

---

## Design Studio 3: Sketch to Create

### Team Members

|                  |          |
|------------------|----------|
| Robert Maldonado | 56105664 |
| Ankit Jain       | 96065117 |
| Archita Ganesh   | 21387180 |
| David Bui        | 52854769 |
| Alfredo Gaytan   | 81661851 |
| Caleb Chu        | 30286026 |

# Table of Contents

|  |    |
|--|----|
| <b>1 - Global Design Decisions</b>                                     | 2  |
| 1.1 - Primary Audience   | 2  |
| 1.1.1 - Stakeholders   | 2  |
| 1.2 - Goals  | 2  |
| 1.3 - Assumptions  | 2  |
| 1.4 - Constraints  | 2  |
| 1.5 - Ideas  | 2  |
| <b>2 - Application Design - Everyone</b>                               | 4  |
| <b>3 - Interaction Design - Ankit Jain &amp; Archita Ganesh</b>        | 7  |
| <b>4 - Implementation Design - David Bui &amp; Caleb Chu</b>           | 11 |
| <b>5 - Architecture Design - Robert Maldonado &amp; Alfredo Gaytan</b> | 18 |
| 5.1 - Database Type  | 20 |
| 5.2 - Database Schema  | 20 |
| 5.3 - Database Version Control & Management System                     | 22 |
| <b>6 - Design Processes</b>  | 25 |
| <b>7 - Reflection</b>  | 26 |
| <b>8 - Design Artifacts</b>  | 26 |

# 1 - Global Design Decisions

## 1.1 - Primary Audience

- Industry Level Programmers and Testers

### 1.1.1 - Stakeholders

- Students
- Software companies
  - Consumers of the products made by the company

## 1.2 - Goals

- The user should be able to draw over the code, which will contribute to the generation of test cases
- To auto-generate test cases with the help of drawing feedback
- To allow other developers and students to collaborate on projects
- The system overall should be easily usable for software developers of all levels

## 1.3 - Assumptions

- The person testing the code knows how the code works
- The person testing the code knows how testing works and how to implement them
- Our system can observe the files in a file system in real time, allowing user to edit the code with whatever IDE they want, while our system can see the code updates in real time
- The software is running on operating systems that support touch, such as Windows, Linux, Android, and iOS.
- The system would know which statements can be tested based on whether the variable is static or dynamic

## 1.4 - Constraints

- The user must be able to sketch to create test cases.
- The test cases will be constrained to the parameters the users input for the functions
- Code will be static and cannot change in the middle of sketching

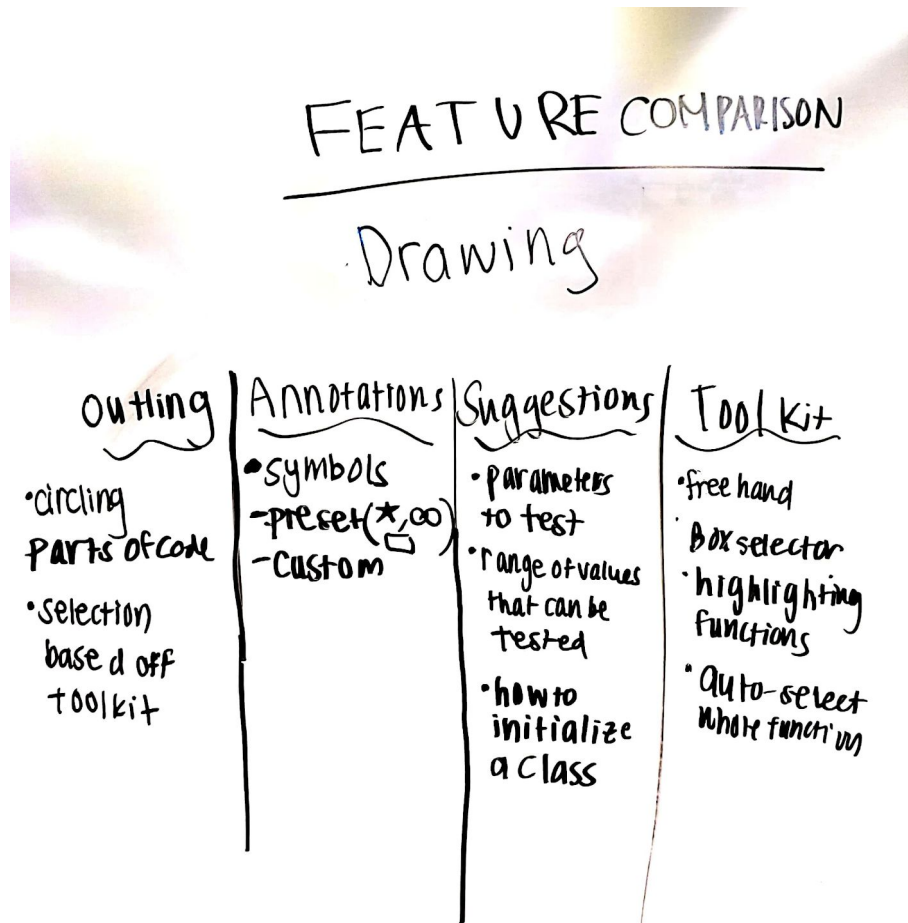
## 1.5 - Ideas

- Annotations
- Drag and drop
  - Code selection (through circling)
- Generating Tests
  - When generating a test for a function that is a part of a class, the system needs to use the default constructor to initialize that object
- Start Screen: “What kind of testing are you trying to do”
  - Unit Testing
  - System Testing

