

QM Lab Course Project

Design Document

Group 3

Angela Stankowski (ans723), Bengin Lee (bel529), Connor Nettleton-Gooding (cwn973), Corey Hickson (crh208), Darwin Zhang (ddz369), James McKay (jlm012), Jordan Wong (jtw289), Mack Mahmoud (mtm464), Matt Hamilton (mch986), Michael Kelly (mlk121), Michael Ruffell (mar492), Mitchell Lau (chl929), Royce Meyer (brm979), Shane Williamsom (saw056)

February 7th, 2016

I. Introduction

For our design document, we are aiming to use the requirements and create a system architecture, plan out certain design model and design patterns we may want to use, and look at classes we may want, how our system will flow with a sequence diagram, and time estimates of different portions of the project.

II. Architectures

For our system, we are hoping to employ a number of architectures. Overall, we would like to have a broad architecture which we can use to describe generally how our system works. This section will look at different architectures and how we might employ them.

3-Tier Architecture

The most common architecture for a web application would be the 3-tier architecture, with a front end client that handles all of the visual and interactive aspects that users would handle, with a application tier handling all of the collaboration and ensuring each client is updating to the most recent version of the document, and a backend storing and managing all of the data we choose to keep (currently undetermined).

2-Tier Architecture

A 2-tier architecture would effectively host our entire application on the client side with minimal interaction happening with the database, except for storage. Our complete from end client would send relevant data back to our backend as needed.

Server-client

Alternatively, we could setup a client side application which would run the bulk of the work, while occasionally interacting with the server side. This would be very similar in style to the 2-tier architecture.

Model-view-controller

Lastly, is a variation on the model-view-controller. The primary focus of the application would be the view and its functionality, while being controlled in the back with a controller, and storing relevant information in our model as needed. The benefit of this would be possible greater utilization of the model, as we could break apart the actual app and have state information in the model.

A. Decision Matrix

To help determine which architecture we should pursue, we created a decision matrix as seen below.

| | | | | Rankings | | | | | |
|-----------------------|---------------|------------|------------------------|--------------|-----------------|------------|----------|-------------|-------|
| | Applicability | Modularity | Ease of Implementation | Adaptability | Collaborability | Useability | Security | Testability | Total |
| Weight (1->5) | 4 | 2 | 2 | 4 | 5 | 4 | 3 | 5 | |
| | | | | | | | | | |
| Architectures (1->10) | | | | | | | | | |
| 3-Tier Architecture | 8 | 7 | 7 | 8 | 8 | 9 | 9 | 8 | 235 |
| 2-Tier Architecture | 7 | 6 | 8 | 6 | 6 | 8 | 6 | 5 | 185 |
| Server-client | 6 | 5 | 6 | 6 | 5 | 5 | 7 | 5 | 161 |
| Model-View-Controller | 6 | 8 | 7 | 7 | 8 | 8 | 5 | 8 | 209 |

Figure 2.1: Decision matrix of different architecture possibilities for our system

We choose to make our matrix based off of applicability, adaptability, usability, and security to meet stakeholder requirements. We choose modularity, ease of implementation, collaborability, and testability to meet the team's requirements to have something we would be successful in using.

From this diagram, we can see that a 3-tier architecture comes out as the most likely to succeed architecture for our overall system. Considering the diagram, as well as our based on our research, we have decided to go forward with a 3-tier architecture for our overall system.

B. Architecture Diagrams

Using the 3-tier architecture, we have come up with a broad diagram of how our system will work. In our client level, users will be able to access the QM Lab web app, and interact with it through there. That client will work with the application level to handle input and push output back to the client.

The application will be running using the Google Realtime API, shown on the diagram. While our system does not include creating the Realtime servers, or Google Drive storage, they

have been included to help understand the scope of the architecture (beyond what we are creating). Additionally, we have included a path to our PostgreSQL server for any data storage or data handling.

