**1 Introduction**

The Accreditation Board for Engineering and Technology (ABET) establishes standards for engineering and computer science programs across the United States in order to ensure their quality. ABET accreditation is “assurance that the understanding and experience of graduates meets the established standards of their profession.” In principle, ABET accreditation is a gold stamp that can be associated with an academic program or a student’s degree, certifying its value. In a world where lives depend on the quality of engineering projects, ABET accreditation is a critical standard for academic programs to meet.

Northern Arizona University’s Computer Science program first received ABET accreditation in October of 1996, and has been continuously accredited since then. In order for a department to receive ABET accreditation, it must establish a complex, multifaceted process for continually monitoring program effectiveness and recording improvements; this process centers on extensively documenting course content, achievement of learning outcomes, and alumni success. Some of the many possible instruments for measuring program effectiveness include:

* Analysis of students’ grades and performance in various courses.
* Analysis of course contents, their delivery methods, and outcome fulfillment.
* Surveys of current students, program alumni, and employees.

The enormous amount of data collected through these various instruments, as well as instruments themselves, is subjected to regular meta-analysis to yield insights and improvements of program’s overall effectiveness in achieving its mission.

Every six years, the ABET monitoring schema itself, the collected data, and the effectiveness to which the data was used to improve the degree program are audited by ABET in a rigorous year-long procedure. It is through this recurrent process of measurement, analysis, and auditing that ABET accreditation serves as a tool for fostering continual improvement of a given degree program.

The major challenges in the ABET accreditation process are task management, and the archival of critical ABET data. All tasks must be assigned to faculty members and carried out on a well-defined schedule. The major problem with the current manual system is that there are an overwhelming number of tasks to complete over the course of the six-year-long process, and people easily lose track of what they are supposed to do and when they are supposed to do it. Furthermore, with the current system, it is often the case that critical ABET data is not properly collected, is lost, or is not properly analyzed. What is needed is an automated system that manages all of the complexities of the ABET task scheduling, and serves as an efficient and centralized data archival system.

In section 2 of following paper, the ABET accreditation process is discussed in detail. Through a case study and a follow-up analysis, section 3 introduces problems and challenges that manifest through the current system of managing the ABET process. Our approach to solving these problems is presented in section 4. In section 5, we list the set of functional requirements our team elicited through meetings with our clients. Section 6 presents our system’s architectural model from a high-level, and then drills down into its various underlying components. In section 7, descriptions of our system’s major modules are provided and, in section 8, we provide documentation for our datamodel.

**2. Background**

In order to present readers with our solution to the aforementioned problem, it is necessary to first provide background information about the ABET accreditation process; the intent is to give readers a deeper understanding of how the ABET accreditation process works, as well as problems created for faculty members.

**2.1 ABET Model**

The main concept behind ABET accreditation is that a department defines a set of educational objectives, and then carefully monitors the progress of students, as well as the achievement of a set of learning outcomes they define to serve as criteria for meeting the defined objectives. Every department has its own variation of the ABET model, but fundamentally each variant must be structured similarly (i.e. based on objectives, outcomes, and assessments), so that it can be applied to the accreditation process.



**Figure 1.**

The fundamental ABET model (Fig. 1) is as follows: First, a department defines a vision statement, which is essentially a set of high-level goals it aims to achieve; this statement is very high-level, and attainment cannot be measured directly. A simple example of a vision statement could be: “Graduates are able to effectively integrate into the corporate context.” In order for a program’s mission to be assessed, the vision statement is broken down into smaller pieces; these entities are educational objectives that must be met in order for the goals in a department’s vision statement to be achieved. Together, objectives serve as a more detailed description of the program’s vision statement; however, they are still high-level concepts. Following from the previous example, an objective could be: “Graduates are technically proficient.”

In order to meet defined educational objectives, a department must further break them down into learning outcomes, which describe the set of concrete educational achievements that would have to be met to satisfy each objective. Continuing on from the example above, an outcome could be: “Graduates are able to effectively communicate in writing.”

Each outcome is then given any number of measurement tools that serve as a way to quantify how well the given outcome is being met. In the context of our example, a measurement instrument could be, among other tools, a rubric to evaluate student performance in a writing intensive course. Each measurement tool results in multiple datasets, and is given both an analysis and an improvement strategy.

**2.2 ABET Process**

After its definition, a department’s ABET model must be implemented as an active ABET process, so that the required documentation can be generated, and accreditation can be achieved.



**Figure 2**

This process is a six-year long procedure consisting of two cycles (Fig. 2). The outermost cycle represents the six-year-long time period during which a department defines educational objectives, reports are generated, and the department is audited by an ABET committee. The inner cycle represents a much shorter cycle during which a department’s learning outcomes are continually evaluated using measurement tools that result in extensive documentation. In addition, all components of a department’s ABET process must periodically be submitted to meta-analyses so that weaknesses can be recognized, achievements can be identified, and improvement strategies can be both formulated and implemented.

**3.0 Problems and Challenges**

The motivation for our project manifests out of the massive task management and data collection process that is dictated by the ABET accreditation process; this motivation is illustrated below through a case study, and an analysis of the problems that manifest.

**3.1 Case Study: Northern Arizona University (NAU)**

Consider NAU’s Computer Science program. Our vision statement is as follows:

“The Computer Science programs produce graduates who are immediately able to contribute effectively in either corporate or academic contexts. Our educational philosophy emphasizes realistic software development challenges with a focus on teaming, communication and project leadership; our curriculum promotes innovation, design, and exploration of the latest advances in the rapidly changing field of computer science.”

As explained earlier, this statement is very vague and cannot be “assessed” as written. This statement is further broken down into four objective statements:

1. Graduates are technically competent and prepared for leadership and professional practice with strength in design, problem solving, communications, and teaming.
2. Graduates are grounded in computer science and related mathematical fundamentals and prepared for advanced education and lifelong learning.
3. Graduates have an understanding of the scope and implications of the rapid and increasing integration of software-driven technologies into personal and professional spheres of modern society.
4. Graduates integrate quickly into the workplace or advanced education due to an emphasis on high quality teaching, advising and mentoring.

These statements are more apprehensible; however, the scope of each statement is still large. Each objective is associated with several outcome statements. For example, Objective 1 is associated with five outcomes:

* 1. Possess professional skills and knowledge of the software design process.
  2. Ability to function effectively in both co-located and distributed software development teams.
  3. Possess abilities to effectively communicate orally.
  4. Possess abilities to effectively communicate in writing.
  5. Abilities in creativity, critical thinking and problem identification, formulation and solving.

**3.2 Task Management**

Referring back to fig. 1, consider that each outcome is given any number of assessment/measurement tools, which can come in many forms including: rubrics, course evaluations, or surveys. Also, remember that each measurement tool can result in multiple datasets, an analysis, and an improvement strategy. We have asserted that NAU has 4 objectives, and that each may have around 5 outcomes; this gives us 20 outcomes.

(4 Objectives) x (5 Outcomes) = 20 Outcomes

Now, consider that each outcome could be given 5 measurement tools; this results in 100 measurement tools.

(20 Outcomes) x (5 Measures) = 100 Measures

If each measurement tool results in 3 datasets, 1 analysis, and one improvement strategy, there are now 500 assessment tasks that need to be carried out for each outcome.

(100 Measures) x (3 Datasets, 1 Analysis, and 1 Improvement Strategy) = 500 Tasks

Now consider that each outcome could be evaluated each year; this results in 6 evaluations over the course of the six year long process, and now 3000 tasks have to be scheduled, carried out, managed, and monitored! This is quite an overwhelming undertaking, especially if it is carried out in a manual fashion.

(500 Tasks) x (6 Evaluations) = 3000 Tasks

Not only does this process involve the need for extensive documentation and task completion, all tasks must be carried out according to a well-defined schedule in order to ensure proper and accurate evaluation of program outcomes; this increases the complexity of the process.

**3.3 Data Collection and Archival**

Consider that all ABET materials must not only be continually collected according to a well-defined schedule over the course of the six-year-long accreditation process, but analyses must also be performed in order to determine how well learning outcomes are being achieved, and where improvements could be made. The improvements made to a given program as a result of analyses must also be documented and archived. Furthermore, when it comes time to generate reports, all ABET data must be pulled together into a comprehensive report for the ABET auditors.

Currently, all ABET materials exist as distributed sources; this makes it difficult for faculty members to locate critical ABET data necessary for accurate analyses. In addition, this significantly increases the difficulty of pulling data together to generate reports. Considering that ABET accreditation hinges on continual improvement of a given degree program through data collection, analyses of student progress and achievement of learning outcomes, as well as improvements made as a result of analysis, failure to properly archive and locate critical ABET materials at key points in time results in a breakdown of program efficiency and can, in the worst cases, lead to a loss of program accreditation.

**3.4 Failures of Current System**

Given that the ABET accreditation process is an overwhelmingly massive task management and data collection process, handling it in a manual fashion is inefficient and error-prone. Someone must be continually managing the process, delegating tasks to faculty, and monitoring who is supposed to be doing what and whether it is getting done. Furthermore, the person administrating the process must establish an archival system for all the data that is generated.

The result of having one person in charge of all of these tasks often leads to breakdowns in the process including:

* Failure to collect critical ABET data
* Failure to properly archive collected data
* Failure to analyze archived data on schedule
* Assigning and tracking assessment tasks

The main reason for these failures is simple: with so many tasks to complete over the course of a six-year-long process, it is too easy for people to lose track of what they are supposed to do, and when they are supposed to do it.

**4 Proposed Solution**

In order to address the issues outlined in the previous section, our approach was to develop a flexible tool that academic faculties can utilize to model the particular ABET process (academic objectives and learning outcomes) employed by their department, and that comprehensively supports execution of that ABET model over time. Our main goal was to create a system that could essentially drive the ABET process by ensuring that critical ABET tasks are completed when they are supposed to be, and that ABET documentation is properly archived and easily accessible. This required that implementation of two major components: a task management system, and an archival system.

**4.1 Task Management**

The learning outcomes of a department’s ABET model are evaluated cyclically using measurement tools, which can take many forms such as: rubrics, course evaluations, or surveys. In order to obtain results from these measurement tools, they must be populated with informative data. Achievement of this manifests in the form of many tasks that have to be completed by faculties. In addition to completing tasks, faculty members must fulfill their obligations on a well-defined schedule; this is so that when outcomes are up for evaluation, all the supporting data has already been collected, and evaluations can be made accurately. In order to assist faculties in achieving these goals, our system provides the following features:

**4.1.1 Task creation and Assignment to Individuals**

There are three distinct types of tasks that are handled by the Zabeta system: course tasks, assessment tasks, and one-time “todo” tasks. Completion of course tasks is necessary when an outcome is measured with an instrument, which is itself embodied in a course. Assessment tasks are tasks that must be carried out in order to evaluate program effectiveness, but are not linked to a course. One-time “todo” tasks are those that are generated, and carried out once to fulfill an arbitrary program need. The Zabeta system allows for creation of all of the aforementioned tasks and, in addition, provides functionality for associating them with faculty members.