## 2 - Application Design - Everyone

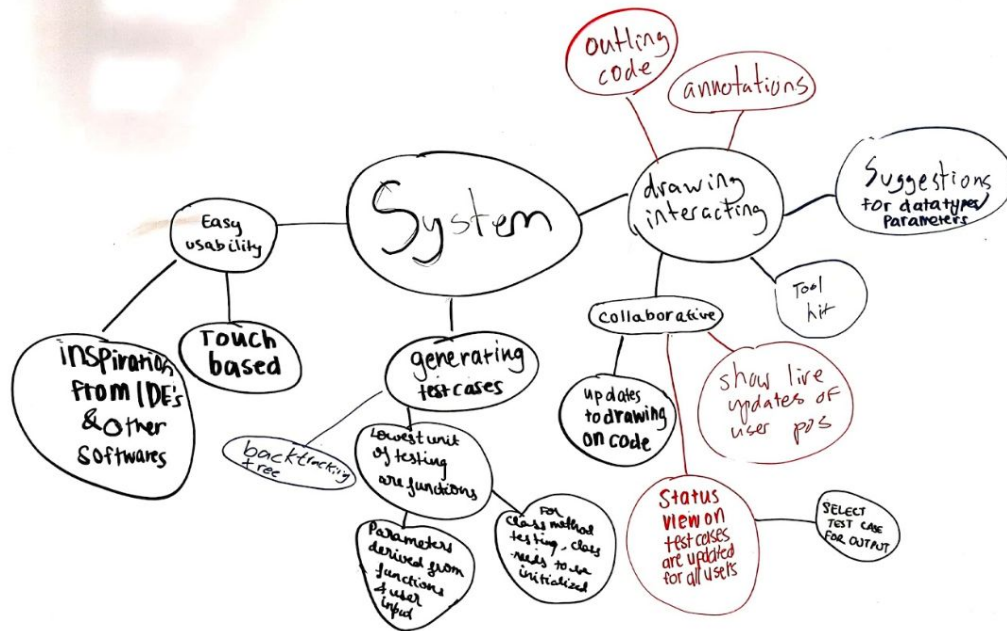
### Analysis - Feature Comparison

To determine which features to include for our final application design, we decided to break down the goals to specific features we could use to implement in our application design and how the interface would look like. To get the basic layout on how to generate test cases using a touch based system, we decided to break down possible features that the system could integrate for this touch based software. The picture below breaks our ideas for how the user would draw over the test cases to tell the software the user wants to test a specific function. This process of feature comparison helped with the process of generating alternatives. In turn, it helped with determining which specific features we wanted to implement: circling parts of code, suggestion buttons for arguments, and a toolkit to help with selecting sections of the file.



## Synthesis- Mind Mapping

As to get a clear view of what features our application would offer, we employed the use of mind maps where we basically listed all our ideas associated with the system and connected the ones which had a certain degree of similarity.



## Our Chosen Approach

Our software includes 3 primary tabs to assist in the process of software testing: Main, Test Cases, and Testing Code View.

The main view is where users will interact with each other. This view is the first tab that will be presented to the user when a sketch is started. There are four main components of the Main View the sketch, the navigation bar, the suggestion bar, and the test case preview.

When looking at the sketch, the user is presented with a view of code that can be drawn over, using the tool they have selected. The user uploads a file of source code that they want to use as a sketch, this code is used to create a new sketch that they can draw over, the source code cannot be changed once drawn on. If the user wants to upload a new file of source code, it will generate an entirely unique sketch for collaborators to draw on.

The navigation bar provides users many settings for their document. It provides four options on how they want to select (code, free-hand, box select, highlighting, or auto-select), an

option to share the sketch with another user using their email and shows how many users are on the document at a given time.

The suggestion bar is a view that is shown above a function once a user has selected a function that they are going to start using, it provides users the opportunity to select input options for each parameter of the function.

The test case preview is a view that is shown on the right of the document once a function is selected. It gives the user an option to enter some parameter values, but also gives the user the ability to see a preview of the test case that is being generated for that function. These test cases are based off the range of values that the user has selected or entered. There is a box at the end of each test case that is green if the test passes and red if the test fails.

The test table view is a display that helps the tester view all the test cases generated for the whole program. The purpose of this tab is to easily view all test cases in one place as an organized tabular format that can be scrollable. This tab shows the testing file name on the top with the total number of test cases, so the tester will know how extensive the testing was. The tests are then grouped by functions. As a way to organize and understand the tests, our system allows the users to enter a description for that function. Each test case has a specific id that is displayed in the format(Function Number\_ Test Case Number). The test name column displays the actual test with the specific arguments passed into the parameters that would generate the test results. The test data column is another optional column for testers to put in descriptions for each test to differentiate what each test case is about. The table view then displays the expected results which the user must input and the actual results is the test results generated. The status displays whether a test case has passed or failed. Since, collaboration is key to the Sketch to Create Test Generation software, we have placed the collaboration tool on this page as well. The document can be shared and the users can view which test case their collaborators are on with the colored circles displayed on table.


The test code view is the third tab in our system. This view shows the code behind the test cases in a test suite format, like a java unit test suite. If a specific test case is clicked on the test table, the user will be taken to this test code view to view the mechanics behind the test case. The user is also given power to customize the code in the tests, just in case they wanted any minute adjustments to the test cases.

### 3 - Interaction Design - Ankit Jain & Archita Ganesh

Interaction Design is associated with how the user interacts with the system, as to what the user sees and experiences upon usage of the system.