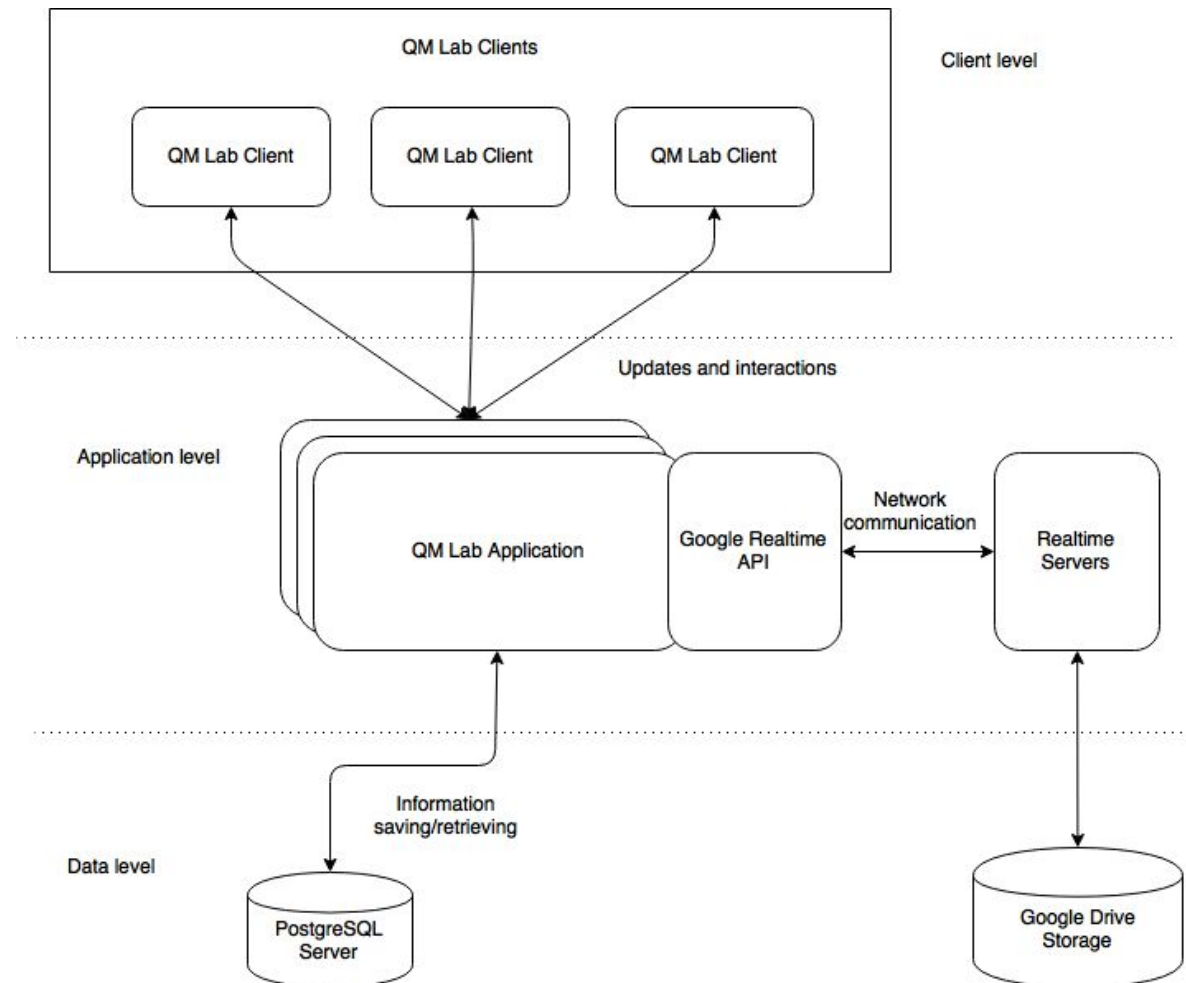


Figure 2.2: Overall diagram of our system architecture

III. Models

In our system, we are considering a number of different models to design our system with. Below are some of those patterns and models and what we might use them for.