**4.1.2 Task Reminders**

Faculty members are swamped with different tasks over the course of a school year, and the tendency to forget ABET related tasks is very high. To prevent any tasks from being forgotten, the system will send task reminders automatically according to a pre-determined reminder schedule. Reminders are sent both before they are due and after (if tasks are not fulfilled).

**4.1.3 Auto-Generation of Tasks**

Assessment tasks are auto-generated by the Zabeta system; they are scheduled according to a learning outcome’s evaluation cycle. Furthermore, when a new course offering is created, the Zabeta system auto- generates all of the associated course tasks and assigns them to the instructor of the course. This is useful because the system handles all the complexity of realizing that tasks need to be created/scheduled, and assigning them to faculty members at the right time. Effectively, all of this eliminates the possibility of a significant amount of human error in the system. The intended result is that tasks become active according to a well-defined schedule, can be carried out appropriately, and that administrators of the system do not have to worry about task management.

**4.1.4 Task Monitoring**

Through auto-generation of tasks, faculties are relieved from a significant amount of task management duties; however, in order to further ensure that tasks are completed on schedule, the Zabeta system will provide a graphical task monitoring component that allows users of the system to visually monitor the progression of task achievement throughout the ABET process. This is a key feature of the system, because it quickly allows faculty members to visually determine what tasks have and have not been completed, as well as what their temporal relationship is with their associated learning outcome evaluation.

**4.1.5 Monitoring Task Computation Statistics**

Data without interpretation is useless. In order to make improvements to courses and their associated departments, collected data should yield a statistical representation of how assessment measures are performing. The Zabeta system will provide data interpretation facilities such as graphs and grade distributions.

**4.2 Archival System**

With the current manual system, all ABET documents generated through tasks exist as distributed sources in shared drives, word documents, binders, or file cabinets. As such, ABET materials are not easily accessible, are often lost, or are not properly analyzed, which degrades program effectiveness. This also makes the task of compiling all the distributed ABET materials into a comprehensive report error-prone and time consuming. Furthermore, changes to one component of the system are not reflected across the entire system; this leads to inconsistencies in both the data and its interpretation. In order to address these issues, the Zabeta system provides faculty members with an archival system that provides the following:

**4.2.1 Storage and Organization of all Collected Data**

One of the key features of the Zabeta system is the provision of mechanisms that result in persistent storage of all ABET supporting documents and materials. This is achieved through a file management system that allows users to upload, download, and view all ABET documents and materials. This ensures that documents don’t get lost and allows for visualization of archive status and completeness; graphical representation of data collection status and how complete it is a great way to get an overall picture of the progress without going into process details. The Zabeta system will incorporate a graphical timeline to represent this big picture.

**4.2.2 Centralization and Accessibility**

Centralization of all ABET materials is achieved through the provision of a modern web application. This allows faculty members to easily access the ABET system from virtually anywhere, providing quick and efficient access to a consistent set of ABET materials. In addition, this eliminates the existence of inconsistencies in between ABET materials across the system.

**4.2.3 Versioning**

A key feature of the Zabeta system is its versioning capabilities. The versioning system ensures that nothing ever gets deleted; all changes made by users are auditable and can be rolled back. This prevents the breakdown of process integrity by ensuring that erroneous modifications to the system can be fixed. Furthermore, this makes it possible one to determine what changes were made by which faculty members. This, in turn, makes it possible for those faculty members involved to be approached for clarification. The system provides two types of changes: major and minor; major revisions are made only be administrators, and minor revisions can be made by any faculty member.

**4.2.4 Support for Analysis and Report Generation**

Through a centralized, consistent, and accessible archival system, it is easy for faculty members to locate and access all materials necessary to perform analyses of student progress, program effectiveness, and outcome achievement. By making all of the supporting documents and analyses accessible to faculty members, the Zabeta system will in turn, facilitate efficient report generation by relieving faculties from the arduous and complex task of locating supporting ABET documents.

**4.2.5 Customization of Instrument Building**

Our system was designed to be as flexible as possible. One area where this is especially important is collecting assessment data. It is impossible to define exactly what data needs to be collected, so we need a quick and easy way to allow new fields to be added to assessments. In doing so we can build ad hoc instruments made up of whatever data needs to be collected.  
  
This flexibility is accomplished by allowing an instrument creator to define the assessment using wiki syntax. The wiki syntax is called WikiForms. In addition to providing a way to markup the assessment with typographical attributes, WikiForms allow arbitrary fields to be added to the assessment. Later when the assessment is being performed a user enters in the requested values and they’re serialized to JSON and stored in the system.

**5 Customer Requirements**

In order to obtain a concrete understanding of what features our client desired in the system, it was necessary to go through an iterative process of requirements acquisition. During weekly meetings, discussions over the desired system features were carried out with our client; after these meetings, our team decomposed the high-level ideas into concrete functionality, which was then submitted to our client for feedback. Upon receipt of this feedback, our team modified our design/implementation such that it more closely fit our clients’ needs/vision. During this continual process of acquisition, implementation, obtaining feedback, and modification, our team was able to compile a concrete set of requirements, which are examined in detail in the following subsections.

**5.1 Authentication**

The system must provide a simple way to enforce control over resources. The login system will provide support for both Central Authentication Service (CAS) and Open Authorization (OAuth) capabilities. Third-party authentication system must also be supported therefore it is up to organizations to choose their authentication preference.

**5.1.1 Use Case: User Login**

Scenario: Bob tries to log in.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires the ABET ADMS.
2. Bob is presented with either CAS or OAuth authentication screen.
3. Bob enters his user name and password.
4. The system displays the main UI.

System Behavior: The main architectural component in play here is the Authentication System of choice.

**5.2 Access Control**

The system must provide access based on user roles. As such, there are a total of 5 access control roles provided by the system. At the very top level, we have the “superuser;” this role is used by company staff, and allows the user to override all access control provided by the system. The next level is University Administrator; this role allows the user access to all records that belong to a given university. The next level is Program Chair; this role allows the user access to all records that belong to a given program. A department chair is able to manage the program’s information and member users. The next level is faculty; this user has limited access to records that belong to a program, and only records related to that program. The lowest role is disabled; this can be used if a faculty member leaves the university or changes programs; this also allows a user to request access to a program, where by the user occupies a disabled role until they are granted access by the program’s chair.

**5.2.1 Use Case: Access Control**

Scenario: Bob tries to log in and activate a new user Chris.

Actor(s): Bob, an administrator of A1 University’s CS department. Chris is a faculty member of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the ABET ADMS.
2. Bob enters his user name and password.
3. The system recognizes him as an administrator.
4. The screen displays the user interface with ‘activate user’ option.
5. Bob selects the ‘activate user’ option.
6. Chris is now activated.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the Object Manager (OM). The UI handles the front-end display including the workspace, buttons, and menus. The OM handles the creation of objects.

**5.3 User Interface**

The system must provide a web-based graphical user interface to facilitate user interaction and system management. The interface will allow users to interactively define and manage all data and tasks associated with the ABET accreditation process. The interface will provide access to varying perspectives based on usage scenarios and user privileges. The different perspectives will be:

* Class: The system will provide access to specific class information that will identify how well a class has met the requirements of its associated outcomes. This will be accomplished through access to assessment instruments.
* Course: A course perspective will be provided that identifies the intended outcomes associated with each course.
* Faculty: A faculty perspective will be provided that will allow for the identification and updating of faculty tasks and their current status for completing them.
* Administrator: An administrator perspective will be implemented that provides functionality for creating and modifying outcomes, measures, and faculty tasks.

**5.3.1 Use Case: Task Management Display**

Scenario: Bob wants to look through the pending ABET tasks and see who is in charge of that task.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects the option to display pending ABET tasks.
3. The screen displays pending ABET tasks.
4. Bob selects ‘search’ to filter through the pending tasks to see which instructor is in charge of each task.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the Object Manager (OM). The UI handles the front-end display including the workspace, buttons and menus. The OM handles the listing of pending ABET tasks and searching capability.

**5.4 Creation and Modification of Objectives**

Users must allow users to create and modify program objectives for their respective program of study. Functionality for creating and modifying the various components of program objectives will be available to administrators:

* Objective university – the university to which the outcome is associated with
* Objective program – the academic program to which the outcome is linked
* Objective index – identifier of the given objective
* Objective description – overview of the given objective
* Objective outcomes – set of outcomes linked to the given objective

Objectives will have up to n associated learning outcomes and will be versioned.

**5.4.1 Use Case: Creating an Objective**

Scenario: Bob wants to create department objective.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘create objective’.
3. Bob enters the objective to text field.
4. Because Bob is an administrator he has an ability to either increment the major version or save it as a minor version.
5. Bob selects ‘save as a minor version’ and exits.

System Behavior: The main architectural components in play here are the User Interface (UI) module, the Objective module, and the Version module. The UI handles the front-end display including the workspace, buttons, and menus. The objective and version modules handle the creation of objective and appropriate versioning.

**5.5 Creation and Modification of Outcomes**

Users must be able to create and modify learning outcomes for their respective program of study. A given set of outcomes must be linked to a defined program objective. Functionality for creating and modifying the following components of program outcomes must be available to administrators:

* Outcome university – the university to which the outcome is associated with
* Outcome program – the academic program to which the outcome is linked
* Outcome index – identifier of the given outcome
* Outcome summary – overview of the given outcome
* Outcome scheduling cycle – the cycle upon which a given learning outcome is evaluated
* Supporting courses – set of courses linked to the given outcome
* Outcome measures – set of measurement tools linked to the given outcome

Outcomes will have up to n associated measurement instruments, and will be versioned.

**5.5.1 Use Case: Modifying Outcomes**

Scenario: Bob wants modify outcomes associated with objective.

Actor(s): Bob, a faculty member of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘objective’ that is associated with the outcome that he wishes to modify.
3. Bob selects outcome and select ‘modify’.
4. Bob enters changes to the text field.
5. Because Bob is an administrator he has an ability to either increment the major version or save it as a minor version.
6. Bob selects ‘save as a minor version’ and exits.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the Outcome module. The UI handles the front-end display including the workspace, buttons, and menus. The outcome handles the creation of outcomes.