#### Analysis- Persona

In order to understand, which types of users would use our system and the purpose/primary goals, we created our basis of the interaction design based on two persona. These personas are a fictitious representation of user groups who would directly interact with the system and what their needs might be. The persona breaks down our audience of industry-level programmers to programmers who are more experienced and those who are in the beginning phases of their careers. The first persona of Bob Rogers is a representation of older programmers in the industry who don't like the tedious process of testing, but enjoy programming. Our system is designed to be user friendly, easily usable, and collaborative for programmers who want to make the testing process easier to use.



## PERSONA

**Name:** Bob Rogers  
**Occupation:** Software Developer  
**Company Size:** 100  
**Education:** M.S in Computer Science and Software Development  
**Age:** 47

**Background**

Bob is a software developer for the past 23 years in the Silicon Valley and he loves his job which involves programming and offering solutions to various technical issues, but he despises testing. He would love to have a collaborative tool which aids him during the testing process.

**Needs**

- Fast and Easy to use tool for testing
- Collaborative Tool for Testing
- Immediate Feedback and Responsive Tool

The second persona of Olivia Johnson describes a person in their early stages in the tech industry and finds interesting problems with the current process of testing such as sharing and collaborating. This pushed us to make an easy-to-use software that is collaborative with the



ability to share the code and test cases with colleagues, while being able to see where they are located on the document.



# PERSONA

**Name:** Olivia Johnson  
**Occupation:** College Student and Quality Assurance Intern  
**Company Size:** Startup (up to 25 people)  
**Education:** B.S in Software Engineering  
**Age:** 20

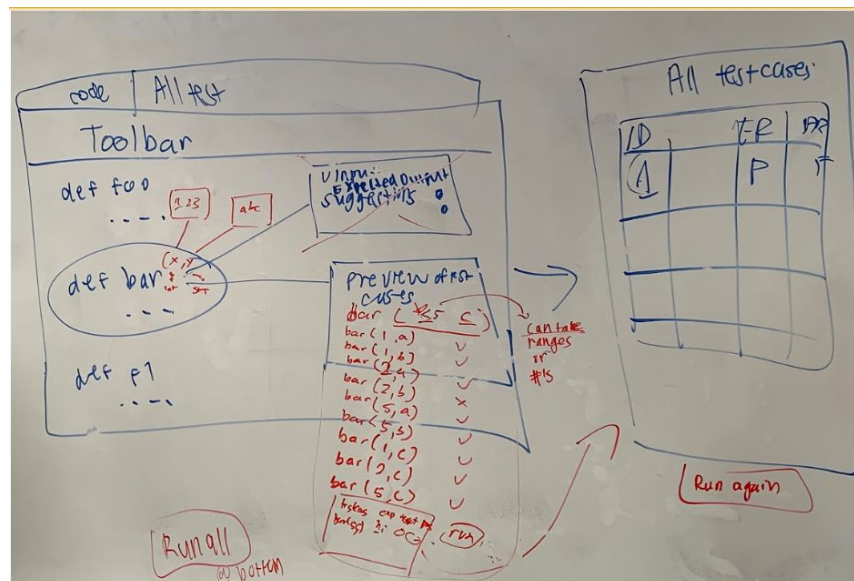
**Background**  
 Olivia is currently a junior at University of Sciences. Atlanta. She loves to study about testing and quality assurance and hopes to build a career based off her interests. She is currently working a startup as a QA Intern and has found extremely difficult to share her test cases and results with her colleagues and is looking for a software tool for the same. She uses an iPad for all her needs such as studying, note taking, coding etc.