A. Flyweight Design Pattern

A structural design pattern that focuses on using minimal memory by being an object that shares its data along with other objects. The reasons as to why this pattern was considered was because it was that it had the highlight of being very easily managed, similar to excel. This could apply to the different objects (eg. agents, maps, etc) for quick and fast access.

B. Singleton

A creational design pattern that restricts the instantiation of a class to one object. The advantage of this pattern over global variables is that you are absolutely sure of the number of instances when you use singleton, and, you can change your mind and manage any number of instances (control and flexibility).

C. Observer (Publish-Subscribe)

A behavioural design pattern that is used for one-to-many relationship between objects such as if one of the objects is modified, its dependent objects are to be notified automatically. By adapting the observer pattern, we gain the benefits of supporting the principle of loosely coupled designs between objects that interact and to allow sending data to many other objects in a very efficient manner.

IV. Classes

For our current design, we have briefly looked at classes we may need or classes we might want to consider. In future iterations of the design, we will go into specific detail and define the different classes and their usage (including function calls, etc.).

A. Login Page

A potential class that will provide the user with a login and a password in appropriate text boxes and a confirmation of this operation (a “submit” button). If a user already has an account they should be able to login or else they will have to create an account. The system should save this information and their projects created.

B. Main Page

The main page is available to the user following their login. The basic function of this page is a menu guideline. It will contain the ability(submenu) to create and edit pages and potentially to edit your profile or view other profiles (which would mean an addition of a chat system or a friend system).

C. Project

A section of the main with submenus to allow:

1. Project
 - Add new
 - View
 - Edit
 - Delete

- Create group
- 2. Import
 - Ability to import a project
- 3. Authorize to
 - Edit/view for other user
 - Give privilege levels to only view certain items for certain users
- 4. My profile
 - View asset(s) of project
 - Projects that are created/given to me

D. Edit

On creation or edit of a project from the main menu, the program will be sent to the edit page. The edit page will contain a menu of **Assets**:

1. Add
 - New page, duplicate page
 - UML, Sequence diagrams, etc
 - Text, pictures
2. Import
 - Pictures, fonts
 - Other projects
3. Edit
 - Change layout
 - Change font, colors, size of file
4. Export
 - The project
 - Possibly a .pdf, .png, .jpg filetype of the project

The user clicks on necessary item in the menu in order to perform operation. There should also be a way to see who else is currently working on a project at the same time as another.

Since this is an early rendition of our design document, the exact functions and classes or even tools have not been completely decided yet. Therefore, there are not functions with descriptions included (eg. pre, post, error conditions), instead are just brainstorming and loose descriptions of what we are currently planning for.

These are potential other modules and classes that compose them:

- Authenticator - way of identifying if an account's username and password match during login, or even for other cases.
- Database - again, undecided if it should be used, but it would need many classes and to communicate with other modules in order to store or retrieve the data the program needs as it's called by the client.
- Server (composed of classes) - What the program sends requests to and is responded back or given an error if the connection is unsuccessful.

- Log - a class that tracks down who did what and at what time, what errors were made, and at what time they were made.

We may also want to consider a class that tracks the different objects on the screen. Additionally, it would be helpful to have a class that manages our interface with the Google Realtime API.

V. Detailed Classes

The following is an addition for ID2.

For our software, we have begun creating a detailed design using a UML class diagram. It can be seen below, or at better resolution within the 'doc' folder (under design_doc_diagrams) on our GitHub repository.