**5.6 Creation and Modification of Measurement Instruments**

Achievement of all learning outcomes is gauged through the use of measurement instruments. As such, users of the system must be able to create and modify measurement tools, embody them in courses if necessary, and link them to outcomes. Functionality for linking individual measurement instruments with up to n learning outcomes is a necessary provision of the system. Measurement tools will be instruments embodied in Wiki Forms such as:

* Rubrics – forms indicating how well certain standards of performance are being met
* Surveys – forms for collecting statistical information about department performance in certain aspects of an academic program
* Course Improvement Documents – documents indicating faculty-defined ideas for improving courses

**5.6.1 Use Case: Modifying Assessments**

Scenario: Bob wants to modify assessments.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘objective’ that is associated with the assessments that he wishes to modify.
3. Bob selects assessment and selects ‘modify’.
4. Bob enters changes to the text field.
5. Because Bob is an administrator he has an ability to either increment the major version or save it as a minor version.
6. Bob selects ‘save as a minor version’ and exits.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the Assessment module. The UI handles the front-end display including the workspace, buttons, and menus. The assessment handles creation of assessments.

**5.7 Scheduling and Task Management**

Provision of a scheduling and task management component is a key feature of the system. Faculty members must be able to login to the system and check what tasks they are responsible for, when those tasks are due, as well as the status of their associated tasks. Administrators will be able to assign tasks as well as monitor the status of all tasks. The system must send email reminders for upcoming tasks as well as overdue tasks.

**5.7.1 Use Case: Task management**

Scenario: Bob wants to check which tasks is Chris responsible for.

Actor(s): Bob, an administrator of A1 University’s CS department. Chris is a faculty member of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘see all tasks’.
3. The system displays the list of all tasks and faculty members associate with them.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the Task module. The UI handles the front-end display including the workspace, buttons, and menus. The task module handles the creation of tasks and listing of current tasks.

**5.8 Versioning**

The system must provide functionality for versioning. The system must keep track of major and minor revisions. Major revisions are made by administrators, and are changes made to the objectives, outcomes, and/or assessments that affect the departmental structure. Minor revisions are made by faculty, and are small changes made to objectives, outcomes, and/or assessments that do not affect the departmental structure. Administrators will be able to track of revisions by certain users, programs, dates, and semesters. Version-able objects will be:

* Forms – forms are used to collect ABET data
* Courses – units of instruction in one topic area
* Objectives – high-level statements indicating desired academic results from application of program curriculum
* Outcomes – concrete educational achievements serving as criteria for meeting defined educational objectives
* Measurement instruments – tools for evaluating program effectiveness

**5.8.1 Use Case: Versioning an Objective**

Scenario: Bob wants modify an objective.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘objective’.
3. Bob selects ‘modify objective’.
4. Bob enters changes in the text field.
5. Because Bob is an administrator, he has an ability to either increment the major version or save it as a minor version.
6. Bob selects ‘save as a major version’ and exits.
7. The major version number is incremented. Their current version is now ver2.0.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the Version module. The UI handles the front-end display including the workspace, buttons, and menus. The version module is responsible for keeping track of all versions, both minor and major.

**5.9 File Management**

The system must provide a mechanism for uploading documents and attach them to components of our data model. Users must be able to upload various types of documents such as pdf, doc, docx, etc. Users will be able to view all documents associated with objects. Users will be able to upload documents associated with course offerings such as:

* Course syllabi – outline of topics to be covered in a course as well as the rules for assessing student performance
* Surveys - forms for collecting statistical information about department performance in certain aspects of an academic program

**5.9.1 Use Case: Uploading documents**

Scenario: Bob wants to upload a pdf contains survey information to CID.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘see all documents’.
3. Bob selects ‘CID’.
4. Bob enters the CID field and select ‘upload file’.
5. Bob selects a pdf from his local file and selects ‘upload’.
6. PDF file is now uploaded onto CID.

System Behavior: The main architectural components in play here are the User Interface (UI) module and the CID module. The UI handles the front-end display including the workspace, buttons, and menus. The CID module is responsible for keeping track of all uploaded documents.

**5.10 Supports for Report Generation and Data Analysis**

In order for the data collected by the system to be used in a meaningful manner, it must be organized so that data for reports and analyses can be pulled together from various locations and compiled into a single document. The system must be able to generate reports and export reporting information to a usable format, such as pdf, doc, docx, etc. Users will be able to backup and restore data from the system.

**5.10.1 Use Case: Reporting**

Scenario: Bob wants to create a word document with collected data.

Actor(s): Bob, an administrator of A1 University’s CS department.

User Steps and system responses:

1. Bob fires up the system and enters login information.
2. Bob selects ‘create pdf document’.
3. Bob selects data to fill the document from drop down menu.
4. Bob selects ‘export’ and a pdf containing selected data is created.

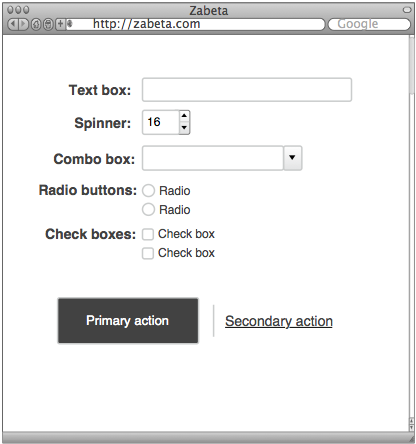
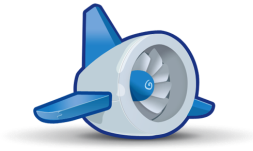
System Behavior: The main architectural components in play here are the User Interface (UI) module and the Data store module. The UI handles the front-end display including the workspace, buttons, and menus. The data store is responsible for saving, backing up, restoring, and retrieving of data.

**5.11 Non-Functional Requirements**

The website should start loading immediately. The client-side HTML4/CSS5/JavaScript6 must complete loading in less than 5 seconds or display a loading message. No API7 call can exceed 30 seconds in execution time. The auto-shared feature of the Google Datastore8 should not enforce limitations on the data storage requirements for this application. Sort order optimization declarations are required for all datasets that are requested with a sort order.

**6. Software Architecture**

At the highest level, Zabeta uses the client-server software architecture. Our server is in fact a “cloud” application running on Google App Engine. A simple diagram of the high level architecture is shown in Figure1. With the client-server design choice we have divided our application into two sets of concerns. On the server side, we deal with data, business logic, and persistent storage. On the client side, we deal with presentation of data and manipulation of data. The client is the GUI for end user interaction with the Zabeta system.



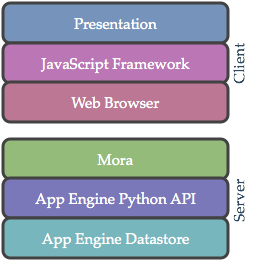
**Client**

**Cloud**

**Figure 3.**

## 6.1 Architecture Overview

Both the client and the server are divided into several more components. We can represent all the components visually using a stack as shown in Figure 4. In the next sections we will explain each component in the stack in detail. We will start at the bottom of the stack and work our way to the top. At each stage we will describe the primary purpose of each component and how it interacts with its neighboring components.

 **Figure 4.**

## 6.2 Server Architecture

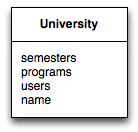
Our server is built using Google App Engine. Google App Engine is a platform for building web applications. As such, it is really a “cloud” application that acts like a server exposing a RESTful service API. The server has three primary components that together provide a RESTful service API that the client uses.

### 6.2.1 App Engine Datastore

At the bottom of the stack we have the App Engine Datastore. The *Datastore* “is a schemaless object datastore” [Google1]. It is responsible for persistently storing raw data. In datastore terminology raw data is a data *entity* (sometimes called a data object). An entity is a set of typed *properties* associated with a *kind*. Each entity has a globally unique identifier (GUID) that exactly specifies the entity within the datastore. Data entities are defined within our application using the App Engine Python Datastore API.

### 6.2.2 App Engine Python API

Google App Engine allows us to build the server using object oriented methodologies. In particular, we use the Google App Engine Python Datastore API to define data entities in terms of *models* with typed properties and *methods*. The models encapsulate our data along with our business logic. Model definitions are Python classes. In particular a model is a subclass of the App Engine provided



class University(db.MoraModel):

# collection of semesters that belong to the university

semesters = db.ReverseReferenceProperty('Semester',

'university')

# collection of programs that belong to the university

programs = db.ReverseReferenceProperty('Program',

'university',

polymorphic=True)

# collection of users that belong to the university

users = db.ReverseReferenceProperty('User',

'university')

# the name of the university

name = db.StringProperty()

**Figure 5.**

classes db.Model or db.PolyModel[[1]](#footnote-1). The class name is the model’s kind which is in turn the kind of all data entities created from the model. The model’s properties are defined by declaring instance variables in the model’s class definition. Each property is assigned a type. As a concrete example see Figure 5 which shows a model represented in UML on the left and the in-code representation on the right (we’re using Mora style models and properties here which will be described in the next section).

With the vanilla App Engine environment one would define request handlers that would handle sending and receiving model representations. The request handlers would make a remote interface to the models. However, we are using an App Engine micro-framework called Mora to build the remote interface. Mora provides some additional functionality that is well suited for Zabeta.

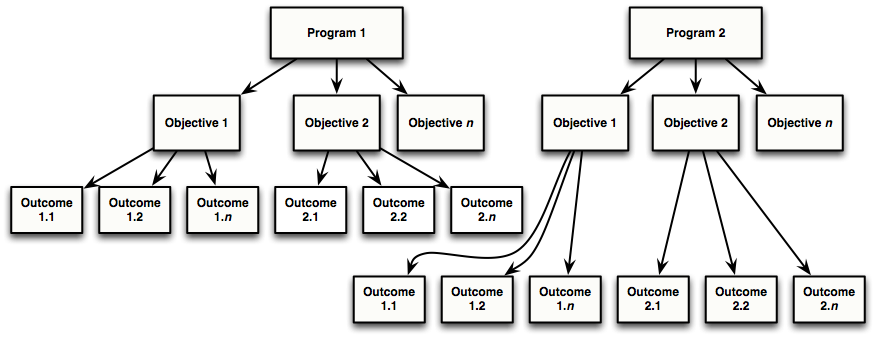
### 6.2.3 Mora

Mora bridges the gap between the models on the server and the web browser. Mora provides a nontraditional interface to the models which will be described. First let’s reconsider the ABET model from the solution overview. In the ABET model (see Figure [abet model]) we have *nodes* that belong to other nodes. For example, objectives belong to a program, and outcomes belong to an objective. As can be seen in the ABET model, the relationships are deeply nested. The graphic is simplified for clarity; the true representation has many *edges* not shown. A program has an edge connecting it to each outcome; objectives and outcomes have an edge back to their program as well. We’ll call the “true representation” the *semantic structure* of the ABET model, since this structure tells us how the entities relate to each other and provides us with a complete view of the data. With the extra edges included, clearly the data is a *directed cyclic graph*.

Since we’re using an object datastore and the datastore API to define our models, the semantic structure of the ABET model is preserved within our data model. If we were to build a traditional RESTful service API to send and receive model representations we would break this semantic structure. We’ll demonstrate this by showing how a traditional RESTful service works and then we will show how Mora works.

