Software Architecture Documentation

Co-op Evaluation System

Senior Project 2014-2015

Team Members:
Tyler Geery
Maddison Hickson
Casey Klimkowsky
Emma Nelson

Faculty Coach:
Samuel Malachowsky

Project Sponsors:
Jim Bondi (OCSCE)
Kim Sowers (ITS)

Table of Contents

Revision History

Version	**Primary Author(s)**	**Description of Version**	**Date Completed**
v1.0	Emma Nelson, Maddison Hickson, Casey Klimkowsky, Tyler Geery	Initial revision	October 27, 2014
v1.1	Casey Klimkowsky	Update after receiving feedback from Lisa on 10/31/14	November 3, 2014
v1.2	Emma Nelson	Validate changes	November 6, 2014

1 Introduction

The purpose of this document is to provide a detailed architecture design of the new Co-op Evaluation System by focusing on four key quality attributes: usability, availability, maintainability, and testability. These attributes were chosen based on their importance in the design and construction of the application.

This document will address the background for this project, and the architecturally significant functional requirements. Each of the aforementioned quality attributes will be described through a comprehensive set of scenarios followed by an architectural overview, which includes a bird's eye view and a full description of patterns and tactics that will be used to address the core quality attributes. This will be followed up by a look at a couple views into the system. Finally, acknowledgements, references, and appendices will round out the document.

The intention of this document is to help the development team to determine how the system will be structured at the highest level. It is also intended for the project sponsors to sign off on the high-level structure before the team shifts into detailed design. Finally, the project coach can use this document to validate that the development team is meeting the agreed-upon requirements during his evaluation of the team's efforts.

2 Background

RIT's current Co-op Evaluation System, an application used by OCSCE, has a number of performance, reliability, usability, and maintainability issues. Among others, session timeouts and submission timeouts are inherent problems of the current Co-op Evaluation System. A new version started from scratch with up-to-date technologies needs to be developed.

The purpose of this project is to re-engineer the Co-op Evaluation System in order to leverage newer web technologies while also improving performance and user interaction. Since we are essentially recreating the CES, the new system has to interface with any external components that the current CES uses or the replacement systems, as determined by ITS. These components include Shibboleth for RIT user authentication, the ITS mail server for sending emails to users, and an Oracle SQL database for storing system information. Refer to the Software Requirements Specification for a context diagram and a detailed description of how these components interact. The context diagrams are also available in section 5.1 of this document.

The system must comply with the development guidelines provided to us by ITS, as defined by the EWA Student Development Guidelines wiki page. At a high level, these guidelines include approved application frameworks, build tools, application server technologies, database standards, and several other technology standards. Although these constraints will primarily affect the detailed software design, we still need to take them into consideration when creating the system architecture.

3 Functional Requirements

Many of the features involve saving, updating, or viewing evaluation forms, and thus will need to be accounted for in the architecture due to amount of interfacing with the database required. The system must support concurrent reads from, and writes to the database.

Additionally, the system may have the need to interface with several external APIs. The system must interact with ITS's email server in order to send emails to students, employers, and other users. Furthermore, although the particular services are unknown at this time, it is likely that the system will have to interface with an external report and form generation tools. These features are architecturally significant, as the system should be designed in a modular fashion so that external services may be swapped in and out with ease to handle future updates.

For a detailed description of all functional requirements, refer to the Software Requirements Specification.

4 Quality Attributes

The following tables describe concrete scenarios for the top four quality attributes that must be included in the final system. The architectural drivers are prioritized in order of significance.

4.1 Usability

Scenario	The end user wants to discover what features are available to them.
Source	End user
Stimulus	End user wants to learn system features
Artifact	Co-op Evaluation System
Environment	At runtime
Response	The system provides an interface that feels familiar to the user and follows good practice in website interface design to improve learnability
Response Measure	Number of errors in completing a task, ratio of successful operations to total operations

Scenario	The end user wants quick access to core features for their user class to improve efficiency of use.
Source	End user
Stimulus	End user wants to improve efficiency

Artifact	Co-op Evaluation System
Environment	At runtime
Response	The system displays pertinent information by default on the user’s home screen without forcing them to dig into nested menus to find the information for which they are looking.
Response Measure	Task time

Scenario	The user wants to receive user- and situation-appropriate error messages when an error occurs.
Source	End user
Stimulus	Minimize impact of errors
Artifact	Co-op Evaluation System
Environment	At runtime
Response	The system will provide visual feedback stating whether or not a given action was successful. If it was not successful, the system will provide details on what went wrong and how to rectify the situation, when possible. Furthermore, the system will send the user a follow-up email with the status of any submissions in the case of Work Reports and Co-op Evaluations.
Response Measure	User satisfaction, amount of time lost, amount of data lost

Scenario	User wants to know which actions are available and that the action they choose is being executed correctly.
Source	End user
Stimulus	Increasing confidence and satisfaction
Artifact	Co-op Evaluation System
Environment	At runtime
Response	The system will only display actions that are currently available as “active”. Any other options with either be hidden or greyed out and unclickable. Furthermore, the system will provide visual feedback stating whether or not a given action was successful.
Response Measure	User satisfaction, gain of user knowledge

4.2 Availability

Scenario	The system times out before a Work Report or Employer Evaluation can be submitted.
Source	Internal to system
Stimulus	The maximum time allowed on form submission page has elapsed
Artifact	Work Report or Employer Evaluation
Environment	Normal operation
Response	Notify the user that their session has timed out, and that the current form was not submitted.
Response Measure	Number of session timeouts

Scenario	The application server fails or becomes unresponsive, causing the entire system to fail.
Source	Internal to system
Stimulus	A fault within the application server
Artifact	Application server
Environment	Degraded
Response	The entire system shuts down until the application server is brought up again.
Response Measure	Percentage of uptime (Minimum of 95%)

Scenario	The system is compromised by a Denial of Service (DoS) attack.
Source	External to system
Stimulus	The system receives more requests per second than it can handle
Artifact	Web service
Environment	Normal operation
Response	Standard response according to RIT Systems and Operations protocol.