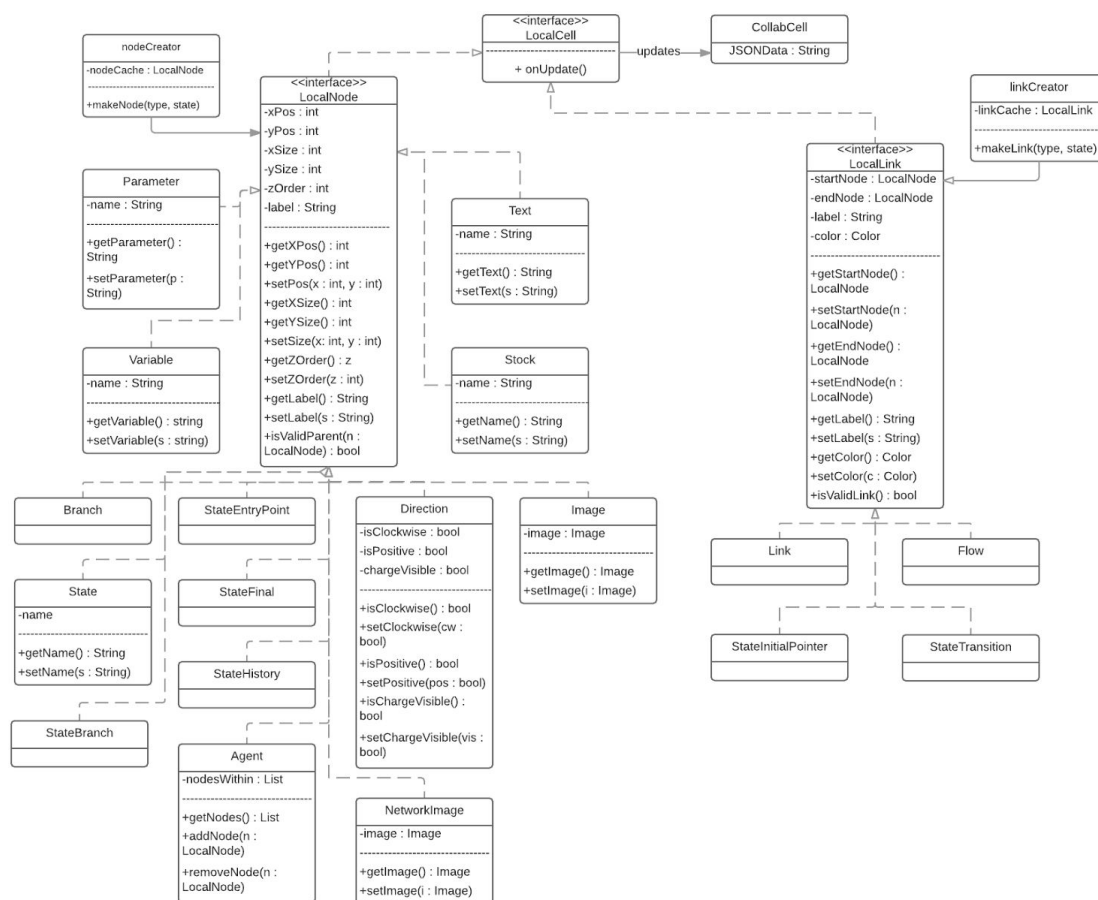


Figure 5.1: Detailed class diagram of the modelling objects

This detailed class diagram will be the contract with the development team on starting the actual software. With this, in the next deliverable, the development team will begin to implement the classes. For the next deliverable, we will complete the detailed class diagram (to include a controller for the application).

VI. Detailed Class Descriptions

The following is an addition for ID2.

