailed feedback to users.

8.2 Missing Features

The two largest feature sets that were left out of our implementation are the administrative tasks of reporting and emailing. Although we had originally intended to complete these feature sets by the conclusion of the project, we ultimately had to define them as out of scope as the scope of our project was too large for the timeframe we were given. Forty-five of our requirements are defined as out of scope in our SRS.

The three in-scope requirements that remained in progress are updating the status of evaluations, assigning forms to departments, and validating evaluations for correctness on the client side.

Lastly, the three in-scope requirements that remained not started are automatically changing the progress status of evaluations from Pending to Open, automatically initializing the next term based on the RIT academic calendar, and transferring department user privileges to another department user.

8.3 Unplanned Features

No unplanned features were added before the conclusion of the project. We performed extensive requirements elicitation towards the beginning of the project to ensure that no unplanned features would be added during later stages of the project lifecycle.

8.4 Explanation of Discrepancies

As mentioned previously in this document, the scope of this project was simply too large to complete in the timeframe we were given. Another issue that prevented us from implementing all of our requirements was all of the technical difficulties that we encountered. These

difficulties included setting up our development environment, learning unfamiliar frameworks and tools, and deploying our project to ITS's servers. Furthermore, the test user accounts given to us by ITS expired during the project timeline as well, which contributed to a delay in acceptance testing.

9 Project Reflection

9.1 What Went Right

We had several aspects of the project go well for us. One that was consistently positive for both semesters was team communication. We used Slack as our primary tool for online communication. Slack allowed us to consolidate several channels of information, such as GitHub and Trello, into a single location. It also allowed for group messages between all development team members, and individual conversations between two team members. We used comments on Trello cards as a means of status updates, which would then show up on Slack soon after.

Something else that went well for us was usability testing. Usability testing was performed at the beginning of the spring semester using our clickable mock-ups. Since testing was performed early, we were able to implement some of the feedback we received.

One aspect that was key to the success of this project was the amount of support we received from our sponsors. Both our project sponsors from OCSCE and ITS wanted to see this project succeed, so they helped us throughout the duration of the project whenever possible. ITS primarily helped us with the technical side of the project, providing us with deployment instructions to their servers, sample code for Shibboleth authentication, test RIT user accounts, and much more.

Lastly, another success with this project was our development process. For each major system feature, we had a front-end developer and a back-end developer pair up for the development of that feature. Teaming up into sets of two cut down the lines of communication for a given feature and allowed for easier integration.

9.2 What Went Wrong

For the first semester, we mostly encountered process-related challenges, and we then encountered product-related challenges in the second semester. In the fall, our two biggest challenges were holding effective meetings and managing the scope of our project. It was difficult for us to hold effective meetings primarily because of the physical layout of our meeting room. The two sides of the room were far apart from each other, which made it difficult for us to engage our project sponsors. We remedied this by moving the tables in the room closer together so that we could hold discussions more easily.

As for the scope of the project, we were optimistic in the beginning about how many features we could complete by the end of the academic year. However, as the project timeline progressed, we began to realize the scope of the project was too massive for us to complete by the end of the academic year. We solved this by reducing the scope to better fit the schedule since we were unable to extend the timeframe.

In the second semester, we experienced slippage in our schedule for almost every development milestone. This problem is discussed in-depth earlier in Section 5.

Something else that went wrong in the second semester was the lack of time spent on testing. Unfortunately, even though we had developed a formal test plan, testing often fell by the wayside due to the large scope of our project and limited development time. Our intent was to deliver unit tests for each release at the end of each iteration; however, in actuality, Anu, our co-op, ultimately wrote most of our Java unit tests, and these were delivered independently of each system feature. We also had some difficulties with acceptance testing. Our intent for acceptance testing was to perform acceptance tests after the delivery of each milestone; however, acceptance testing was delayed each time due to our schedule slippage and difficulties with providing our project sponsors with the testing materials (e.g. failed delivery of emails and expired test user accounts).

9.3 What Would We Change

We could have done more research into our development tools and frameworks, then used this knowledge to better estimate the scope of the each feature in order to create a more realistic starting schedule. Although we had a comprehensive design to begin development, there were still technical details of this design that we had not fully thought out (e.g. use of Spring controllers).

Another thing we would change was onboarding Anu earlier on in the project timeline, preferably during detailed design. She was unable to start working with us until after Intersession and had to spend a large amount of time in the beginning of the spring semester getting caught up to speed. It would have been greatly beneficial if this was done in the prior semester or during Intersession; however, her start date on the project was out of our control.

9.4 Lessons Learned

Looking back, we should have been more proactive in making updates to our project documents as changes occurred. We had to spend a good chunk of time at the end of the project making sure all of our documentation was up to date, which could have been minimized if we had taken the time to make the changes as we progressed.

Something else we could have done was to ask our project sponsors what their priority was for implementation versus documentation. We feel that we decided to “go for gold” and try to complete as much functionality as possible in the timeframe we were given. Although we were able to deliver intensive documentation that our sponsors were satisfied with, we believe we could have inquired earlier about our sponsor’s preference regarding documentation versus functionality.

10 Appendices

10.1 Glossary

Term	Definition
CES	Co-op Evaluation System
EWA	Enterprise Web Applications, a division of ITS
ITS	Information and Technology Services
OCSCE	Office of Cooperative Education and Career Services
RIT	Rochester Institute of Technology
SRS	Software Requirements Specification
UID	University Identification Number