Response Measure	Length of attack (The DoS attack does not continue for longer than 1 minute.)

Scenario	The system is unable to handle a large number (over 6,000) of concurrent user requests.
Source	External to system
Stimulus	The system receives more concurrent requests than it can handle
Artifact	Web service
Environment	Normal operation
Response	The system blocks additional concurrent user requests, and displays a message to the user to try accessing the system again later.
Response Measure	Load time (Takes no longer than 10 seconds for pages to load)

Scenario	A bug or fault in the application causes a system-wide failure.
Source	Internal to system
Stimulus	A bug or fault in the application
Artifact	The component in which the bug or fault occurred
Environment	Degraded
Response	The system handles any and all exceptions that may occur, so that the system may fail gracefully.
Response Measure	A meaningful error message is logged, indicating what it was that caused the application to fail.

4.3 Maintainability

Scenario	Code base is large and complex making it difficult to add new features
Source	Developer, Maintainer
Stimulus	New feature or additional functionality is desired
Artifact	Detailed Design Document, The Co-Op Evaluation System

Environment	Design time, runtime
Response	Refactor code to be simpler and contain the new functionality
Response Measure	Time spent refactoring

Scenario	Lack of documentation hinders usage, management, and future upgrades.
Source	Developer, Maintainer
Stimulus	Documentation was not made a priority throughout the development of the application and thus does not provide the most up-to-date information on the system features and functionality
Artifact	System documentation
Environment	At runtime
Response	Developers spend time improving documentation
Response Measure	Time spent understanding the system that would have been aided by more robust documentation

Scenario	Excessive dependencies between components and layers and inappropriate coupling to concrete classes prevents easy replacement, updates, and changes.
Source	Developer, Maintainer
Stimulus	Developer wish to replace, update, or modify part of the system.
Artifact	The Co-op Evaluation System
Environment	At runtime
Response	Re-design the system with well-defined layers, or areas of concern, that clearly delineate the system's UI, business processes, and data access functionality. Application configuration for commonly changed parameters, such as URLs, will maintained outside the code base following RIT EWA standards.
Response Measure	Time spent redesigning and refactoring

4.4 Testability

Scenario	There is a lack of test planning.
Source	System verifier
Stimulus	Subsystem integration completed. Testing did not start early during the development life cycle.
Artifact	The Co-op Evaluation System
Environment	At design time
Response	Prepare a test environment
Response Measure	Test coverage, percent executable statements executed, length of time to prepare test environment

Scenario	Test coverage of the system is inadequate.
Source	All testers
Stimulus	Subsystem integration completed os system delivered. Because both manual and automated tests don’t cover a large portion of the project, the testing team wish to expand the test suite.
Artifact	The Co-op Evaluation System
Environment	At development time
Response	Provide a fuller testing environment, which includes automated tests and coverage reporting.
Response Measure	Percentage of paths or executable statements covered

Scenario	Testing is not consistent throughout the system.
Source	System verifier, Client acceptance tester
Stimulus	Subsystem integration completed os system delivered. Automated or granular testing cannot be performed if the application has a monolithic design
Artifact	The Co-op Evaluation System
Environment	At deployment time

Response	Design the system to be modular to support testing. Design components that have high cohesion and low coupling to allow testability of components in isolation from the rest of the system.
Response Measure	Test coverage, percent executable statements executed, length of the longest dependency chain in a test

5 Architecture Overview

5.1 Big Picture

The major subsystems are described in high-level details below in Section 6 Views. In short, they are divided up by layers. There is the view, business logic, domain model, data interaction layer, and data source. Each of these divisions is vital for the system to operate. Refer to Section 6 for further details.

5.1.1 System Context

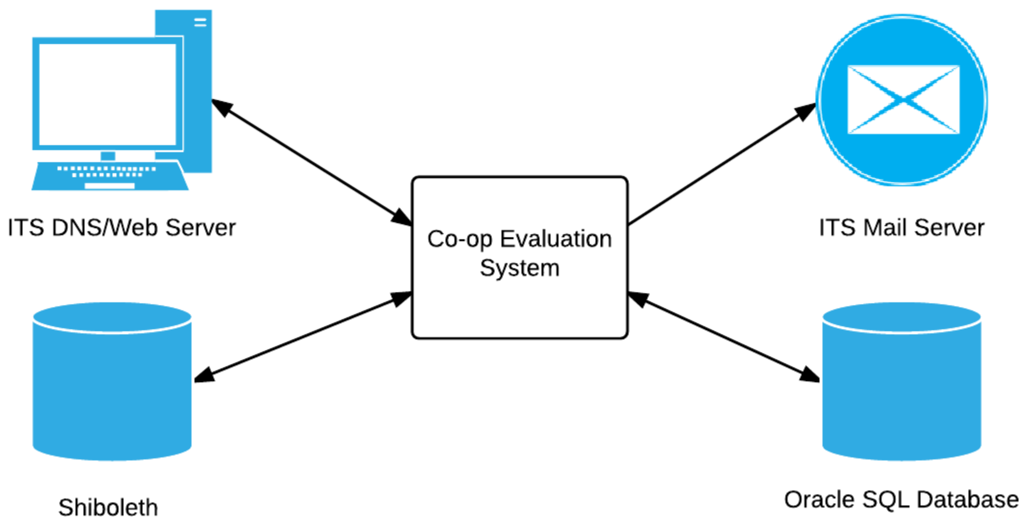

Figure 1. Context diagram of the Co-op Evaluation System

The above diagram outlines the major components of the overall system, subsystem interconnections, and external interfaces.