#### 6.2.3.1 Traditional RESTful Service Interface

Let’s consider a subset of data that Zabeta stores (see Figure 6) and how it would be retrieved using the traditional methods.

**Figure 6.**

In the traditional RESTful fashion each kind (or *resource*) is given a base URI for the collection. A request handler is mapped to the URI using routes. Request handlers responds to HTTP methods such as GET, POST, PUT, and DELETE at the routes. This is how a browser or any other client would interface with the system.

We can visualize the resource to URI mapping in Table 1.

**Table 1.**

| Resource | URI |
| --- | --- |
| Programs | <http://domain.tld/programs> |
| Objectives | <http://domain.tld/objectives> |
| Outcomes | [http://domain.tld/outcomes](http://domain.tld/objectives) |

To retrieve the programs from the system a client sends a HTTP request to the programs’ URI. The HTTP request to “get” these resources would look like the request in Listing 1.

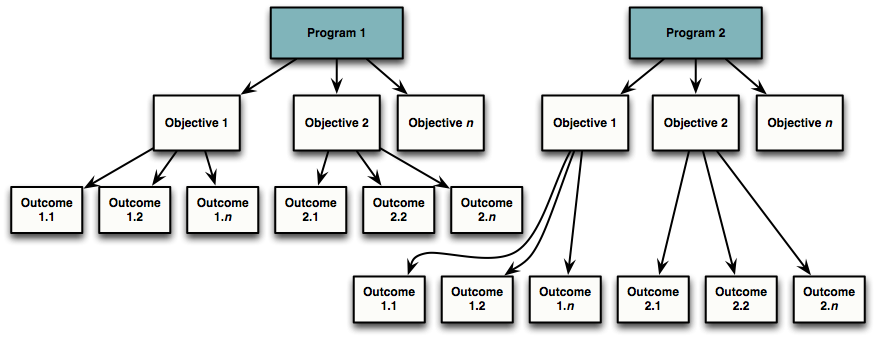
GET /programs HTTP/1.1

Host: http://domain.tld

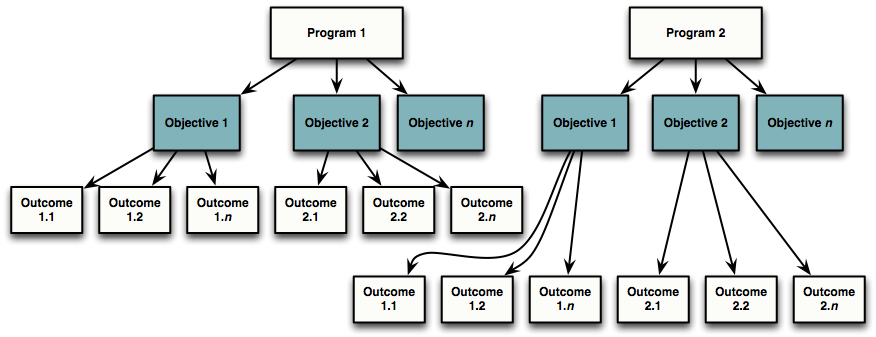
**Listing 1.**

Upon receiving the request the system would respond by sending a representation of all programs. This is illustrated in Figure 7.

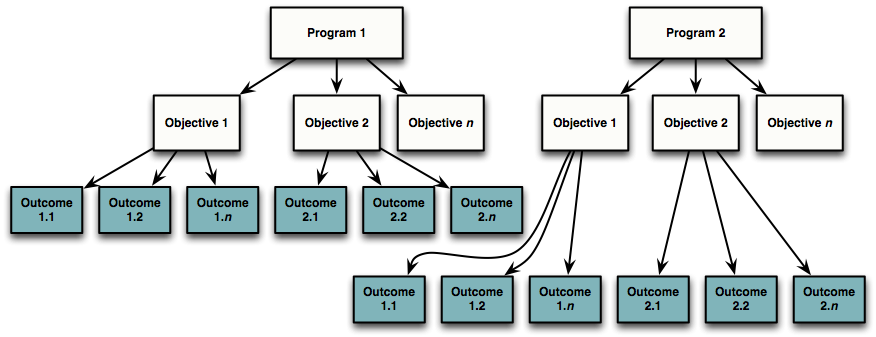
**Figure 7.**



Likewise to retrieve the objectives from the system a client sends a HTTP request to the objectives’ URI. Upon receiving the request the system would respond by sending a representation of all objectives. This is illustrated in Figure 8.

**Figure 8.**

Finally to retrieve the outcomes from the system a client sends a HTTP request to

****

**Figure 9.**

the outcomes’ URI. Upon receiving the request the system would respond by sending a representation of all outcomes. This is illustrated in Figure 9.

Notice that the traditional method retrieves *all* resources in a collection regardless of how it relates to the other resources. The semantic structure is not preserved. To get just the resources that are related to another resource, additional information must be included in the request. Usually this additional information is a SQL-like query appended to the URI or included in the request body. For getting Objective 2’s outcomes the request may look like the request in Listing 2.

GET /outcomes?objective\_id=ag4cXAQn HTTP/1.1

Host: http://domain.tld

**Listing 2.**

Now let us examine how Mora handles requests.

#### 6.2.3.2 Mora’s Graph Interface

By using the micro-framework Mora we can provide an API that preserves the graph structure of the data. Again consider Figure 6, which shows a subset of data that may be in Zabeta. Suppose that you obtained a model *key* for Program 1. A model key is a globally unique identifier that exactly specifies the entity within the datastore. To retrieve a representation of Program 1 we send a request to the APIs base URI with the model’s key appended to the end. We’ll cal this sending a request to a node, in this case the node is for Program 1. The node of any entity is addressable by its canonical URI. Thus, if Program 1’s entity key is ‘ag7kZAQw’ then the canonical URI is the URI in Listing 3.

http://domain.tld/graph/ag7kZAQw

**Listing 3.**

Listing 2 also shows us the APIs base URI http://domain.tld/graph, all request handled by Mora begin at this base URI. Finally the HTTP request to “get” a representation of Program 1 would look like the request in Listing 4.

GET /graph/ag7kZAQw HTTP/1.1

Host: http://domain.tld

**Listing 4.**

Upon receiving the request the system would respond by sending just the representation of Program 1. That is, just a single entity representation. Things become more interesting when we make requests for collections of entities. As we can see in Figure 6 entities are associated with other entities. We can say Program 1 has many objectives. Specifically it has Objective 1, Objective 2, through Objective *n.* It would be convenient if we could request just the objectives associated with Program 1 succinctly. Mora works in just this way.

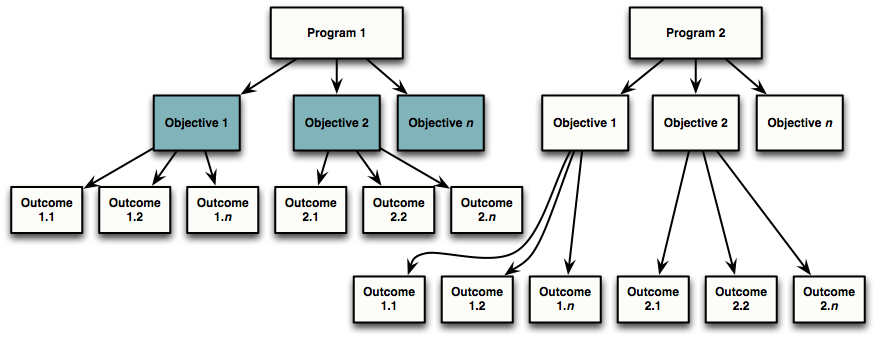
To get just the objectives associated with Program 1 we begin our request as if we were requesting Program 1 and then to the end of the URI we specify the *edge collection* we want. We say that we’re sending a request to Program 1’s “objectives” edge collection. In this case the request would look like the request in Listing 5.

GET /graph/ag7kZAQw/objectives HTTP/1.1

Host: http://domain.tld

**Listing 5.**

Upon receiving the request the system would respond by sending a representation of just the objectives that belong to Program 1. This is illustrated in Figure 10.



**Figure 10.**

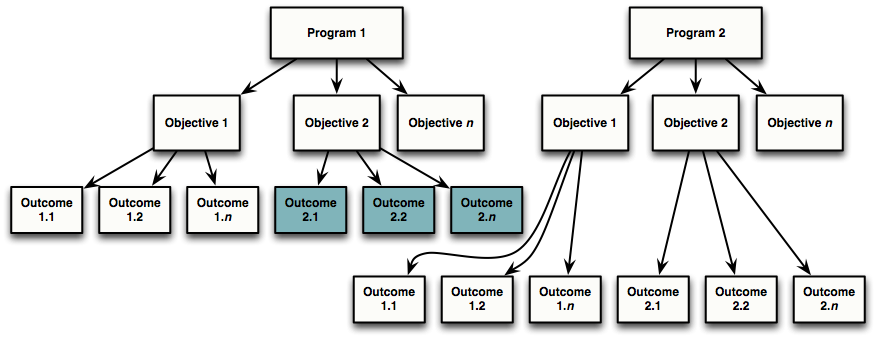
Likewise, to get the outcomes associated with a particular objective, one would query the objective’s “outcomes” edge collection. Let’s say we want only the outcomes that belong to Program 1’s Objective 2. From the response above we can get Objective 2’s key and suppose the key is ag4cXAQn. We begin our request as if we were requesting Objective 2 and then to the end of the URI we specify the “outcomes” edge collection. The request is shown in Listing 6.

GET /graph/ag4cXAQn/outcomes HTTP/1.1

Host: http://domain.tld

**Listing 6.**

Upon receiving the request the system would respond by sending a representation of just the outcomes that belong to Objective 2. This is illustrated in Figure 11.



**Figure 11.**

## 6.3 Client Architecture

The client is a web application built to run in a web browser. The client has three primary components that together provide a user interface for the application. The purpose of the client is to present data and to allow the user to manipulate the data.

**Figure** **12.**

****

### 6.3.1 Web Browser

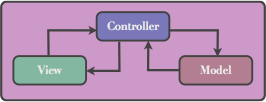
In the same way that App Engine is the platform our server is built on, the web browser is the platform our client is built on. In essence the web browser is further divided into three sub components an HTML rendering engine, a CSS engine and a JavaScript execution environment. The web browser makes requests to Mora using HTTP (HTTP is how a browser communicates with a server). The browser also loads responses from Mora into our scripts running in the browser’s JavaScript interpreter. Figure 12 shows the web browser interacting with Mora.