**Needs**

- Easy to use
- Collaborative tool
- Touch Based Application

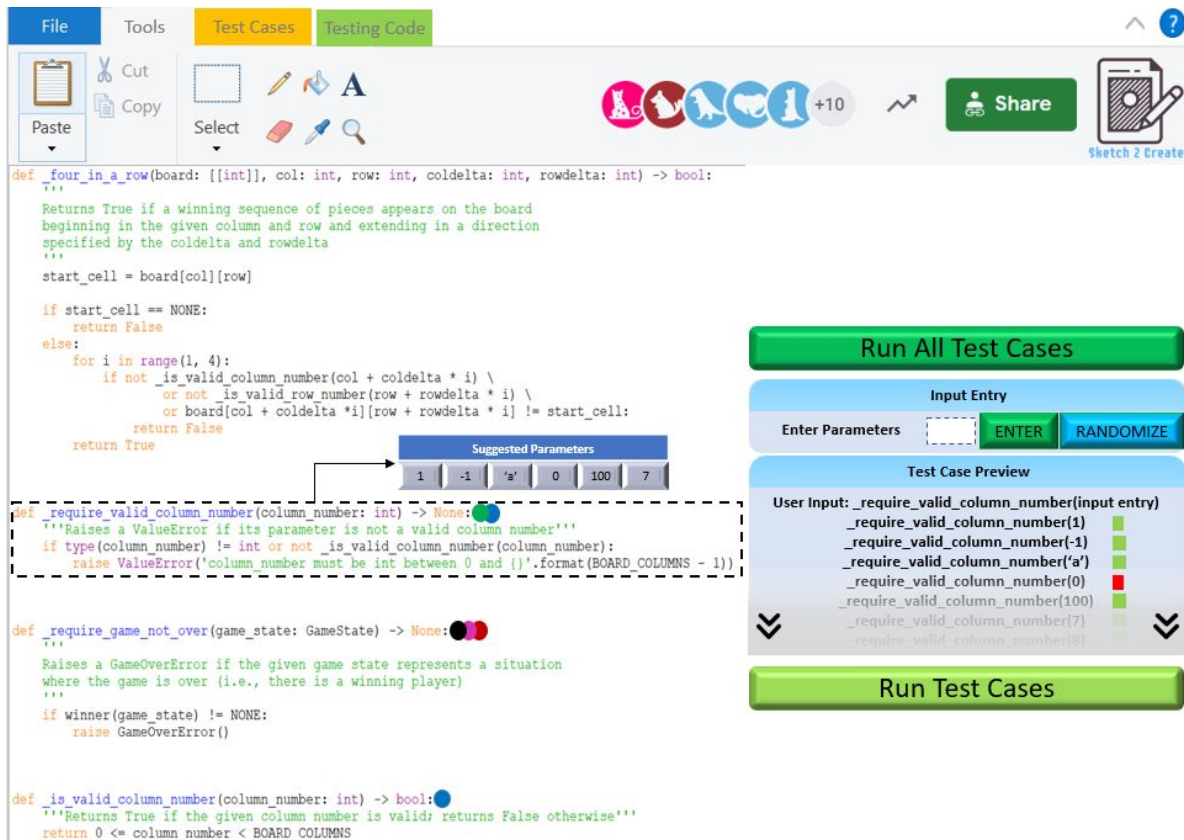
## Synthesis- Design/Making

To formalize our thinking and design process, we decided to display the idea of how the interaction would be represented through the use of the design method: design/making with mockups. This first image is the initial plan for the mockup which we first drew on the board before making a proper digital mockup.



The first mockup shows a representation of our main view with the code and how test cases are generated using the touch based software. The user will be able to sketch or draw over the code to circle which part of the code they would like to test. After circling the function(s) they want to test, a list of suggested arguments for the parameters pop up for function, which the tester can click. These arguments are used in the test cases that are displayed in the test case preview. The other way the test cases gets its argument is with the user typing into the box “input entry” the desired input for the test case. This scrollable test case viewer shows the test cases generated from the clicking of suggested parameters and user’s input entries. The results of the test cases (green box-passing and red box-failing) are generated after the user presses the run test cases button (for only that function) or run all test cases button (for the entire program).

The second mockup represents the test cases in a tabular format where the user can view the entire program’s test cases. The user has the options of entering text for the description, test data description which summarizes what the test case is about, and the expected results. The other parts of the test table view are automatically filled in by the system. This design allows users to customize certain descriptions and understand the tests through this refined organization of crucial testing elements.



| <div> <div>FileToolsTest CasesTesting Code</div> <div> Testing File: connectfour.py<br/> Number of Test Cases = 400<br/> Description: Verifies validity of column number entry </div> <div> <div>Share</div> </div> </div> |                                    |   |                  |                |        |
|--|------------------------------------|---|------------------|----------------|--------|
| Test Case ID   | Test Name                          | Test Data                                 | Expected Results | Actual Results | Status |
| 1_1  | _require_valid_column_number(3)    | Column number is between 1 and board size | Valid            | Valid          | PASS   |
| 1_2  | _require_valid_column_number(1)    | Column number is 1 [minimum value]        | Valid            | Valid          | PASS   |
| 1_3  | _require_valid_column_number(7)    | Column number is 7 [maximum value]        | Valid            | Valid          | PASS   |
| 1_4  | _require_valid_column_number(-1)   | Column number is less than 1              | Invalid          | Invalid        | PASS   |
| 1_5  | _required_valid_column_number(100) | Column number is greater than board size  | Invalid          | Invalid        | PASS   |
| 1_6  | _require_valid_column_number(0)    | Column number is zero [null value]        | Valid            | Invalid        | FAILED |
| 1_7  | _require_valid_column_number('a')  | Column number is not of type 'int'        | Invalid          | Invalid        | PASS   |
| Description: Verifies whether game is over   |                                    |   |                  |                |        |
| 2_1  | _require_game_not_over(NONE)       | Game State is null name                   | Invalid          |                | PASS   |

## Evaluation- Cognitive Walkthrough

In order to evaluate how our system interaction would work and assess what to change, we decided to do a cognitive walkthrough of each interaction the user could possibly have so we get insight as to how a user would experience the application.

### Cognitive Walkthrough for Sketch-to-Create:

1. User opens the application on a touch-based device such as a tablet
2. User is represented by a black circle and sees the code uploaded by a peer or uploads a code file from file option and selects 'Share' to share the test cases and code with his/her peers.
3. User selects a function by means of the 'select' option and drawing over the desired function with tools offered by the navigation bar present.
4. Upon selection, the software displays suggested parameters for the basis for test generation which can be selected, entered by the user or randomized. Test case preview gives insight to expected test case results. User can swipe down to view more test cases in the preview.
5. User selects either 'run all test cases' which runs test cases for the entire file or select 'run test cases' which runs the cases based on entered parameters or suggested parameters. This leads the user to the Test Cases tab.
6. User is now able to view different test cases and how they were run and if there were any occurring errors and also if any test cases had an issue upon running.
7. User can save the test cases code by going to the 'Testing Code' tab and then selecting the 'Save File' option.

## 4 - Implementation Design - David Bui & Caleb Chu

For the implementation design, we chose pseudo code as our main design method because we thought it was the best way to represent our system, as it helps communicate the system's implementation details, such as its data structures. Each action, data structure, object, etc were carefully named with specific naming conventions to make design more readable and easy to understand. In addition to the naming conventions, we made sure that each part in the pseudo code is understandable with comments at areas where we thought needed extra explanations.

We organized this structure into two main classes: the test case class, and the State class. The “test\_case” class is a test case object that contains five attributes: Test ID, newTest, Expected\_result, Actual\_output, and status. Each of these attributes are displayed to the user after generating each test case. We then came up with the idea of using a master container, which is the “State” class. This class stores all of the test case objects, the changes that the user made while interacting with the system, and the individual functions that represent the user's possible actions while interacting with the system. We had the state class store the changes within a data structure for efficiency's sake. This makes communicating with the server much more seamless as we only need to send the changes back and forth with the server whenever a change is made rather than the whole state.