5.1.2 User Interactions

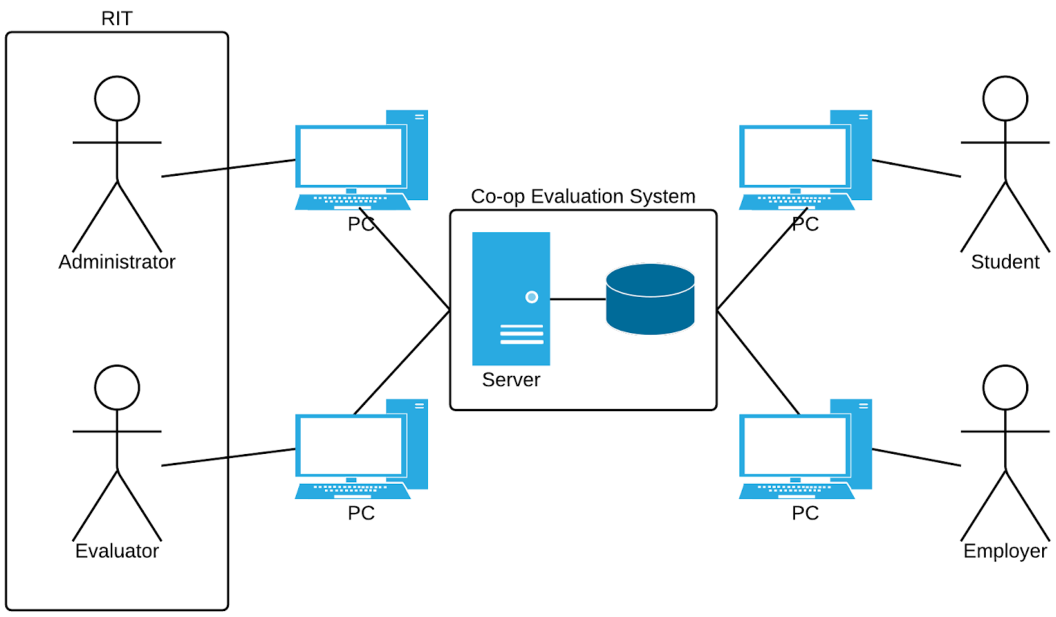

Figure 2. User interaction with the Co-op Evaluation System

The diagram above show a high-level view of the user interaction with the system as well as the interaction between technologies involved.

5.1.3 Data Flow

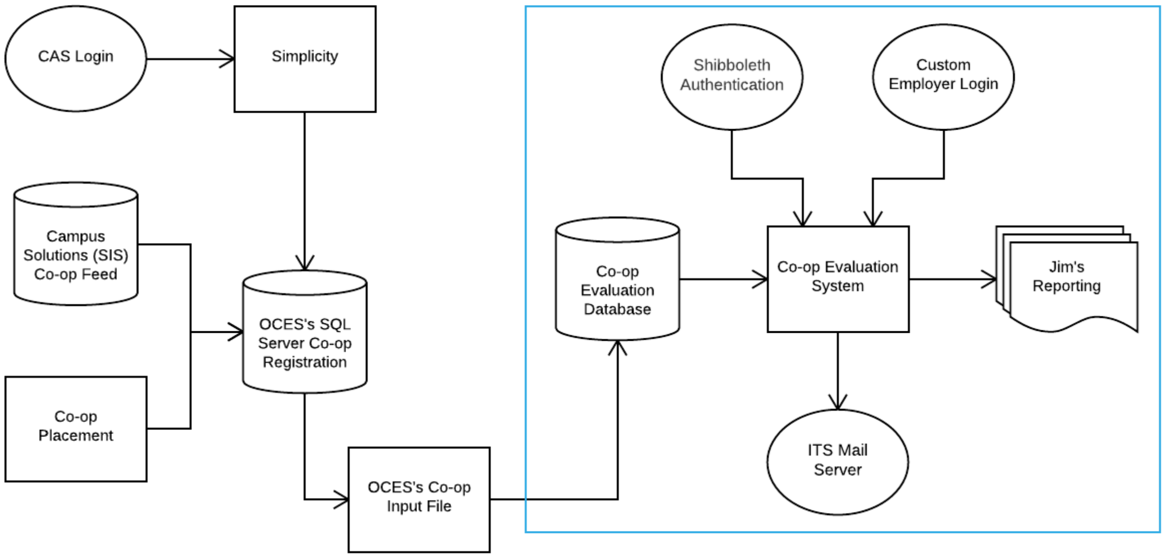

Figure 3. The flow of data into and out of the Co-op Evaluation System

The above diagram shows the basic flow of data into and out of the system at a high level. Our system and direct interfaces are represented inside of the blue container, with the outside entities depicting how data is created and imported into our system. Our system does not deal with data creation, however it is responsible for importing and storing it.

5.2 View Introduction

The two views that we have detailed are a Logical (Module) View and a Process (Component-and-Connector) View. The Logical View describes the layered structure of the system, while the Process View describes the client-server structure of the system. The Logical View shows how the the system is structured as a set of functional code units, or modules, whereas the Process View shows how the system is structured as a set of computational elements that have runtime behavior (components) and interactions (connectors).

The Logical View is a higher-level view of the system than the Process View. The Logical View shows the layers that compose the system, and the hierarchy of these layers. The Presentation Layer is encapsulated by the client, whereas the the lower layers are encapsulated by the server. The Process View also details the interactions between the client and the server and the specific components that comprise each.

Although it is standard to use a 4+1 View Model when describing a system architecture, we omitted the Physical View, Development View, and Use Case View from this document. At this time, we do not have the details of the system deployment, which would be outlined by the Physical View. We also do not yet have implementation details of the system, which would be outlined by the Development View. Both of these aspects of the system will be covered at a later date in separate documents. Finally, the Use Case View, which describes the various use cases of the system, has already been outlined in our Software Requirements Specification. Reference Section 4 of the Software Requirements Specification for more information on the Use Case View.