### 6.3.2 Backbone

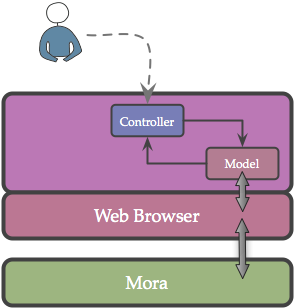
Backbone.js is the JavaScript framework we use to structure our client code. Just as the App Engine Python API gave structure to our server code, Backbone.js gives structure to the client code. With Backbone.js we divide our application into three types of constructs:

1. ***Models:*** encapsulate data and so-called business logic (*i.e.* the computations that process data).
2. ***Views:*** are the parts of the applications the user sees and interacts with. They show data and respond to user actions.
3. ***Controllers:*** simply coordinate between the views and models.

The three components models, views, and controllers together make up what is called the *Model-View-Controller pattern* or MVC for short. Thus, Backbone.js is a MVC framework for web applications. Figure 13 shows how a controller coordinates with a model and a view. A typical MVC application is composed of many models, views, and controllers. The number of each type of object is application dependent, the complexity of the application as well as the granularity of your abstractions influences this number.

**Figure 13.**

#### 6.3.2.1 Interacting with the Server

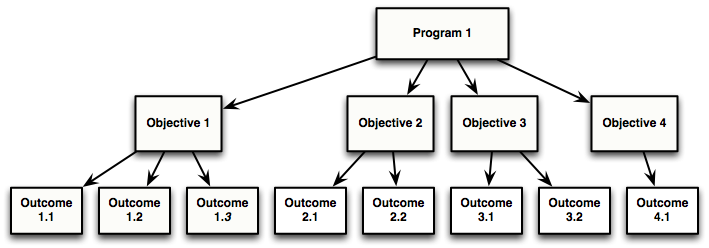
We can walk through a typical interaction between these constructs. Let’s say the user has just navigated to the objective’s view in our application. When this happens, application passes control to the objective controller. The objective controller begins by instantiating a special kind of model known as a *collection*. The collection’s type is set to be an Objective model. Next, the controller instantiates a view that displays a single objective. The controller then tells the collection to *fetch* the objective representation data from the server. The collection knows how to fetch the data from the server and how to parse the data into a set of instantiated objective models. The collection asks the web browser to make a request directly to the server using the Mora API on its behalf. The browser makes the request to the web server and when it receives the response, it loads the data directly into the collection that made the request; the collection parses the data and instantiates the models. This process is illustrated in Figure 14 (the instantiation of the view is left out for clarity).

**Figure 14.**

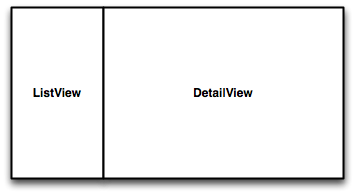
When the view was instantiated it sets up a *listener* that will send a message to the view when the collection loads its models. After the view receives this message it renders itself.

#### 6.3.2.2 Constructing Views

A view typically has a model it is responsible for displaying. As mentioned in Section 6.3.2.1 a model can be a single model or a collection of models. Let’s look at how the semantic relationships of data are presented to the user in our views. We will display collections of objectives that belong to a program. Figure 15 shows the data we want to display.

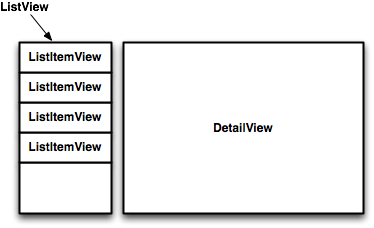
****

**Figure 15.**

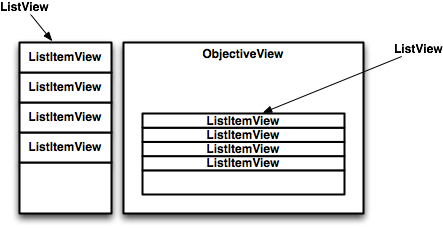
The controller will have the objective collection load Program 1’s objectives. The view will receive a message letting it know that the objectives have been loaded. The controller is managing two views here: a ListView that displays all the objectives, and a DetailView that displays the ObjectiveView. See Figure 16.

**Figure 16.**

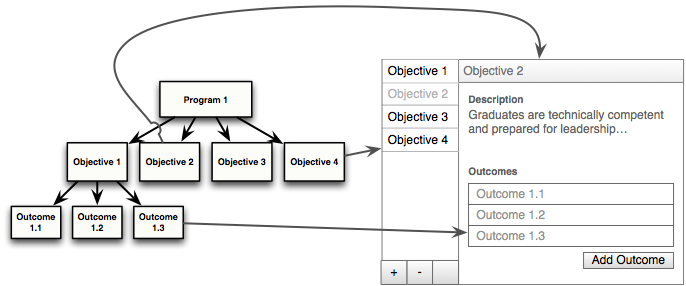
The ListView is composed of subviews. Each objective in the list is displayed using a ListItemView. See Figure 17. When the controller first displays the two ListView and the DetailView it takes the first item in the collection of objectives and tells the DetailView to display it. When a user clicks on an item in the list its view the item notifies the controller that it is now the selected item. The controller then tells the DetailView to display the selected item.



**Figure 17.**

The DetailView is really a proxy for the ObjectiveView. The ObjectiveView has a subview that displays its outcomes. The outcome list subview is another instance of ListView and its subviews are again ListItemViews. See Figure 18. When the ObjectiveView displays itself it creates an internal collection for the outcomes and gets the data for the collection from the server if necessary. It then tells the ****outcome list subview to display itself.

**Figure 18.**

****

**Figure 19.**

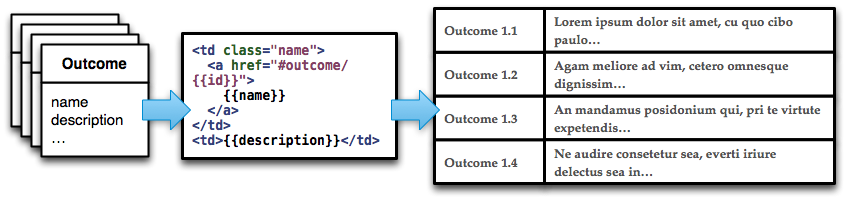
A complete diagram of the views and the data they’re composed of is displayed in Figure 19.

### 6.3.3 Presentation

Finally we reach the presentation layer. This is composed of two components Handlebars and Bootstrap.

#### 6.3.3.1

Handlebars is a client side templating engine. Given a model or collections models, the templates construct the HTML that the views render and display. Figure 20 shows the process of constructing a view from a collection of models using a template.

**Figure 20.**

Here you see a set of models on the left. These are the models we previously described in Section 6.3.2. The models get passed through a handlebars template (center) and are transformed in to the list you see on the right. The list is a view which we also described in Section 6.3.2.

#### 1.3.3.2

For a web application it is important to have a simple and consistent user interface. To see what Bootstrap gives us consider the HTML form in Listing 7.

<form class="well">

<label>Label name</label>

<input type="text" class="span3">

<span class="help-block">Example block-level…</span>

<label class="checkbox">

<input type="checkbox"> Check me out

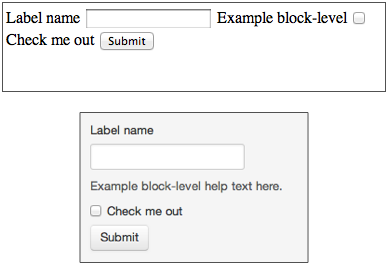
</label>

<button type="submit" class="btn">Submit</button>

</form>

**Listing 7.**

**Figure 21.**

****

This is code for a very basic HTML form. Figure 21 shows two possible displayed versions of this form.

The top form is displayed without Bootstrap -- it's plain and its layout is simplistic. The bottom form is displayed with Bootstrap. It looks a lot nicer. Bootstrap gives us this consistent look and feel across our entire application.

[Google1] <https://developers.google.com/appengine/docs/python/datastore/overview>

**7 Detailed Module Descriptions**

This section details the concepts behind modules required by this system. Each module description explains the approach used to satisfy a given requirement.

**7.1 Authentication**

Universities use different methods of keeping track of their users, and no single authentication method can be used to authenticate every university. There are two major competing standards for authenticating users. Central Authentication Control or CAS is an open standard commonly used by the academic institutions. oAuth is an emerging authentication standard used by many different web services. oAuth has the additional benefit that a University can choose to use a 3rd party authentication provider such as Google. Only the administrator of a University has the ability to choose how users are authenticated.

Both CAS and oAuth have a unique identifier for each user. When a user logs in, we retrieve the AuthenticationRecord based on this unique identifier. The AuthenticationRecord contains a reference to the User record, which allows a single User record to be associated with multiple universities.

If a user is able to successfully authenticate using the authentication method defined by the University administrator and an AuthenticationRecord cannot be found, then a new one is created with “disabled” privileges. The user is able to modify their own user record in order to fill out their personal information. They are also given a list of all Programs associated with this University. Using this list of programs, they are able to request access to any program. It is up to the chair of that program to allow this user account access to a given program.

**7.2 File Management**

In order to allow for collection and archival of critical ABET data and documentation, it was necessary to implement a file management system. The file management system we developed is flexible enough so that users can upload scanned documents in multiple formats such as pdf, jpeg, docx, etc.

All files are stored within the Google Blobstore Python API[[2]](#endnote-1). These files are not accessed on the file system directly with a path; instead, each time a file is uploaded an associated token is provided by the Google Blobstore Python API to reference this blob directly.

Files are associated with records in the Google Datastore using the Blobstore reference property type, which contains the associated Blobstore token. This allows our API to fetch metadata about files that have been uploaded to the system. Some records have a list of Blobstore references, which allows any number of files to be associated with a given record.   
  
In order to associate an uploaded file with a record, the user must provide the record’s key to the following URL:

http://localhost:9999/file/UploadFrame/KEY

File uploads are used to attach thumbnails to the following collections:

* University
* Program
* User
* CourseOffering

Arbitrarily long lists of any file type can be attached to the following collections:

* + Minutes
  + TodoTask
  + CourseTask
  + AssessmentTask

**7.3 Scheduling**

A major requirement for the Zabeta system has been the development of an effective task management and scheduling system. Currently, the Zabeta system allows users to create tasks, assign them to users, and associate them with outcomes; these types of tasks include assessment tasks and course tasks. In addition, users are able to create simple one-time “todo” tasks. All of the aforementioned tasks are coordinated according to start and end times.

The Zabeta scheduling system ensures that assessment tasks get rescheduled periodically according to a user-defined cycle. In order to maintain flexibility in the Zabeta system, users are allowed to define a cycle for constraining the rescheduling of tasks based on their specific school year model (i.e. terms or semesters). In order to allow this, Team Zabeta has defined a set of rules that allow the large amount of possibilities that manifest through this requirement to be implemented. We have expressed this language in Extended Backus-Naur Form[[3]](#endnote-2) (EBNF):

<cycle-string> ::= <operator>(<scalar><term>)|<term>)

<operator> ::= 'start of' | 'end of'

<scalar> ::= 1 | 2 | ... | Infinity

<term> ::= t1 | t2 | ... | tn

When a user creates a new course offering, the system creates a NewCourseOfferingTask, which fires on the day the course is scheduled to start. When this happens, all of the Course’s associated CourseTasks are created and linked to the Course.