//Every test case object is stored within a class object called “test\_case” which contain the four attributes listed.

Class test\_case

```
{
    Test ID;
    newTest;
    Expected_result;
    Actual_output;
    Bool status;
}
```

Class State

```
{
    //container of all the test_case class objects, drawings, and changes
    List_of_tests = [ ];
}
```

//changes contains the recently made changes made such as drawings, added test cases, etc.

changes = [ ];

choose\_tool()

```
{
    return selected tool;
}
```

//Runs when user initially uses their tool to select a part of the code or draw to come up with test cases. Dependent on using selection-based tools.

select\_code(tool)

```
{
    Parameters = dict();
    generate_template();
    while(user doesn't click somewhere else out of the function)
    {
```

For argument in function\_arguments:

```
{
    //this stores the type of suggestion as a key in the dictionary with
    their associated suggestions as the value
    //suggests more than one suggestion
    //prompt the user select from the suggestions or input their own
```

show\_Suggestions(suggest(argument.type()));

if(user\_action is select\_fromSuggestion()

```
{
    select_fromSuggestion();
```

```
}
```

Else if (user\_action is input\_data())

```
{
    input_data();
```

```
}
```

```
}
```

//generate\_tests will generate tests and add them to the master container of all the test cases and also add them to the lists of changes.

generate\_tests(generate\_combination(Parameters));

changes.append(reference of what was drawn);

```

    }
}

select_fromSuggestion(reference of parameters)
{
    Parameters[type of selected suggestion]] = selected suggestion;
}

input_data(reference of parameters)
{
    Parameters[type of input] = input;
}

generate_combination(dictionary of possible_parameters))
{
    //generates a set of unique tuples containing all possible arguments of the function
    based on the user's selection
    Function_parameters = set([ tuple ]);

    //Using the parameters, generate possible combinations and store them in
    Function_parameters as tuples

    //possible code to generate
    For value in possible_parameters.value()
    {
        Tupler = tuple()
        For value in possible_parameters.value()
        {
            tuple.add(value[0])
        }
        Function_parameters.add(tupler)
    }
}

suggest(type)
{
    Return list of suggestion where the type of suggestion == type
}

```

```

generate_tests(combinations)
{
    //returns a list of test cases
    tests = [ ];
    For possible_param in combinations
    {
        Test_Case = new Test_Case Object;
        //user generates a new test case with all the possible parameters.
        Test_Case.newTest= newTest + possible_param;
        Test_Case.ID = ID that's related to the Test Case name
        // initially, the expected_output attribute of the Test_Case object is blank,
        // but the user can edit this field to add one
        Test_Case.expected_output = " "

        //initially, the actual_output is blank until the user executes the test case
        Test_case.actual_output = " "
        tests.append(Test_Case);
        changes.append(Test_case);
    }
    Self.list_of_tests = tests;
}

select_testCase()
{
    //returns the test case that user selects
    return test case;
}

input_expectedResult()
{
    test_case.expected_result = user_input()
}

input_testScenario()
{
    test_case.testScenario = user_input();
}

```



```

run_allTests(list_of_tests)
{
    For test in list_of_tests
    {
        Try:
            If(expected_output isn't stated)
            {
                Run test;
                //sets the actual output of the test_case object equal to the
                output
                Test.actual_output = output;

                //by default, the test case passes if the user has not indicated
                an expected output. In this case, the test case only checks to
                see if the test throws an exception.
                Test.status = true;
            }
            Else if( expected_output is stated)
            {
                Run test;
                //sets the actual output of the test_case object equal to the
                output
                Test.actual_output = output;
                If(actual_output == expected_output)
                {
                    Test.pass = true;
                }
                Else if(actual_output != expected_output)
                {
                    Test.pass = false;
                }
            }
        Except:
            Test.actual_output = error_message;
            Test.status = false;
    }
}

run_selectedTest(selected_testCase)
{

```



Try:

```

If(expected_output isn't stated)
{
    Run test;
    //sets the actual output of the test_case object equal to the output
    Test.actual_output = output;

    //by default, the test case passes if the user has not indicated an
    expected output. In this case, the test case only checks to see if the
    test throws an exception.
    Test.test_case.status = true;
}
Else if( expected_output is stated)
{
    Run test;
    //sets the actual output of the test_case object equal to the output
    Test.actual_output = output;
    If(actual_output == expected_output)
    {
        Test.pass = true;
    }
    Else if(actual_output != expected_output)
    {
        Test.status = false
    }
}

```

Except:

```

Test.actual_output = error_message;
Test.status = false;

```

```

}

```

```

delete_testCase(selected_case)
{
    Delete selected_case;
}

```

```

}

```

//runs the overall system

```

main()

```

```

{

```

```

while(system is open)
{
    If (user doesn't load an old state or user chooses a new state)
    {
        currentState = newState();
    }
    Else if (user loads from an existing state)
    {
        currentState = load(existing_state);
    }
    if(user selects code, generates test cases, or draws)
    {
        currentState.select_code(user's tool);

        //creates a copy of the state and sends it to the database
        send_to_database(currentState.changes);
        current_state.apply(currentState.changes);
    }
    if(changes are made by collaborators)
    {
        Collab_changes = fetch_from_database(changes);
        current_state.apply(collab_changes);
    }
}
}