5.3 Patterns and Tactics

5.3.1 Architectural Drivers and Tactics

Usability

In order to maintain our focus on usability and iterate quickly on our user interface, we will be using the tactic *Separate User Interface*. This allows us to keep the user interface separated from the backend business logic and data source, thus enabling changes to be made easier or even for the user interface to be swapped out in the future, if another modernization is needed. The team will utilize a model-view-controller architectural pattern to accomplish this task. More details will be provided in Section 5.3.2 Patterns.

As the user will have the ability to undo or rollback certain actions (e.g. archiving an evaluation), the tactic *Support User Initiative* will be used to handle a reasonable amount of error correction.

Availability

Availability is an important architectural driver for the system, as the system must be available in order for students and employers to submit evaluations. Additionally, the system must be available for co-ops to be approved or denied, and for administrators to perform various managerial tasks. If the system is unavailable for any of these tasks, cascading delays may occur during any stage of the co-op evaluation process.

In order to achieve the best availability, we plan to use *Exceptions* for fault detection, *Active Redundancy* and *State Resynchronization* for fault recovery, and *Transactions* for fault prevention. Unhandled exceptions may put the system into a weird state; in order to avoid this, the system will gracefully handle *Exceptions* when a fault occurs to prevent the system from becoming unavailable.

In order to recover from a fault, the system will utilize *Active Redundancy*, which means that all system components will respond to events in parallel. As a result, all components will always be in the same state. This state is dependent on the response from one component, which is usually the first component to respond to a fault, and all other states are discarded. This tactic is often utilized with client-server configurations, as the downtime of systems using this tactic is usually only milliseconds. *Active Redundancy* must be used with *State Resynchronization*, as the component being restored must have its state upgraded before it is returned to service. The state of this component is synchronized using a single message containing the state that it must be returned to.

Finally, the system will use *Transactions*, which are bundles of sequential steps, to help prevent faults from ever occurring. *Transactions* help to prevent collisions among concurrent users, and help to prevent any data from being affected if one step in a process fails.

Maintainability

Maintainability is an important architectural driver for the system because the system will be maintained by ITS on regular basis once the system is deployed in production. This is also an important service at RIT so if defects arise, fixes must be able to be deployed as quickly as possible.

Being able to identify the area in which a defect is occurring is a major part of maintainability. This becomes easier to do when the system is not heavily coupled and each feature is modularized. When each feature is modularized it can be easier to pinpoint the root cause of defects because each feature can be tested independently to help find the issue.

Additionally, the system must be heavily documented and these documents must be organized and stated in an easy-to-understand manner. Once we have left, our documentation will be the only source of knowledge of the system, and will be used to understand the system when the system needs to be updated.

Testability

Testability must be determined in early stages of development. Due to the iterative nature of our methodology, regression testing will be one of the primary forms of testing conducted. Therefore, the tactic *Able to Stub/Mock* is highly valued to us. This tactic will allow us to create tests and quickly use them to test the system when changes are made or new features are added to the system.

Another tactic for testing our team plans to conduct is *Separating the Interface from Implementation*, which is a form of providing input and capturing output. Separating the interface from the implementation allows substitution of implementations for various testing purposes. This will also allow us to write tests without having to touch the interface itself.

5.3.2 Patterns

Service-Oriented Pattern

The service-oriented pattern compartmentalizes different features within the application. This pattern describes a collection of distributed components that provide and/or consume services. Service consumers are able to use these services without any detailed knowledge of their implementation.

This allows for easy maintainability, as each service can be altered without affecting the other services of the system. Additionally, each service can be tested independently of each other, which can be useful in pinpointing potential defects.

Domain Model and Data Mapper Patterns

The domain model pattern incorporates both behavior and data into an object-oriented model of the application domain. When using this pattern, the model of the domain is organized primarily around the nouns in the domain. The domain model is then separated from the database with the use of the data mapper pattern.

The data mapper is a layer that sits between the database and the domain model, which handles the loading and storing between the database and the domain model; therefore allowing both to vary independently. This separation of the database and domain model means that the domain objects do not have any knowledge that that database exists, and the domain model does not know that the data mapper exists.

The separation introduced by the data mapper pattern supports modifiability, as either entity can be modified independently of each other. The use of the domain model pattern supports increasing complexity and thus enhances extensibility as well.

Client-Server Pattern

Clients initiate interactions with servers, which provide a set of services. The clients invoke services as needed from those servers, and then wait for the results of those requests. The client is responsible for displaying and performing small updates on the data, while the server handles data management.

The client-server pattern supports modifiability and reuse, as it factors out common services, allowing them to be modified in a single location. This pattern also supports scalability and availability by centralizing the control of these resources and servers.

Model-View-Controller Pattern

The model-view-controller (MVC) pattern separates user interface functionality from application functionality. With MVC, application functionality is divided into three types of components: models, which contain the application data; views, which display the underlying data and interact with the user; and controllers, which mediate between the model and the view and manage state changes.

The MVC pattern supports usability, as it allows the user interface to be designed and implemented separately from the rest of the application.