A critical component of the scheduling component of the Zabeta system is its ability to monitor the completion of tasks by assigned users and send reminders as necessary. In order to achieve this, the Zabeta system dictates that a given university defines a schedule for how reminders will be given both before and after a task is due. For example, reminders for an arbitrary task could be sent six months, one month, one week, and one day before it is due. This has been achieved with the introduction of two fields:

nags\_before[] – A string list of times to remind a user before a task is due

nags\_after[] - A string list of times to remind a user before a task is due

We have also developed a language that constrains the way in which users define a nagging schedule for reminders; this language is expressed in EBNF below:

<nagging\_cycle> ::= <scalar>|<term>

<scalar> ::= 1 | 2 | ... | Infinity

<term> ::= days | months | years

Every day, a Cron job is run that executes the scheduling script. When this happens, the system checks for expired assessment tasks, and then reschedules them as necessary. Next, all current tasks are checked against a Program’s defined nagging schedule, and reminders are sent out to users appropriately. The scheduling script also checks for NewCourseOfferingTasks that need to be fired, and if one is found, the script creates the NewCourseOffering and all the associated CourseTasks.

A utility module has also been implemented (task\_util.py), which provides the functionality for the initial creation of tasks and the interpretation their scheduling cycle.

**7.4 Sessions**

After a user logs in we need to keep track of their browser and state. To do this, we a session handler is required. Session state is driven by the GAE-Session handler; this session handler uses memcachd[[4]](#endnote-3) and the Google datastore[[5]](#endnote-4) for persistence between page loads. Although memcachd is a fast, it can lose records. In this case, the session handler will retrieve the session from the google datastore and repopulate memcachd. When the client logs in, they are issued a cookie that will keep their session active for 7 days; this cookie consists of a random token and a timestamp. The token is used as a key to obtain the session state from the database.    
  
Every seven days, a Cron job is run to remove expired sessions from the Google datastore. Memcachd records expire naturally after at most seven days

**7.5 Templating**

The goal of templating is to abstract out the structure of web pages from the programs; that is, any kind front end UI structure is not affected by any other specific module. Templates are independent components that are filled with data that the server returns. This provides the ability to easily change how the website looks without having to dig in to the server side code.

Templating is provided by the Handlebar’s JavaScript library, which accepts JSON as input, so it fits in well with our RESTful design. Templates are independent of each other, and have no dependencies besides the Handlebars and jQuery[[6]](#endnote-5) JavaScript libraries.  All the templates are precompiled JavaScript objects, which provide quick load times and general efficiency. Because these templates are all on the client side, they affect load times negligibly.

**7.6 User Access Controls**

In any institution individuals occupy different roles. In the context of this application the roles are tiered.

The highest tier is superuser. This user role has access to all records in the database in every university. This role was created for staff members who will need to maintain our application.

The highest tier for a university is the university administrator. This user has full rights to any record associated with a university. The university administrator is the only user that can create programs, modify how users authenticate, and have access to all programs within the university.

The highest tier for a program is the program chair. This user is able to add and remove users from a program, monitor all tasks within a program, create new courses and plan course offerings.

Most users will occupy the faculty role. This tier allows the user to view what courses and tasks are currently assigned to them. They are also able to view and modify ABET accreditation material.

The lowest access role is disabled. This is a role used as a state for new members of the system prior to a chair's approval. This state is also used if a user has access has been removed.

At no point can a record be associated with multiple programs or multiple universities. In order to enforce these rules, most records need a reference to what University and what Program they belong to. There are some exceptions, however. For instance, there are users that do not belong to any University or Program and their user record can only be modified by that specific user. A University cannot belong to a program, and can only be modified by an Administrator; the chair can only modify a Program. Other than these exceptions, all records associated to a program can be modified by a faculty member.

Vulnerability in our code base could be leveraged to undermine User Access Control entirely, making user accounts useless. There are two vulnerability types that must be addressed by any web application: Cross-Site Request Forgery (CSRF) attacks and UI Redress.

To prevent CSRF attacks, we are disallowing Cross-domain requests. To enforce this rule, we check the “Origin” and “Referrer” http headers of all incoming requests. If an incoming request does not explicitly originate from the same domain as the host, then a “CrossDomainRequestDenied” exception is thrown, and an HTTP error of 403 is returned to the client. To prevent UI Redress, we disallow our web application to be displayed in an iframe by any other domain. In order to enforce this rule, we set the http header:  x-fame-options: SameOrigin

**7.7 Versioning**

For archival reasons all ABET materials must be tracked. At no point can ABET accreditation material be deleted or overwritten as this may jeopardize the ABET audit process. Users of the system should be able to see how the ABET accreditation requirements have changed over time, and revert to an earlier state if desired.

A Minor versioning event takes place any time a record is modified. Conceptually, all versioned records are immutable. When a new record is created, it is pushed onto a stack, and all other records in the system have a pointer to the top of the stack. However, in practice this approach is impossible. When a new record is created, ideally the new record's reference id would just be replaced with the current one; this would maintain all associations. However, the modification of a reference id is not allowed. Instead, the current record is fetched, all of its data-elements are replaced with the modified record, and the old record is preserved by creating a new record to house this data.

When a versioned object is modified, a new copy is created with the commit\_minor version incremented by one. The corresponding VersionTip is modified such that it is now pointing to the latest version. All previous versions can be resolved using a reference to obtain a list of all records that are pointing to the tip.  
  
A Major versioning event takes place when a program would like to make extensive changes to their ABET accreditation process. When this event takes place, a “snapshot” is taken of all ABET requirements; this process uses the Program record as the root. The first step is to duplicate the Program record, then all records pointing to this program are duplicated and the references of the old copy are updated such that everything within the snapshot is pointing to records within the same snapshot. The current records and their references remain unchanged.   
  
All versioned records extend from the “Version” model.   For the purposes of logging, the version model contains the following fields:

* commit\_user - The user that made the change
* commit\_timstamp - The time the change was made
* commit\_comment - Justification provided by the user for the change

For the mechanics of Zabeta’s versioning system to be carried out, the following fields are required:

* commit\_major - The major revision of this record
* commit\_minor - The minor revision of this record
* commit\_tip -  The latest version of this record
* commit\_program\_tip\_ref – This is used to point to this versioned records program root

At any point a record can be reverted back to an earlier revision. A list of all revisions of a specific object is made available to the user. When this takes place, the VersionTip is changed.

**8 Data Model Description**

The Zabeta project is implemented in Python using Google App Engine. Our application creates entities, with all associated data stored as properties of these entities. These properties are named values of one of the data types supported by Google App Engine Datastore. App Engine Datastore provides a NoSQL schemaless object datastore, with a query engine and atomic transactions. In the Python API, a model describes a kind of entity, including the types and configuration for its properties. An application defines a model using Python classes, with class attributes describing the properties.

In order to facilitate the requirements outlined by our clients, it was necessary to implement a number of models; these models can be grouped together with respect to the functionality they provide. The groupings, as well as a detailed description of each model are provided below. A UML model providing a graphical representation of the relationships that exist across our models is also provided at the end of this section.

**8.1 Authentication and Access Control**

In order to provide and facilitate a flexible authentication and access control system, a number of models needed to be implemented. These models are described below:

**8.1.1 Authentication Method**

An AuthenticationMethod is information for how a University authenticates its Users. It allows for the University to use OAuth or CAS and stores the associated information.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointer to the University associated with the Authentication Method. |
| program | Reference | Void pointer to the Program associated with the Minutes object. |
| oauth\_url | String | The OAuth providers landing url for the User. |
| oauth\_client\_id | String | String representations of client ID. |
| oauth\_client\_secret | String | String representations of client secret. |
| cas\_url | String | Url to CAS Authentication method. |

**8.1.2 Authentication Record**

An AuthenticationRecord essentially ties together an AuthenticationMethod with a User. It also keeps track of the privileges a User has within the University’s Program.

|  |  |  |
| --- | --- | --- |
| Field | Type | Desciption |
| university | Reference | Pointer to University associated with AuthenticationRecord. |
| program | Reference | Void pointer to the Program associated with the Minutes object. |
| user | Reference | Pointer to the User associated with AuthenticationRecord. |
| oauth\_id | String | String representation of OAuth ID. |
| cas\_id | String | String representation of CAS ID. |
| programs | List | List of key identifiers for programs associated with AuthenticationRecord. |
| privileges | List | Integer list of privileges. |

**8.1.3 User**

The User class holds information about a system user.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointe to the University that the User is associated with. |
| program | Reference | Pointer to the Program that the User is associated with. |
| full\_name | String | Full name of User. |
| display\_name | String | Name of User to be displayed by the system. |
| email | Email | Email address of the User. |
| employee\_id | String | Employee identification string. |
| phone\_office | String | Office phone number of User. |
| phone\_personal | String | Personal phone number of User. |
| office | String | User’s office identification string. |
| join\_date | Date | Date that User joined the University/Program. |
| depart\_date | Date | Date that the User left the University/Program. |
| thumbnail | String | Identification string of User image. |
| webpage | String | Url of User’s website. |
| tasks | List | Key list of User’s tasks. |

**8.2 Institutional Structure**

A number of models were also implemented to accurately model the structure of institutions and their associated programs. These models are described below.