This section describes the detailed descriptions of each of the classes from the detailed classes section. It covers key modules and ignores getters and setters.

| |
|--|
| LocalNode <<Interface>> |
| Input: None Output: None Layer: Data |
| Description: As an interface, this module will be an abstract class that gives explanations properties of many objects that will interface our system. It gives them the backbone of many classes within our system, however does not tell exactly how to implement the methods/attributes as this is an interface class. The objects implementing this interface are typically each one that is not a link (which is simply a connection between two objects that have implemented this interface) such as an image class, or a variable class. |
| Attributes: xPos::int - Keeps track of the node object's x position on the coordinate map yPos::int - Keeps track of the node object's y position on the coordinate map xSize::int - The size of the object in terms of its width x ySize::int - The size of the object in terms of its height y zOrder::int - The order of the object within the framework. The z axis, if you will, making the object able to go on top of one another based on its zOrder. label:String - The static label displayed beneath whichever object this is |
| Methods: pre: n is not null post: none error: to be determined IsValidParent(LocalNode n)::Bool - Checks if the node is a valid parent of node n (ie. an agent is a valid parent for images). True if it is a valid parent, false otherwise. |

| |
|---|
| LocalLink <<Interface>> |
| Input: None Output: None Layer: Data |

Description:

As an interface, this module will be an abstract class that gives explanations properties of many objects that will interface our system. This module is responsible for linking two objects that have LocalNode properties. Henceforth, the classes that will implement this interface are ones that draw connections between two node objects, such as a line or a flow.

Attributes:

startNode::LocalNode - The node object that initiates the connection (link)

endNode::LocalNode - The node object that receives the link

label:String - The static label which is visually displayed beneath the link

color:: Color - The color of the link/object

Methods:

pre: none

post: none

error: to be determined

IsValidLink()::bool - Checks if a link between the startNode and endNode is valid to be created

Direction

Input: None

Output: None

Layer: Data

Description:

A class that implements the LocalNode interface. It is a property that can be used to determine the direction of an object, and thus allowing rotations of the object, back and forth.

Attributes:

isClockwise::bool - true if the direction is clockwise, false if counter clockwise

isPositive::bool - true if the polarity is positive, false if negative

chargeVisible::bool - true if the polarity is to be displayed, false if not

Methods:

The methods of this function are precisely getters and setters. The only difference is that the getters check if the state of the object is in the correct form before returning with a bool.

Agent

Input: Takes various LocalNodes and holds them within itself (visually, as well)

| |
|--|
| Output: None Layer: Data |
| Description: A class that implements the LocalNode interface. The given class is able to create an “agent” object which can hold other objects relationally (who have implemented LocalNode). |
| Attributes: nodesWithin::List<LocalNode> - a list of LocalNode’s that are currently within the agent object |
| Methods: pre: n is not null post: a new LocalNode added inside the agent error: to be determined AddNode(LocalNode n)::void - when an object is inserted into the agent, it must be checked if implements LocalNode and then added to the nodesWithin list. pre: n is not null, n exists in the agent post: the node n is removed from the agent error: to be determined RemoveNode(LocalNode n)::void - when an LocalNode is removed from the agent, remove from the nodesWithin list. |

| |
|---|
| LocalCell <<Interface>> |
| Input: None Output: None Layer: Data |
| Description: This is the local copy of a shareable object. When it is updated, it updates the corresponding shareable object. |
| Attributes: None |
| Methods: pre: none post: the shareable object is updated to match the local object error: to be determined onUpdate()::void - updates the shareable object to match the local object |

| |
|--|
| CollabCell |
| Input: Receives update transfer from a object with LocalCell properties |

| |
|--|
| Output: Changes the JSONData of a property Layer: Data |
| Description: A class that is updated by objects with LocalCell properties. |
| Attributes: JSONData::String - the JSON... |
| Methods: None |

VII. Software Tools

The following is an addition for ID2.

Within this section, we list a number of tools which we are implementing.

Joint.js

A Javascript library which specializes in focusing on the interactions and visualizations of graphs and diagrams. Chosen due to its simplicity and ease of use in creating the visual entities as well as their connecting components in our project, thus, speeding up development.

Google Reatime API

Google Realtime API allows collaboration as a service for files in Google Drive via the use of operational transforms. The API is a JavaScript library hosted by Google that provides collaborative objects, events, and methods for creating collaborative applications. For example, if one wrote code to manipulate maps, lists, and your own custom JavaScript data model objects. When your code makes a change to the data model on behalf of some user, the data model automatically changes for all the users on the document, hence realtime.