6 Views

6.1 Logical (Layered) View

6.1.1 View Diagram

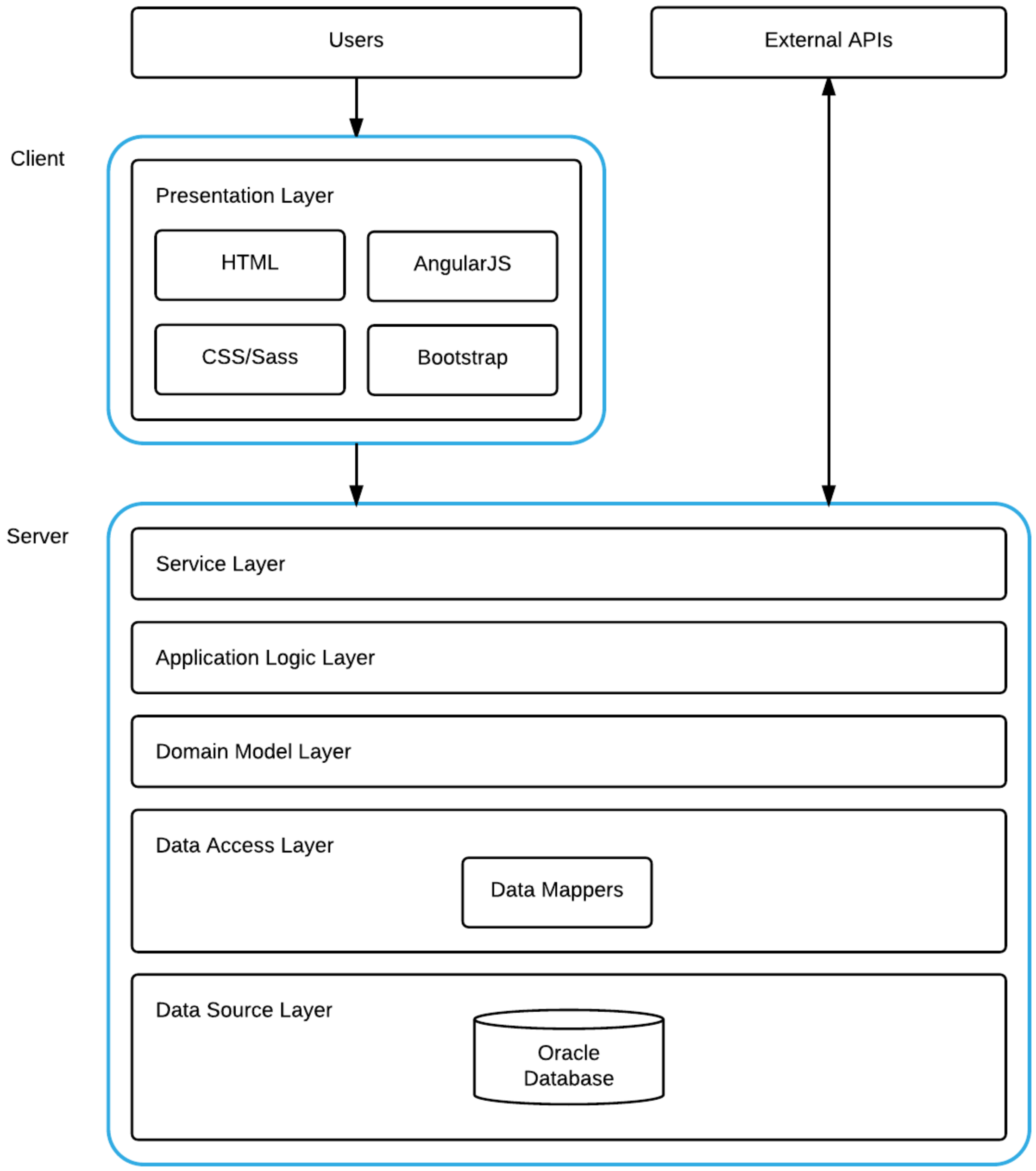

Figure 4. A layered view of the Co-op Evaluation System

Notation Explanation

Each layer is denoted as long rectangle with a black border. The smaller rectangles inside represent a range of elements contained in each layer. Those included are not intended to be a complete list of entities. The large rectangles with blue borders demonstrates the physical separation of the layers between the client and the server. More information on that division will be provided in Section 6.2.

6.1.2 Element Catalog

Elements

Presentation Layer

The presentation layer will provide the user with a graphical interface for interacting with the system. It will be composed of HTML, CSS, JavaScript, and other related files that will run in the user's web browser of choice. The main goal of this layer is to provide everything the user needs to complete their tasks in the system. The quality attributes that pertain to usability and front-end design will have the biggest pull on the presentation.

Application Logic Layer

The role of the application logic layer is to encapsulate the system controllers, which implement much of the core business logic. It will also serve as the connection between the user interface and the domain model, thus maintaining the separation of concerns.

Service Layer

The service layer is in charge of modularizing different features (services) of the application. Certain features, such as reporting, require access to external APIs; the service layer will be in charge of interacting with those external APIs. An individual service can also use the application logic layer to perform core business logic.

Each service will act as a facade, serving as a general interface for a feature that can be accessed by the presentation layer. These facades (interfaces) will provide the ability to manipulate or fix the code underneath each service without affecting the way in which that service is called by other layers.

Modularizing the services within the system will also make it easier to pinpoint potential defects, allowing for easy maintainability. Each service can be tested individually, which allows for good system-wide testability.

Domain Model Layer

The domain model contains all of the system's object representations of data in the system. This also includes associated methods for any objects that contain their own functionality.

Data Access Layer (DAL)

The data access layer contains all of the mappers to the data in the system. The mappers are in charge of the coordination of all communication between the objects in the domain layer and their corresponding tables in the database. This ensures that domain layer objects have no knowledge of the database, its schema, or any SQL interface.

Data Source Layer

All persistent information and any external API integration (e.g. SIS, Simplicity) make up the data source layer. This includes the Oracle database that will contain all of the data for the system. At this time there is no expectation for integration with external systems, but the system should be architected to accommodate such integrations in the future, as there are a few options on the table.

Relations

Presentation Layer to Application Logic Layer

The presentation layer represents the view of the system, and the application logic layer contains the controllers, which house the logic for the different roles. In following with the MVC architectural pattern, the controllers take information from the view and use it to modify or request related data in the model. This prevents the view from directly modifying the model, and instead has the view display changes to the data.

Application Logic Layer to Domain Layer

The application layer modifies the data encapsulated in the domain layer according to the business logic rules before it reports changes back to the presentation layer to be displayed to the user.