```

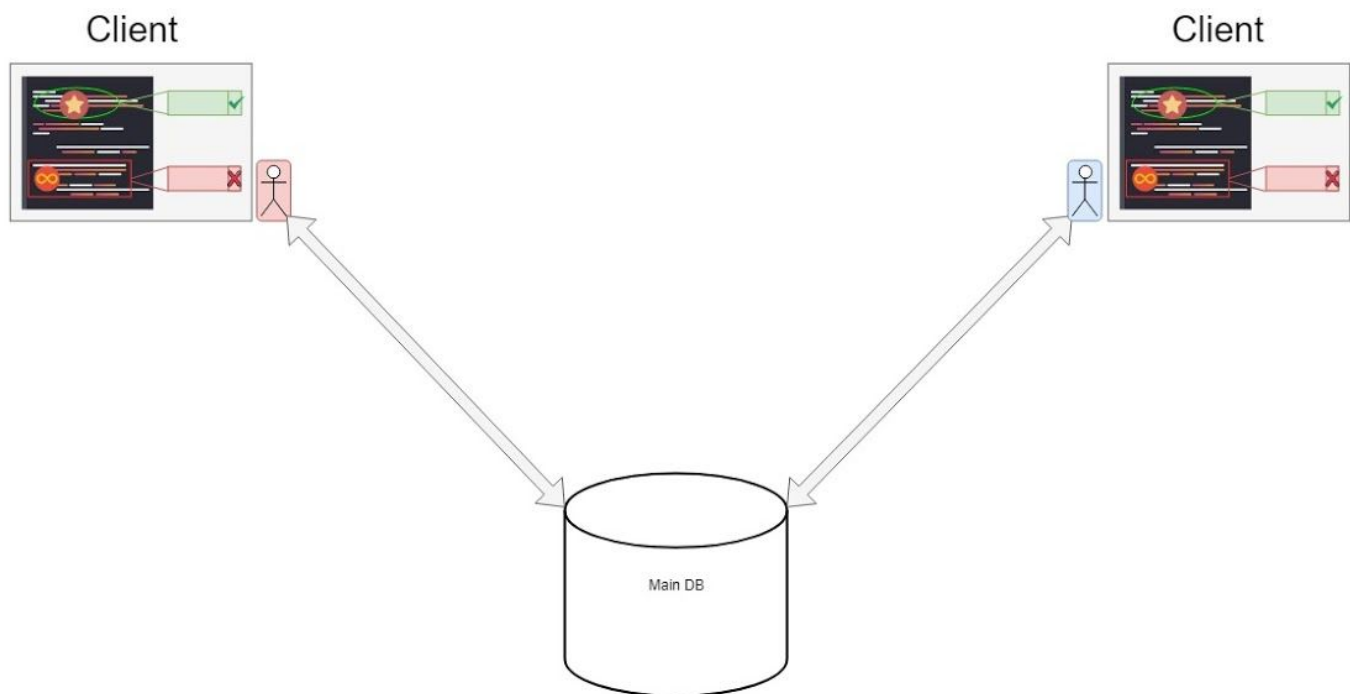
## 5 - Architecture Design - Robert Maldonado & Alfredo Gaytan

### Analysis: World Modelling

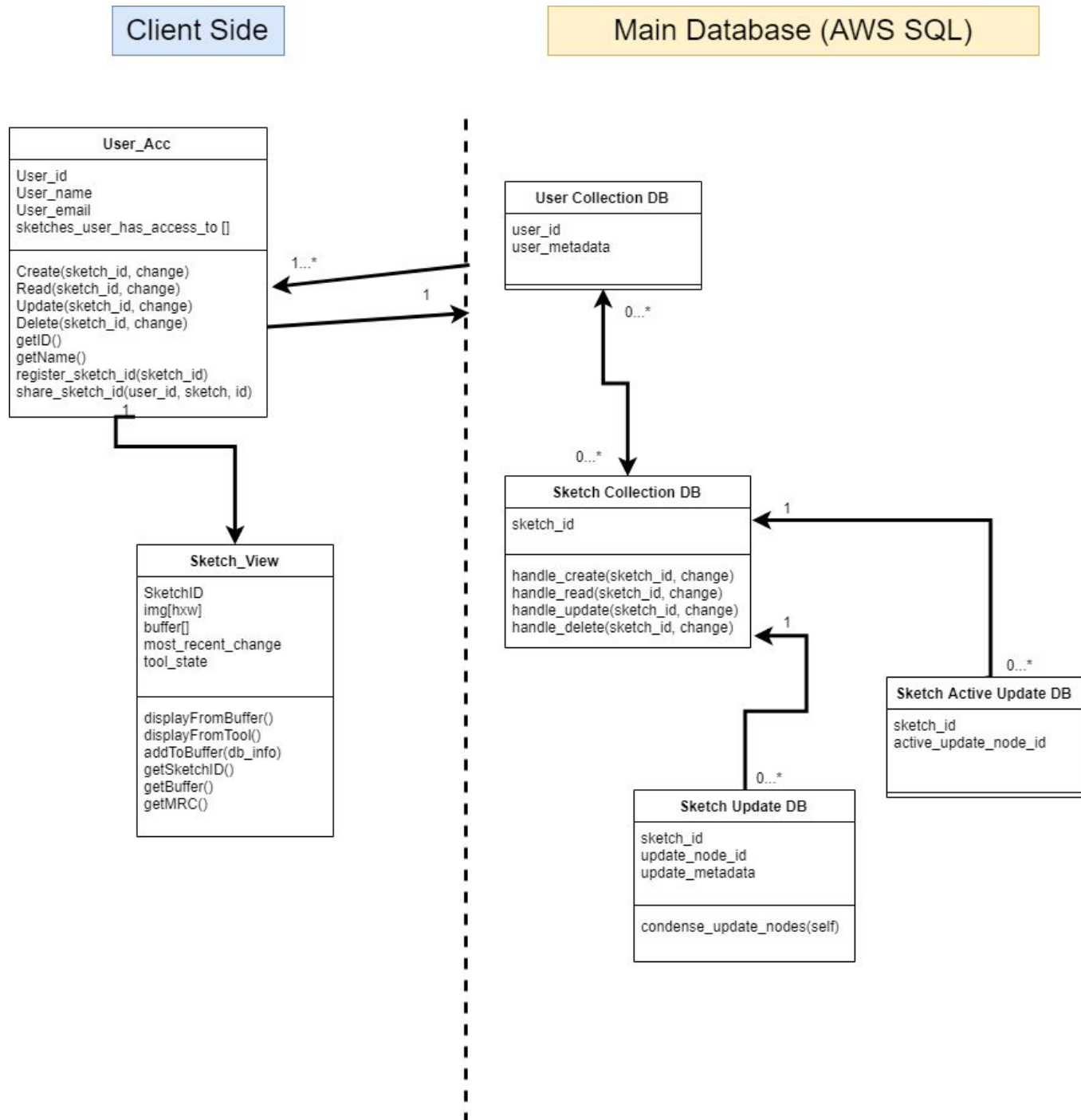
While analyzing the problem at hand, we used the method of World Modeling by thinking about the problem at a global scale. Doing this made us consider all possible problems that could come up in many different global scenarios such as scalability, performance, and reliability.