Selenium

Chosen as the main testing program of our project. Selenium itself is a web application testing framework that allows you to write tests in many programming languages like Java, C#, Groovy, Perl, PHP, Python and Ruby. The advantages of using Selenium include that members in our group have experience with it. The current consensus is that it will be used for all tests.

jQuery, Backbone, and Lodash

Required dependencies of Joint.js. Will help considerably with the interface and framework.

VIII. Sequence Diagrams

Below is a general sequence diagram that shows the order of actions that take place when the user logs into the system and creates/modifies a project. As we go forward in the project, we will create more sequence diagrams to describe our system.

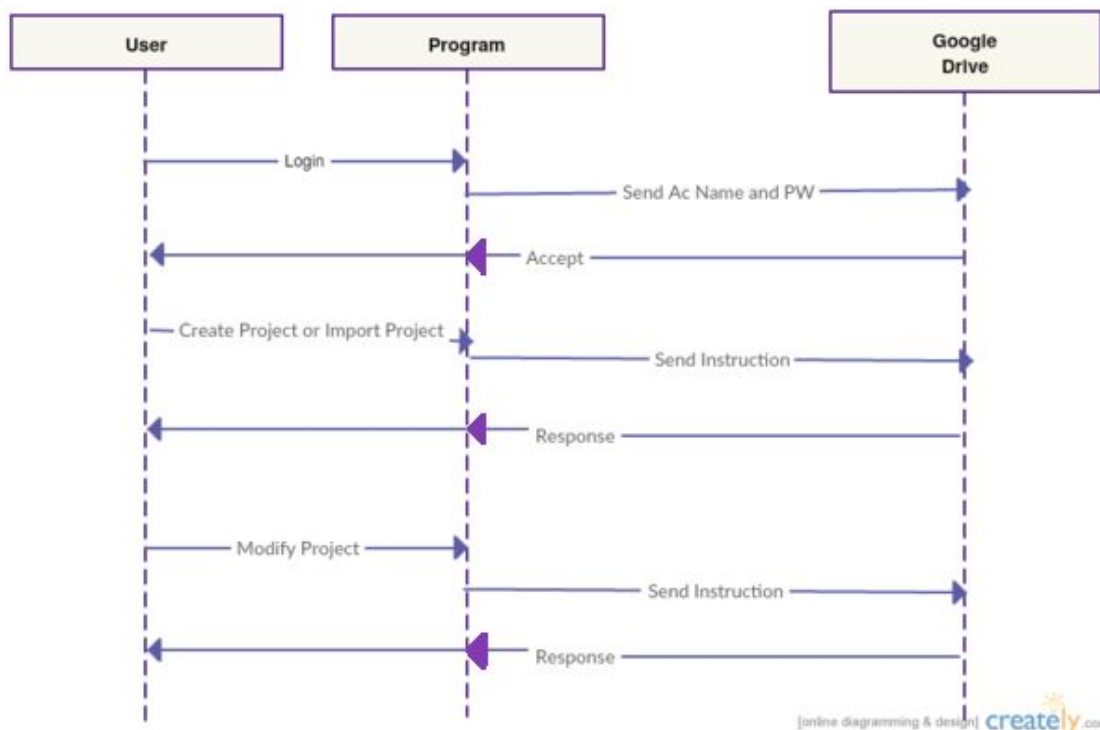


Figure 8.1: Overall sequence diagram of our system architecture

The following is an addition for ID2.