Domain Layer to DAL to Data Source Layer

The domain layer is the active representation of the information stored in the data layer. When it is time to store data according to the procedures defined in the upcoming detailed design stage, the changes to the domain layer will be pushed to the data source layer to create a persistent copy of the data to be stored across sessions through the DAL. All interactions between the DAL and Data Source Layer will indirectly be SQL queries in accordance with the ACID properties (i.e. an ORM framework will generate the SQL statements in most cases).

Interfaces

Interface Identity

The DataMapper pattern acts as the interface between the persistence layer, and the domain layer, the object model. It is displayed in Figure 4 as the DAL. It serves as a way to convert domain model objects to data formated for the database.

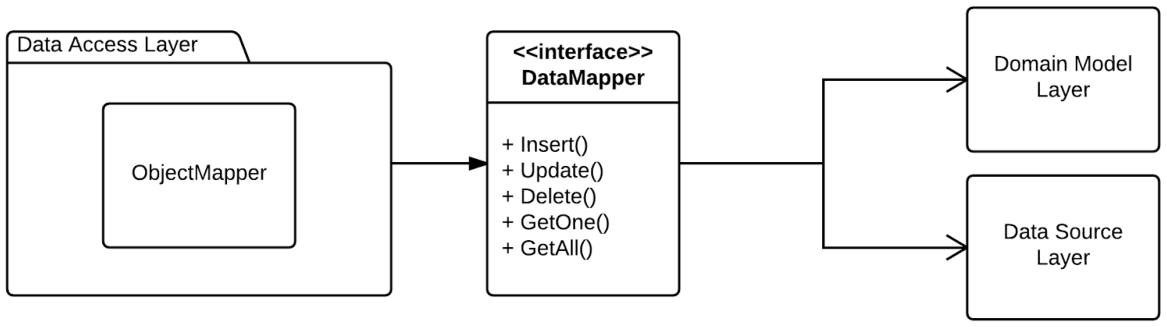

Figure 5. Diagram of Data Mapper Interface

The system also interfaces with External APIs for form and report generation. These APIs provide a set of methods for utilizing their tools that make up the interface between their system and ours.

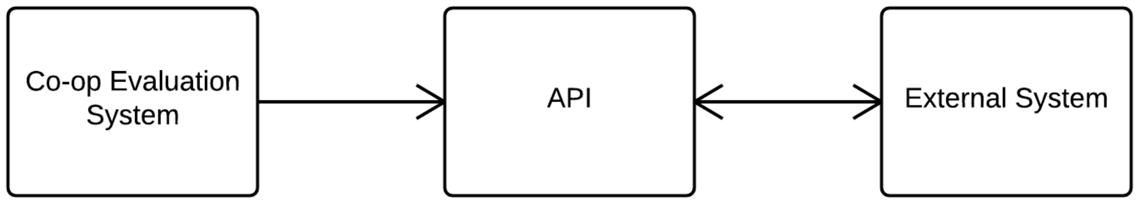

Figure 6. Diagram of External API Interface

Services Provided

Syntax

The mapper interface provides a way for the models in the domain layer to be inserted, updated, retrieved, or deleted from the respective table in the data layer. The object mappers in the data access layer implement the DataMapper interface as shown in Figure 5. The syntax for interaction will consist of SQL queries; however, these queries will likely be hidden behind a technology-specific framework for database interaction such as Spring for Java.

As for the external tools used for report and form generation, their APIs will act as our system's interface to their functionality. Our system will used their exposed methods to access the necessary functionality.

Semantics

Insertion will take the object, write it to the database as a new row, and perform any other necessary operations to save the data such as joins to update relation tables. Update operations will update the associated row in the database to reflect the changes made in the domain model objects. GetOne will select and return a specific element in the table based on an id or other identifying property, whereas GetAll will return all the elements in the table.

Delete will remove the specified row from the table and execute all the cleanup of associated relationships. Each of these operations should be atomic and satisfy integrity constraints.

For each method used in the external APIs, the system will call the method, sending all the necessary data along. The other system will take the data, modify it as requested, and return it in the desired format.

Data Input and Output

The user inputs their information through forms displayed in their web browser of choice. The client then sends that data to the server through the service layer to the application logic layer for processing, manipulation, and transformation before being written to the domain model layer. The information may be forwarded on through the data access layer to the data source layer to be written out to the database for more permanent persistence through the use of CRUD operations. Data may also be retrieved from the data source and manipulated by the application logic layer before being displayed on the view to the user. Of course, any and all transformation of data only occurs as necessary, and there may be a case where the unformated data is called for, such as displaying all the information in a given table.

Other Considerations

Exception Definitions

An `Object Not Found Exception` will be thrown if the database has a row that does not have a correlating model object. An `Element Not Found Exception` may occur if the user tries to update, select, or delete an element that has not been inserted into the table but has an existing correlating model object or if the element has already been deleted from the table. An `Invalid Relationship Exception` might happen if the relationships were incorrectly set at creation, were not properly updated to reflect changes made to the system, or were not removed fully upon deletion.

Quality Attribute Characteristics

The quality attributes supported by this interface include maintainability and extensibility. Maintainability because it moves database interaction functionality from the models to its own layer, thus making all three layers easier to read and maintain. This form of separation of concerns also improves extensibility because it is easy to change the mapper behaviors without editing the model, controllers, or database.

Design Rationale

The decision to use a mapper architectural pattern for this interface was made because it encapsulates the database interaction into a separate layer, thus making it easy to locate, update, and maintain. It also separates out this functionality from the domain objects and the business logic in the application controllers so that they are not calling a database interaction method associated with an object.