**8.2.1 Course**

The Course class inherits from Version and holds metadata related to a given Course. Each Course belongs to a program.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointer to University associated with the given Course. |
| program | Reference | Pointer to Program associated with the given Course. |
| name | String | Course name. |
| description | Text | Course description. |
| catalog\_descr | String | Description of course catalog. |
| catalog | String | Program and Course number. |
| Webpage | String | Url of Course webpage. |
| core\_topics | String | Listing of the core topics covered in the given Course. |

**8.2.2 CourseOffering**

The CourseOffering class inherits from Version and holds information about a specific Course instance. When a course is offered a new CourseOffering instance is created. This instance is associated with a term and a year, such as “spring” “2012”. This course is taught by an instructor so it has a reference to a User. This object contains stats for how well the class performed.

|  |  |  |
| --- | --- | --- |
| Fields | Type | Description |
| university | Reference | Void pointer to the University associated with the CourseOffering. |
| program | Reference | Void pointer to the Program associated with the CourseOffering. |
| semester | Reference | Pointer to the Semester associated with the CourseOffering. |
| instructor | Reference | Pointer to the User who is teaching the CourseOffering. |
| course | Reference | Pointer to the Course object that a given CourseOffering. |
| student\_count | Integer | Number of students in the CourseOffering. |
| section | Integer | Section number of the CourseOffering. |
| final\_grades | StringList | List of the students’ grades at the end of the CourseOffering. |
| tasks | StringList | List of the tasks linked to a given CourseOffering. |
| syllabus | String | String identifier of the syllabus document. |
| website | String | Url of the Course website. |

**8.2.3 Program**

The object module is the wrapper class which contains university, program and user ID as references.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointer to a University. |
| program | Reference | Pointer to a Program. |
| Uid | String | Represents user IDs. |

**8.2.4 Program**

The Program class inherits from Version and holds information about a department that belongs to a University.

|  |  |  |
| --- | --- | --- |
| Fields | Type | Description |
| university | Reference | Pointer to University object associated with a given program. |
| program | Reference | Pointer to Program. |
| name | String | Name of the Program. |
| start\_date | Date | Date of Program installations at University. |
| end\_date | Date | Date of Program termination at University. |
| mission | String | Statement of Program goals/objectives. |
| description | Text | Program description. |
| webpage | String | Url of Program website. |
| thumbnail | String | String identifier of Program image file. |
| docs | String | String identifier of Program docs array. |
| nags\_before | StringList | Defines frequency of reminders before a task is due. |
| nags\_after | StringList | Defines frequency of reminders after a task is due. |

**8.2.5 Semester**

The Semester class holds information about a time period when courses are taught throughout the school year. A given University’s school year model is defined as a list of Semester objects.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointer to the University associated with the Semester. |
| program | Reference | Pointer to the Program associated with the Semester. |
| begin\_date | DateTime | Date and time representing the start of a Semester. |
| end\_date | DateTime | Date and time representing the end of a Semester. |
| name | String | Name of Semester. |

**8.2.6 University**

The University class inherits from Version and holds metadata about a given University.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| name | String | University name. |
| domain | String | Domain name of University. |
| semesters | String | School year model for University. |
| login\_path | String | Path on our system used for Users to login. |
| programs | String | List of Programs associated with the University. |
| thumbnail | String | String identifier of University image file. |
| web\_page | String | Url of University website. |

**8.3 ABET Structure**

In order to provide a flexible structure for housing a given program’s ABET model, as well as access to all of its associated data, a set of models were implemented; these models are describe below.

**8.3.1 Form**

A form is the parent class that houses the WikiForm. This is used to collect data from Users about measures.

|  |  |  |
| --- | --- | --- |
| form\_name | String | Name of the form. |
| assessment\_form | String | WikiForm in WikiForm syntax. |
| require\_attachments | StringList | A list of the attachment names required to satisfy this form. |
| description | Text | Form description. |
| instructions | String | From instructions. |

**8.3.2 Instrument**

The Instrument class inherits from Form and holds information about an assessment instrument.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointer to the University associated with the Instrument. |
| program | Reference | Pointer to the Program associated with the Instrument. |

**8.3.3 Minutes**

Minutes inherits from Version and is a revision notepad associated with a program. Anyone within the program can modify this notepad.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Void pointer to the University associated with the Minutes object. |
| program | Reference | Pointer to the Program associated with the Minutes object. |
| description | Text | Description of the Minutes object. |
| docs | String | String identifier of the docs array. |
| date | Date | Date of the creation of the Minutes object. |
| content | String | Content of the Minutes object. |
| attachment\_names | StringList | A list of file names paired with the blobs below. |
| attachment\_blobs | StringList | A list of blobs IDs to retrieve the file from the Google Blob Store. |

**8.3.4 Objective**

The Objective class inherits from Version and holds all the information about an Objective. Defined by a program, objectives are descriptions of goals that the program intends to meet to maintain ABET accreditation.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Void pointer to the University associated with the Objective. |
| program | Reference | Pointer to the Program associated with the Objective. |
| name | String | Objective name. |
| description | Text | Objective description. |
| index | Integer | Index of the Objective. |
| outcomes | List | List of key identifiers of outcomes associated with Objective. |

**8.3.5 Outcome**

The Outcome class inherits from From and holds information about a given Outcome that has been defined by a Program.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Void pointer to the University associated with the Outcome. |
| program | Reference | Void pointer to the Program associated with the Outcome. |
| rationalize\_instrument | Reference | Pointer to the Instrument. |
| name | String | Name of Outcome. |
| rationale | String | Reasoning behind Outcome. |
| assessments | KeyList | List of keys identifying the assessments associated with an Outcome. |
| last\_evaluation | DateTime | Date and time of last evaluation of Outcome. |
| evaluation\_start | DateTime | Start date and time of the next evaluation. |
| evaluation\_end | DateTime | End date and time of the next evaluation. |
| rationalize\_course | List | List of key identifiers of all Courses linked to Outcome. |
| index | Integer | Index of Outcome. |

**8.3.6 Measure**

This class is a subclass of an Outcome class; it contains information about instruments and courses used to evaluate a given Outcome.

|  |  |  |
| --- | --- | --- |
| Fields | Type | Description |
| university | Reference | Void pointer to the University associated with the Outcome. |
| program | Reference | Void pointer to the Program associated with the Outcome. |
| course | Reference | Pointer to the Course associated with the Outcome. |
| instrument | Reference | Pointer to the Instrument associated with the Outcome. |
| rationale | String | Reasoning behind Outcome. |
| goal | String | Goal of Outcome. |

**8.4 ABET Tasks**

A number of models were implemented that maintain, and provide access to, all of the information related to a program’s tasking system; these models are described below.

**8.4.1 AssessmentTask**

Assessment tasks are tasks that are associated with an outcome that needs to be evaluated on a cyclical basis.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Void pointer to the University associated with the Task. |
| program | Reference | Void pointer to the Program associated with the Task. |
| outcome | Reference | Pointer to the Outcome associated with the Task. |

**8.4.2 CourseTask**

CourseTask class inherits from Task and represents a Task object that is carried out with respect to a given Course. CourseTask objects are rescheduled by the system.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| university | Reference | Pointer to the University associated with the Task. |
| program | Reference | Pointer to the Program associated with the Task. |
| course | Reference | Pointer to the Course associated with the Task. |
| rubric | Reference | Pointer to the assessment Instrument associated with the Task. |

**8.4.3 Task**

The Task class inherits from Version and holds information about a task object. Tasks represent actions that must be carried out to in order to evaluate the achievement of program outcomes. They can be assigned to a list of delegates by a User of the system, or they can be scheduled automatically.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| name | String | Name of the task. |
| begin\_date | Date | Date the task begins. |
| end\_date | Date | Date the task ends. |
| delegates | List | List of keys identifying Users that have been assigned the given task. |
| fulfilled | Integer | Indicates if a task has been fulfilled or not (0=false, 1=true). |
| attachment\_names | StringList | A list of file names parired with the blobs below. |
| attachment\_blobs | StringList | A list of blob IDs to retrieve the file from the Google Blog Store. |
| response | String | Form response. |

**8.4.4 TodoTask**

TODOTask is a sublass of Task; it represents a singleton Task that is not rescheduled.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| description | Text | Task description. |

**8.4.5 ScheduleLog**

The ScheduleLog class holds/logs information about scheduling events.

|  |  |  |
| --- | --- | --- |
| Fields | Type | Description |
| university | Reference | Pointer to associated University. |
| program | Reference | Pointer to associated Program. |
| Task | Reference | Pointer to associated Task. |
| User | Reference | Pointer to associated User. |
| Email | Email | Email address of associated User. |
| timestamp | DateTime | Date and time of scheduling event. |

**8.5 Versioning**

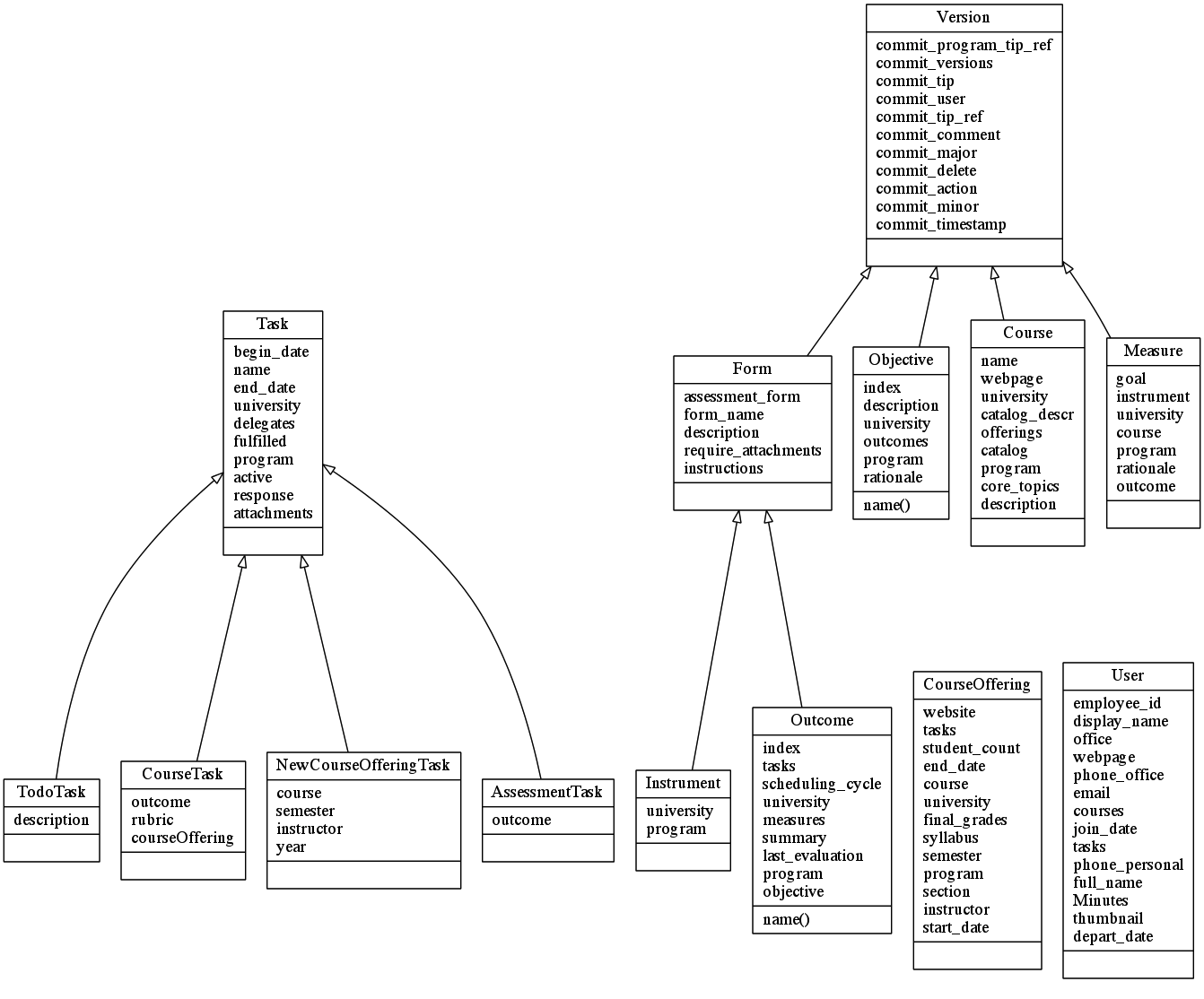
In order to handle all versioning information as well as provide a revisioning system for our clients, a Version model was implemented; this model is describe below.