A general sequence diagram for the visual representation of how our 3-tier architecture:

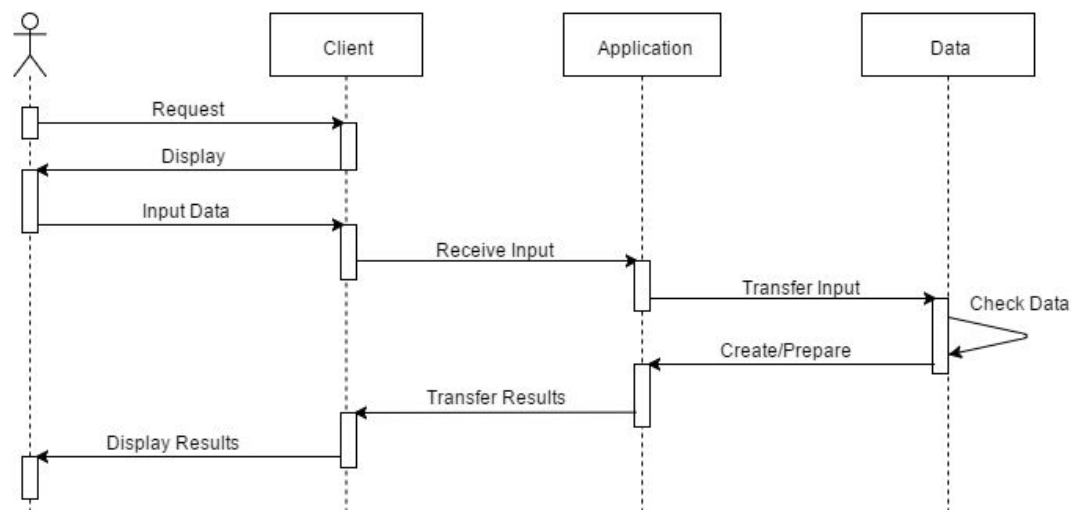


Figure 8.2: General sequence diagram of 3-tier architecture

IX. Time Estimates

To better understand the scope of the project, and how much development time each feature or requirement will need, we have created estimates based off of our requirements for how long we think each portion will take. This is summarized below.

| | |
|----------------------|--------|
| Requirement document | 7 hrs |
| Design document | 13 hrs |
| User manual | 8 hrs |
| Test planning | 10 hrs |
| Testing | 45 hrs |
| Debug | 30 hrs |

| | |
|---|----------------------------|
| Implementation (must have features): | $10+10+8+5+8+8+6 = 55$ hrs |
| Support multiple users (OT) | 10 hrs |
| To create, save, and import | 10 hrs |
| Support system dynamic diagram, stock diagrams, agent, picture and text | 8 hrs |
| One of a small number of shapes and images | 5hrs |
| Create variables and links between components | 8 hrs |
| A link for the project that for sharing | 8 hrs |
| Able to create agent objects which can hold other objects relationally | 6 hrs |

| | |
|---|-----------------------|
| Implementation (should have features): | $10+10+5+5+5= 35$ hrs |
| A login system | 10 hrs |
| A log of who edited what at certain points - potentially save state/backups | 10 hrs |
| Power to select a set of the diagram | 5 hrs |

| | |
|--|-------|
| Ability to resize and color the elements for a diagram | 5 hrs |
| Change the font/style of text within components | 5 hrs |

| | |
|--|---------------------------|
| Implementation could have: | 7+10+20+20+5+13+5= 80 hrs |
| Customized components such as a user specified picture is added and named to the elements available for a certain project/team | 7 hrs |
| Customizable overlay for the software | 10 hrs |
| A chat system - private chat | 20 hrs |
| General templates to guide users | 20 hrs |
| Different styles of components (different looks available for say a UML class diagram) | 5 hrs |
| Undo function possibly based on history | 13 hrs |
| Support of video maybe a slideshow/add pages to a project? | 5 hrs |

X. Conclusion

In conclusion, we have decided that for our overall system we think the 3-tier architecture will provide the best architecture for our web application. We have defined a number of models and patterns we believe will come in handy during the implementation and briefly touched on the classes we may want.

We modelled our system with sequence diagram to describe the flow we expect to have, and last have laid out our predicted time outcomes for various portions of the project.