6.1.3 Rationale

The design above was created to reflect modern web application design standards. MVC, N-layered architectures, and client-server structures are the industry standard for designing web applications. Most major frameworks (e.g. MVC .NET, Rails, etc.) use the MVC architectural pattern as the core of their internal structures. The specific layers chosen are based on the purpose of the system and are commonly found in other similar web applications. They are meant to encourage separation of concerns and high cohesion in the implementation and detailed design.

6.2 Process View

6.2.1 View Diagram

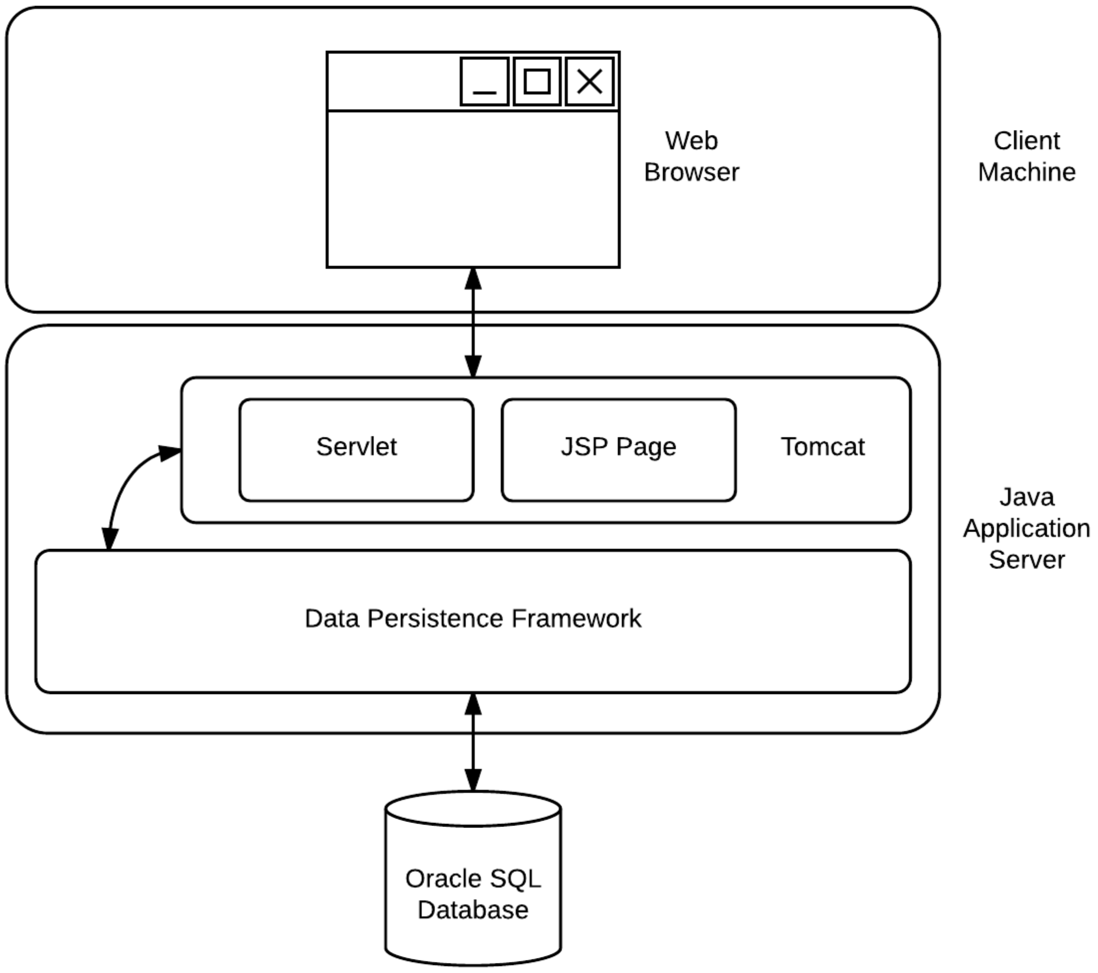

Figure 5. A client-server view of the Co-op Evaluation System

6.2.2 Element Catalog

Elements

Client

A client is a component that invokes services of a server component. Clients have ports that describe the services they require. In the context of the Co-op Evaluation System, the client is a web browser being used to access the system. The client makes HTTP requests using RESTful web services.

Server

A server is a component that provides services to clients. Servers have ports that describe the services they provide. The Co-op Evaluation System will be deployed on an Apache Tomcat web server. Tomcat implements the Java Servlet and JavaServer Pages (JSP) specifications from Oracle, and provides a Java web server environment for Java code to run in. The server provides web services through HTTP, a TCP/IP application layer protocol.

Request/Reply Connector

Request and reply connectors are data connector employing a request/reply protocol, used by a client to invoke services on a server. For the Co-op Evaluation System, the client and server will communicate using RESTful web services. The client will request information from the server using HTTP for normal requests, and HTTPS for secure transactions. In return, the server will respond using HTTP responses.

The server will communicate with the Oracle SQL database using a data persistence framework for mapping Java objects to database records. This communication takes place in the form of SQL statements and stored procedures.

Relations

Client to Server

The client and server communicate with each other using a request-response messaging pattern. The client sends a request to the server, and the server sends a response in return. The client and server use a common language of RESTful web services, so that both the client and server know what to expect. To handle multiple requests at once, the server uses a scheduling system to prioritize incoming requests from clients. The server also limits how a client can use the server’s resources in order to prevent a denial of service attack.

Interfaces

Interface Identity

Both the client and external APIs interface with the server through the service layer. A variety of services will be defined so that the user interface and external APIs may be swapped in and out without having to modify the business logic of the application. By defining a common interface for each service, communication of the user interface and external APIs with the system is kept consistent, and easily modifiable.