### Synthesis: Decomposition

While solving all of the architectural problems we were presented with, we used the method of Decomposition by breaking our system's features down into their essence and ensuring that they can be supported in an architecture aspect.



Note: This figure outlines the most abstracted view of the architecture, with multiple clients communicating to a centralized database.



Note: This figure outlines the shema of the centralized database.

## 5.1 - Database Type

This system will be using Amazon RDS for SQL Server, which will make it easy to deploy, operate, and scale a SQL Server on the cloud. We chose this type of database because it can be used as a centralized database and the relational model of SQL Server supports the types of functionalities used through the rest of the our database design. Additionally, this system will be using delta encoding by storing only the differences between sketches in the database. The database does not maintain a completed state of a sketch, but stores the changes that a user has made to a sketch. The client maintains an updated state by requesting these changes.

## 5.2 - Database Schema

We represent our architecture using a UML-style database schema.

- Client-side objects
  - User\_acc: represents the account that is being used by the user to manipulate the Sketch\_view object.
    - Key attributes:
      - User\_id; to uniquely distinguish other users in the database
      - sketches\_user\_has\_access\_to (User\_id list); a user can share their sketch with other users by getting their respective User\_id attributes and adding it to the sketches\_user\_has\_access\_to list.
    - Key functions
      - Create; allows users to generate new information from initiating a new sketch which then creates a new or a collection of test case objects
      - Update; the user is modifying an existing sketch invoked from the create function previously.
      - Read; allows the user to browse between sections of code to view generated test cases.
      - Delete; allows the user to delete an instance of a test\_case object that was invoked previously by the user with the create function
      - Share\_sketch\_id; allows the user to share their sketch with another user
      - Register\_sketch\_id; adds a sketch\_id to the attribute sketches\_user\_has\_access\_to; this is usually invoked when another user shares their sketch id (as per the previous bullet point)
  - Sketch\_view: represents the sketch object that the user is working on, whether it is their sketch they created or from another user (that was shared from).

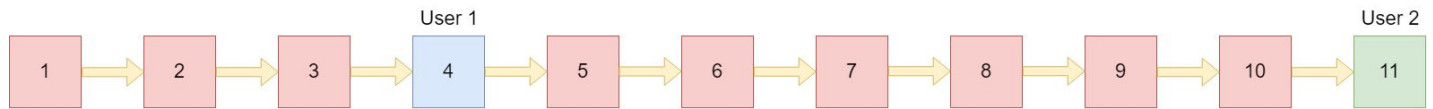
- Key attributes:
    - SketchID; to uniquely distinguish between other sketches in the database
    - img; represents the pixels on the workspace
    - Buffer; used to allocate changes (from another user which is then fetched from the server) that are yet to be loaded into the user's workspace
    - Most\_recent\_change; indicates the most recent change made by the user
    - Tool\_state, indicates which tool the user is using at the moment
  - Key functions
    - displayFromBuffer(); loads in the changes a different user has made from the user's local buffer for the sketch view
    - displayFromTool(); loads in the changes a user has made from the tool they are using
    - addtoBuffer; any change made by a user is then added to the buffer and then later displayed using displayFromBuffer()
- Server-side objects
  - User Collection DB: represents the aggregate of all users in the database
    - Key attributes
      - User\_id; to uniquely distinguish each user
      - User\_metadata; general information on the user, whether that be their name, email, DOB, etc
  - Sketch Collection DB: represents an aggregate of all the sketches in the database
    - Key attribute - sketch\_id; to uniquely distinguish each sketch in the server
    - Key functions - handling all CRUD functions; handles the modifications made by the user in the client side and modifies it accordingly in the database
  - Sketch Update DB
    - Key attributes
      - Sketch\_id
      - Update\_node\_id
      - Update\_metadata
    - Key function
      - condense\_update\_nodes
  - Sketch Active Update DB
    - Key attributes
      - Sketch\_id; ~
      - active\_update\_node\_id

### 5.3 - Database Version Control & Management System

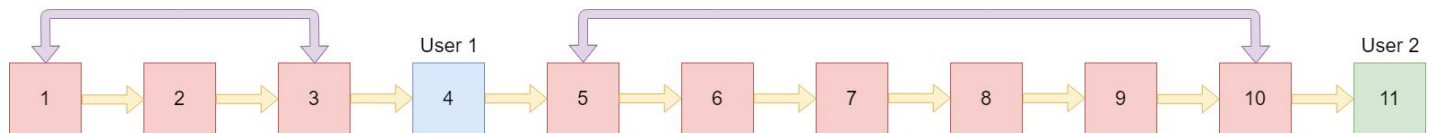
In this system, version control is used to ensure that every client is viewing the most updated version of a sketch. The version control system is very similar to git because each individual sketch object has a history of changes that have been made to it. As a client begins to make changes (like adding new things to the sketch or updating existing things) this list of changes begins to lengthen. For all clients that aren't the source of a change, they are kept up to date on the most updated version of the sketch by requesting these new changes. If the database receives 2 changes at very similar times, the database will chose the first change that came into the database.

