**Changes that need addressing:**

\*Remove or better explain the interface actor in requirements document\*

\*Print functionality in the requirements?\*

\***Need document/illustrations that give sense of how overall system hangs together and the basic way in which different subsections work, both to spread knowledge around within the team and for the markers.**\*

\*"Resources Estimates", there is a column "Time Spent" that appears to list much time in many things, (mocking, smoke testing, system testing, alpha tests) that have no evidence in the codebase)\*

\*Print functionality missing from requirements, despite being incorporated in NDO doc\*

\*This seemed to NDO mysterious: I can't understand what this means or how it relates to the requirements "Interface:​ The interface is an abstract concept that reacts with our system. For instance, If

the user was editing the diagram the interface would be specifically WHAT the user is doing.

The interface might create an object or link two objects together depending on what the user

is doing within the program at any given moment. So in a general sense if the user was

simply adding, editing and creating diagrams, the interface would be what is happening or

how it is being done. "\*

**----------------notes from last meeting------------------------------**

**Show external libraries and what we are using them for (Design team)**

**Extensively using joint.js and google real time api**

**Make a diagram, with a glance (what tools are taking care of what parts of the project)**

**Tools we are using (get help from the others):**

**Setting up google real time is complicated:**

**make a diagram to show how we do it**

Joint.js is extremely cheap to create and destroy objects, thus -> flyweight is no longer needed

Joint.js will handle most events (but stuff like resizing, we will need to handle that)

Joint.js has a class called paper that handles event

**Notes from class:**

- Need to specify pre and post conditions for functionality

- Divide up the system into multiple classes

- If you are putting the work to do this and you must check which methods depends on one another

- What contracts (what things it needs to do its jobs)

- What are the external resources (like the database must be populated)

- What is the job of the code (need to specify those)

- A reliance on getters and setters

- They are like something that comes along automatically

- We want more of the core domain logic rather than getters and setters

- Interface: The interface to the program (GUI or maybe a console) - another use of interface is the interface in the implementation (we are talking about the contract associated with the system (api and the promises that it makes))

- Separate the contracts of how it all happens - in OO try to separate the interface by hiding implementation details (users of an instance cannot access the internals of a class - make private a variable (this is because it may be changes in ways that it shouldn't be changed)

When we use the calls on real time, it gets sent to some server

- Everything is JSON (a method that is built into joint.js)

all global variables are in one file to easily find and they can all be accessed if they are initialized

Separate for the .css

The client side of the application is made through html, css, javascript.

The first time you create the document it creates a model that everything is stored in (you have to grab that and hold on to it) and you have to use that to update the model. Creates the document and model (save it as a variable to update the document). This is a result of one of google's built in functions (This is in the authorization part of the js).

**CHANGES:**

--------------------------------------------------------------------------------------------------------

**Overall System:**

Client: HTML, CSS, Webix.js, Joint.js (jquery.js, lodex.js, backbone.js)

Application: Google Realtime API, Google Realtime Server

Data: Google Drive Storage

**System Overview:**

QM-Lab is a collaborative, qualitative modeling web-app. It can largely be broken

down into 2 pieces. It allows collaboration by using Google's Realtime API, and

easy, user-friendly modeling through Joint.JS

Looking at just the modeling, Joint.JS allows the diagrams to be built with a largely

MVC pattern. The actual diagram's data is stored in a model called the "graph". Each

element of the diagram is stored in the graph as "cells", each with its own

self-contained information. These elements can be further sub-typed very broadly

as "links" and "nodes".

The user can view and interact with the diagram through what Joint.JS calls the "paper".

The paper allows the diagram's graph to be attached, and then renders all information

about the graph in the paper's "view". Depending on the size of the graph versus that of

the paper, some elements may not be rendered (or, rendered outside the view).

Any element currently being rendered by the paper allows for the user to easily click on it

and begin dragging around the paper, as well as linking and delinking with just a click.

The collaborative part, using Google's Realtime API, reacts to "change" events in Joint.JS.

Any time something in the diagram is updated, the change gets saved in the collaborative

model. Then, Google's Realtime API sends an update to all connected collaborators with that

change. Google's Realtime API then informs Joint.JS of that change, before Joint.JS updates

the given diagram element accordingly. From a technical standpoint, the data being transferred

by the Realtime API is simply JSON.

**Tools update:**

**Additions:**

Joint.js: The library specifically deals with the creation of the logic behind the graphs, nodes, and links. It also handles events(mouse click) and resizing components of the objects.

Google Realtime API: We will call upon the realtime API for a lot of things: including finding and loading items - adding new items, attaching listeners to the objects so they can act in “realtime”, authorizing access and giving authority for certain projects, creating/saving/editing with interactions with the server/database.

**New:**

Webix.js:

Webix is a JavaScript and HTML5 framework which was chosen to be used in the project. The library simplifies creating desktop web applications with highly responsive user interfaces. Webix is being used for the general UI and the layout of the system (separating the UI components into manageable/sensible format), such as the setting of the toolbar.

Google Realtime Server, Google Drive Storage:

Although we do not directly access or call upon either of these tools directly, we do use them through the google realtime API. When we call upon the realtime API, this sends a message to the server and, if needed, to the Google Drive Storage. Therefore, allowing users to interact and collaboratively work on projects online in real time(through the API and server) and save/retrieve the data for the projects(through Google Drive Storage).

CSS, HTML:

HTML currently is currently only used for the index page as the view. CSS used for stylization. Each .js library has their own css.

Change (ID3): Flyweight Design is not longer needed as Joint.js has made it extremely cheap to create and destroy objects.

Change(ID3): Changes the detailed class descriptions were not updated in this document, however they are shown in the actually implementation (large descriptions with pre and post conditions on all the functions).