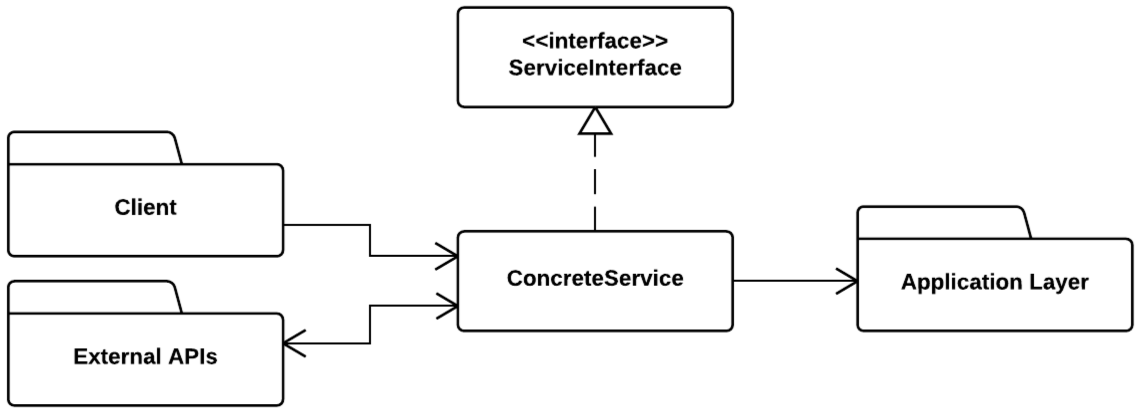

Figure 6. Service interface

Services Provided

Syntax

The syntax involved with using the interface requires any concrete services to implement the service interface. Furthermore, the syntax will depend on whether it is the client communicating with the server, or an external API. If it is the client interacting with the server, then the communications will be RESTful-based, and objects will be sent over HTTP/HTTPS. If the communications are taking place between the server and an external API, then data will be transferred as a serializable Java object.

Semantics

Although the exact semantics of the interface are undefined at this time, and will be defined during detailed design, we can provide a gist of what the semantics will look like. RESTful web services will be defined as functions such as get(), put(), post(), and delete(), one for each of the fixed set of four CRUD operations. Furthermore, additional functions specific to a particular external API may be defined as well.

Data Input and Output

As a result of user interactions with the web browser, the client sends data to the server for processing. For example, the client may output data from the submission of an evaluation to the server, which is then processed and stored accordingly. When the client requests data from the server, the server responds with data to be displayed to the user. This data is sent up as Java objects, and then converted into a format that is expected by the client, so that it may be displayed in the user interface.

The other flow of data into and out of the client-server model is from the server to the database, and from the database to the server. The server performs CRUD operations on the database to request data. In return, the server responds with the requested data, which is then mapped into Java objects through use of the DataMapper interface.

Other Considerations

Possible variability with this interface is the communication protocol used for communication between the client and the server, and any external APIs and the server. The exact communication protocol between any external APIs and the system may be dependent on the particular API. Additionally, instead of using RESTful web services for interactions between the client and the server, the system could employ an XML-based web service protocol, such as SOAP or WSDL.

In the event that an error occurs during communication, the exception will be handled gracefully by displaying a meaningful message to the user and logging the error internally. For HTTP-based communications that fail, the standard HTTP error codes, such as 404 and 400, will be used.

As mentioned earlier, the quality attributes of modifiability and extensibility heavily influenced the decision to use a service-orientated architecture. By using services to interact with external APIs, different external services may be swapped in and out with ease. Furthermore, the user interface may be changed without having to change any underlying application logic. Lastly, this service-based interface also supports extensibility, as new services may easily be built off of the base interface.

6.2.3 Rationale

The client-server pattern is a common architectural model for distributed operations. The server acts as a centralized system that can serve many clients. This pattern suits the Co-op Evaluation System well, as this is how the current system is configured. The system itself lives on a server provided by ITS, and users access the system through a web browser as a clients.

Another benefit of the client-server model is that it provides a separation of concerns. Client-side code, such as HTML, CSS, and JavaScript, are separated from server-side code in Java. Additionally, the client-server model allows for performance analysis and load balancing on the server side.

The client-server model also has a few drawbacks. The server can be a performance bottleneck, and can also be a single point of failure. However, since ITS is providing the server resources for the system, we are not very concerned that these two drawbacks will become major issues.

7 Acknowledgements

The Co-operators would like to acknowledge Len Bass, Paul Clements, and Rick Kazman, the authors of *Software Architecture in Practice*, as well as Dr. Hawker and Professor Kuehl for teaching Software Requirements and Architecture, from which we acquired most of our knowledge (and the template for this document). Furthermore, the development team would like to thank their sponsors, Jim Bondi, Kim Sowers, and the whole ITS team for their input and continuing support throughout the project. Finally, the team would like to commend Professor Malachowsky for all of his constructive feedback and endless support.

8 References

[1] Microsoft. (2009). “Chapter 16: Quality Attributes,” in Microsoft Architecture Application Guide, 2nd ed. [Online]. Available: http://msdn.microsoft.com/en-us/library/ee658094.aspx

[2] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 2nd ed., Westford, Mass.: Addison-Wesley, April 2007.

9 Appendices

Appendix A: Glossary

Term	Definition
CES	Co-op Evaluation System
DAL	Data Access Layer
Evaluation	A form that is currently being filled out by a student or by an employer
EWA	Enterprise Web Applications, a division of ITS
Form	A template that is used to generate an evaluation for a department
ITS	Information and Technology Services
Notification	An email message that will be generated and sent to students and/or employees.
OCSCE	Office of Cooperative Education and Career Services
Report	An aggregation of submissions used to display statistics
RIT	Rochester Institute of Technology
SRS	Software Requirements Specification
Submission	A form that has been completed and submitted to the evaluator
Status	The current state of the evaluation

Appendix B: Issues List

The team is using Trello to track issues; however, below you will find a high-level list of outstanding issues with this document. If finer detail is required, please reference the team Trello board, activity tracker, and/or Google Drive.

Number	Priority	Description