The database management system (DBMS) of our database ensures that this collection of changes does not overwhelm the database. As the database begins to collect changes, it also keeps track of what users are actively using a change (a change is considered active if a user is still awaiting to receive that change, one active changes can only apply to one user). If there are more than 2 sequential changes that are not active, then the DBMS will condense these changes into one (as seen in the diagram below). This is useful when a user's stops getting updates for sometime and then needs to collect the updates that they missed. For example, if you have someone with a fast internet connection and a slow internet connection, if the individual with a fast internet connection keeps making changes, the individual with the slow internet connection will not get updates as fast so instead of having to make many requests for updates, the updates are condensed into one update.

Scenario: Two users are looking at the same sketch. User 1 disconnects from a sketch to attend another meeting. User 2 continues making changes while they are gone.



Step 1: The DBMS detects that there are ranges of inactive changes.



Step 2: The DBMS condenses these changes into one change.  
This will make it easier for User 1 to get to the same state as User 2 once User 1 gets back online.



Note: This diagram shows how the DBMS condenses changes.



## 6 - Design Processes

### Application Design - Everyone

For application design, everyone had their own unique thoughts to share and contributed equally in this aspect. We first started by designing what the application would look like and then focused on the features. Our team came up with many ideas and discussed on what should be a feature and what shouldn't be included. We drew out diagrams, models, mock-ups, charts, that would best represent the way our system would work and how the user would interact with our system. There were many debates and compromises but we finally came up with a thorough application design that our team was satisfied with. After completing our application design and making sure that every team member had a thorough understanding of the overall system, we split up into pairs to work on the rest of the designs.

### Interaction Design - Ankit Jain and Archita Ganesh

For interaction design, Ankit Jain and Archita Ganesh wanted to ensure that the process for users were intuitive, collaborative, and easy-to-use. We first analyzed what type of audience our system would target and how we would design our system according to their needs. This process of analysis was actualized with the two personas that broke down the audience of industry programmers and testers. After understanding the backgrounds of our target audience, we wanted to synthesize the information and understand how our design would look like if implemented through the use of mockups. By drawing our first set of mockups, we were able to collaborate as a team as to how the system will be formatted and what functionality would be include for the user to interact with. This helped us understand how we could make our collaborative system simplistic but effective when generating test cases. We later translated our whiteboard sketches to actual mockups using Microsoft PowerPoint. Each design change and artifact can be noticed on our design artifacts google drive which is linked below. After building the mockup design, we wrote up a Cognitive Walkthrough in order to evaluate how a typical user would possibly interact with the user and get a better view as to how the design actually works and make changes to the mockup as well as the other aspects of design.

### Implementation Design - David Bui and Caleb Chu

For implementation design, David Bui and Caleb Chu first came up with a walkthrough with all the essential features that the user would be interacting with. We drew out diagrams depicting each action within the walkthrough and then proceeded to write pseudo code thoroughly implementing each part. We then thought about how we were going to organize the data and store each test case and action. After discussing, David and Caleb chose two implement two classes: a test\_case class that contains the individual attributes associated with a test case,

such as the Test ID, test name, expected, actual results, and the status. We then decided on a master container, which we called the “State” class, which contained the all of the test cases, changes, and the individual functions that represent the possible actions that user may act upon when interacting with the system. After that, we implemented algorithms that would automatically come up with extra test cases after the user selects the required parameters. Basically, auto-generator works by using an algorithm to come up with all the possible combinations of the parameters. This made coming up with test cases much more easy. We also chose to use another data structure to store the changes in which we send to the database. We chose to send only the changes rather than the entire state of our current system for the sake of efficiency when contacting with the main database.

### Architecture Design - Robert Maldonado & Alfredo Gaytan

For architecture design, Alfredo Gaytan and Robert Xavier Maldonado had the goal of creating an architecture for this application that is scalable and is able to account for the most recent changes made by users. First, we were trying to figure out how to keep track of the changes made by each user. Robert was inspired from a research project where he worked on a non-relational database on python that kept track of changes through differences instead of entire states; this is called delta encoding. After that, we wanted to visually represent our architecture. We thought the easiest way to tackle this part of the design was to brainstorm and create a UML-style database schema that organized the activity between the client and the database server.

## 7 - Reflection

For all four types of designs, the entire team worked together during the meetings to make sure we understand each other's thoughts and approaches. This collaborative process allowed us to formulate alternatives, which led us to finalizing our final ideas. These final ideas were then broken up to the 4 types of Design in which split up the writing component. Throughout this process, we learned how to integrate the different software design methods and learned about the cyclic and iterative process of design. When one component of design changes, another one can alter as well. Every change was made to ensure the goals are achieved in the most effective way possible. This design studio helped us actualize that process and instill an open-minded thinking process that was open to change and constant communication.

## 8- Design Artifacts

Please follow this link to view all of design artifacts used: [click here](#)