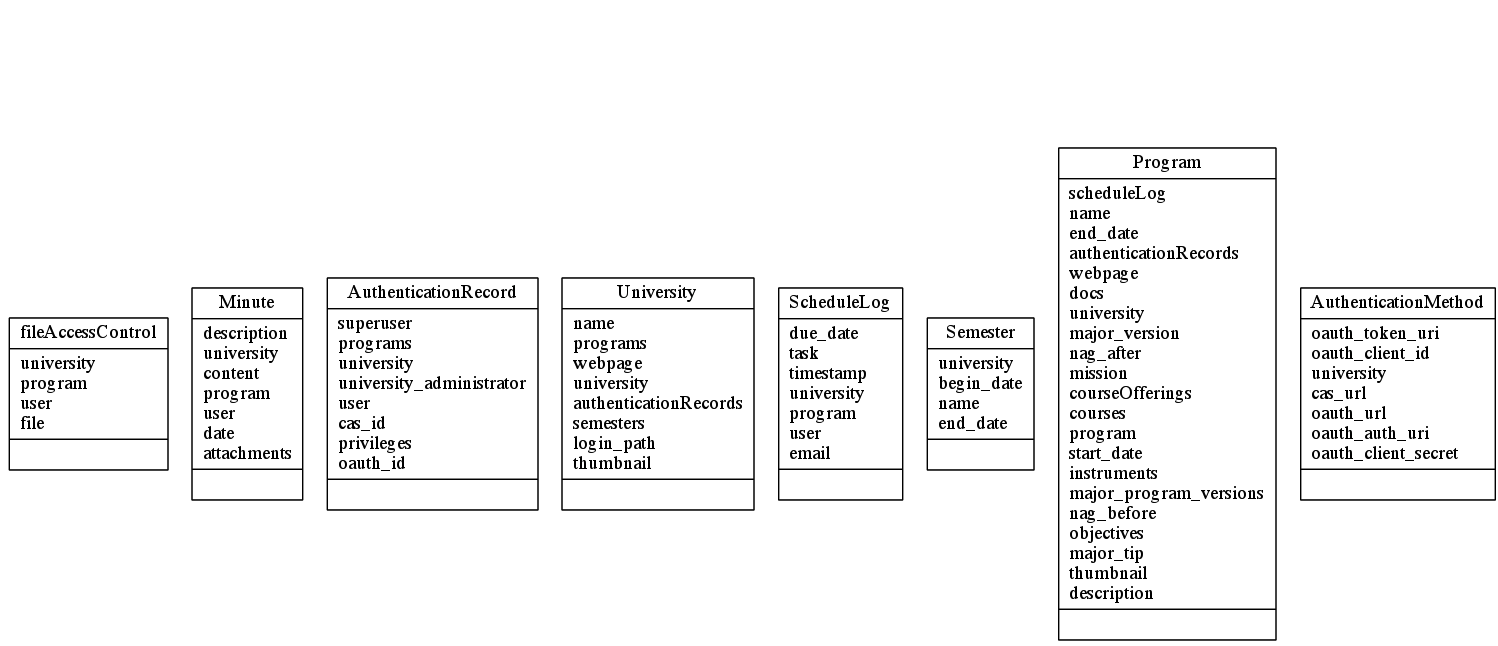
**8.5.1 Version**

Version is the parent class of all versioned objects in the system. The version parent is also a set that contains the union of all child-classes. Therefore, this set can be queried by any value such as a user ID to obtain a list of all changes made by a specific user. A version contains a reference to the user that made the modification and reference to the program that this data-element belongs to and the user that made the change. These two references are for the purpose of audits because a simple query can list every modification made to a program over a date range, or every modification made by a user over a date range. The commit\_comment element is the User’s rational for making their changes.

|  |  |  |
| --- | --- | --- |
| Field | Type | Description |
| commit\_user | Reference | Pointer to the User type who executed version commit. |
| commit\_program | Reference | Pointer to the program associated with the version commit. |
| commit\_object | Reference | Pointer to the actual commit Object type. |
| commit\_minor | Integer | Minor version number. |
| commit\_timestamp | DateTime | Date and time of version commit. |
| commit\_comment | String | Description/rational for version commit. |
| commit\_tip | Integer | The latest version of this record. |

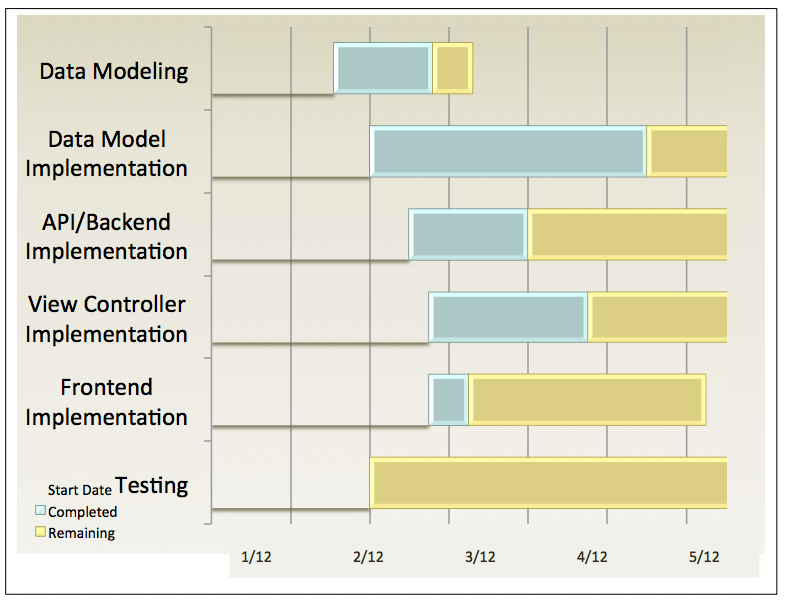
**8.6 Data Model UML**

****

****

**9 Implementation Plan**

Below is an overview of our implementation plan.

****

**10. Conclusion**

A description of our design challenges is provided below as well as a set of closing comments.

**10.1 Design challenges:**

Google's BigTable database has some huge advantages and huge disadvantages. On one hand, it is non-relational, which allows the data to be transparently segmented, and split up between clusters of machines. From the application's perspective, this means that the database can hold effectively unlimited amounts of data, which is highly desirable. However, this database also introduces design challenges. Most notably you cannot “join” or “union” two collections; instead, an additional query is required to resolve this connection.

Even more changing are the limitations on what can be queried. A query cannot contain two inequalities for two different parameters. For instance, the query “select from Task where start\_date > now() and end\_date < now()” is invalid because it contains inequalities for both start\_date and end\_date. To resolve this issue, Tasks are given an “active” bit, in which the start\_date is checked to see if the record is active and then this bit is set, and then the end\_date range is checked. This allows us to use the valid query “select from Task where active=True and end\_date < now()”

Some records must be versionable. Being able to update records directly is very easy, and this easy method cannot be used for all data. The versioning behavior must appear to be transparent from the clients’ perspective. The client only knows simple HTTP methods, which is limited to GET, POST, PUT, and Delete. From the servers perspective these calls have to incur a revisioned state change, while maintaining references between records.

**10.2 Open Issues**

The majority of incomplete functionally resides in the frontend. This application is a web service, and this creates a shift in complexity, putting a large burden on the frontend. With some platform choices, the frontend development is more simplistic and takes less time to develop, the tradeoff being that the frontend is less interesting, less useful and less efficient.

There was not enough time to implement every desired feature. When developing an application on a time frame, it is necessary to prioritize the implementation of features. Some features, such as user management, scheduling, versioning, file management, and course planning where absolute requirements. The final application simply would not be functional without these features. The frontend interface for some of these features could be improved, for instance course planning should have a more complex user interface because of its importance. The current implementation of this feature is a bare-bones approach, and with more time this feature could be more useful.

More testing is needed. You cannot prove the absence of bugs only that a bug exists. At the time of this writing there are no known bugs or exceptions in the code base. That being said, we have to allocate enough time to fully test this application for a production environment. The best test that can be conducted for this application is using it in a real ABET accredited program. The flaws and incorrect assumptions about the applications design will be made apparent. This is the next logical form of testing for this application.

**10.3 Closing Comments**

This application is mostly functional. Development was not rushed, and by in large there was an effort to implement the application in the best way possible. Not every feature could be implemented in the time available, but we did our best given the resources available. The applications current state provides an excellent foundation for this project to grow into a killer app and hopefully a successful company.

**11. Glossary**

ABET- The Accreditation Board for Engineering and Technology, Inc., is a non-governmental organization that accredits post-secondary education program in applied science, computing, engineering, and engineering technology.

CAS– Central Authentication System; is a single sign-on protocol for the web. Its purpose is to permit a user to access multiple applications while providing their credentials only once.

OAuth– Open Authorization; is an open standard for authorization. It allows users to share their private resources stored on one site with another site without having to hand out their credentials, typically username and password.

HTML- Hyper Text Markup Language is the main markup language for web pages.

CSS- Cascading Style Sheets is a style sheet language used to describe the presentation semantics of a document written in a markup language.

Java Script- is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

API- An application programming interface is a source code based specification intended to be used as an interface by software components to communicate with each other. An API may include specifications for routines, data structures, object classes, and variables.

Google Datastore- a data storage method provided by Google for developers to store and serve data from their Google Cloud Storage service.

Python-is a general-purpose, high-level programming language whose design philosophy emphasizes code readability.

Google App Engine- is a platform as a service could computing platform for developing and hosting web applications in Google-managed data centers.

OSX- Mac OS X is a series of Unix-based operating systems and graphical user interfaces developed, marketed, and sold by Apple Inc.

Linux- is a Unix-like computer operating system assembled under the model of free open source software development and distribution.

Windows- is a series of operating system produced by Microsoft.

Google Python SDK- Python platform to implement application on Google App Engine.

1. In Zabeta we use Mora’s db.MoraModel and db.MoraPolyModel rather than the vanilla db.Model or db.PolyModel. [↑](#footnote-ref-1)
2. Google Blobstore Python API- [↑](#endnote-ref-1)
3. Extended Backus-Naur Form- [↑](#endnote-ref-2)
4. Memcache- [↑](#endnote-ref-3)
5. Google datastore- [↑](#endnote-ref-4)
6. jQuery-

   //Need to insert wire frame and class diagram [↑](#endnote-ref-5)