Course Name: Software Engineering

Course Number and Section: 14:332:152

Group #10

VirtualLogicLabs
Report #2

Github: https://github.com/SagarPhanda/VirtualLogicLabs

Date Submitted: March 11th, 2018

Group Members (6):
Sagar Phanda, Khalid Akash, Dhruvik Patel, Vikas Khan, Joe Cella, Yiwen Tao

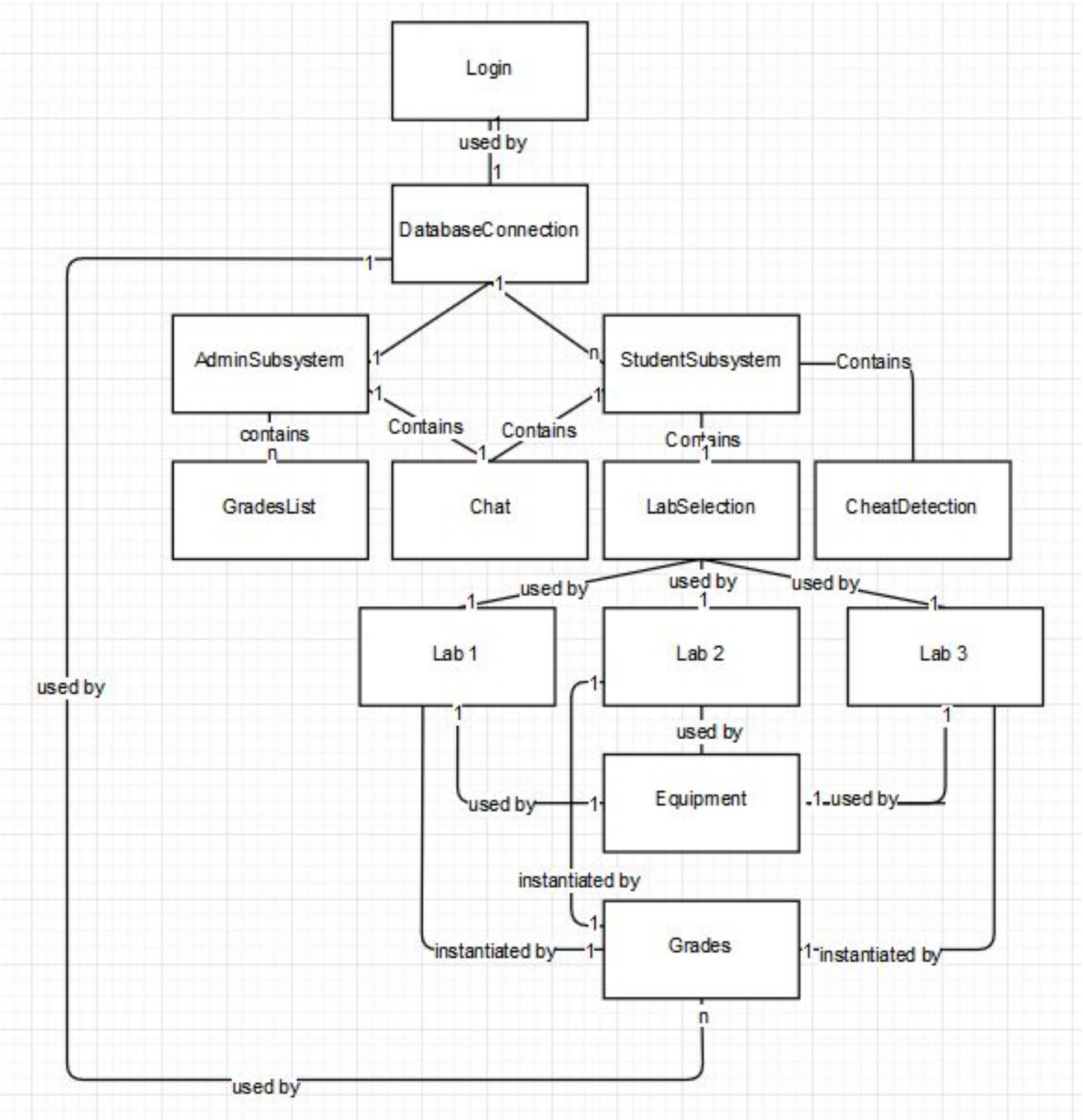
Individual Contributions Breakdown

All team members contributed equally.
Every team member contributed 16.66% of the total report.

Table of Contents:

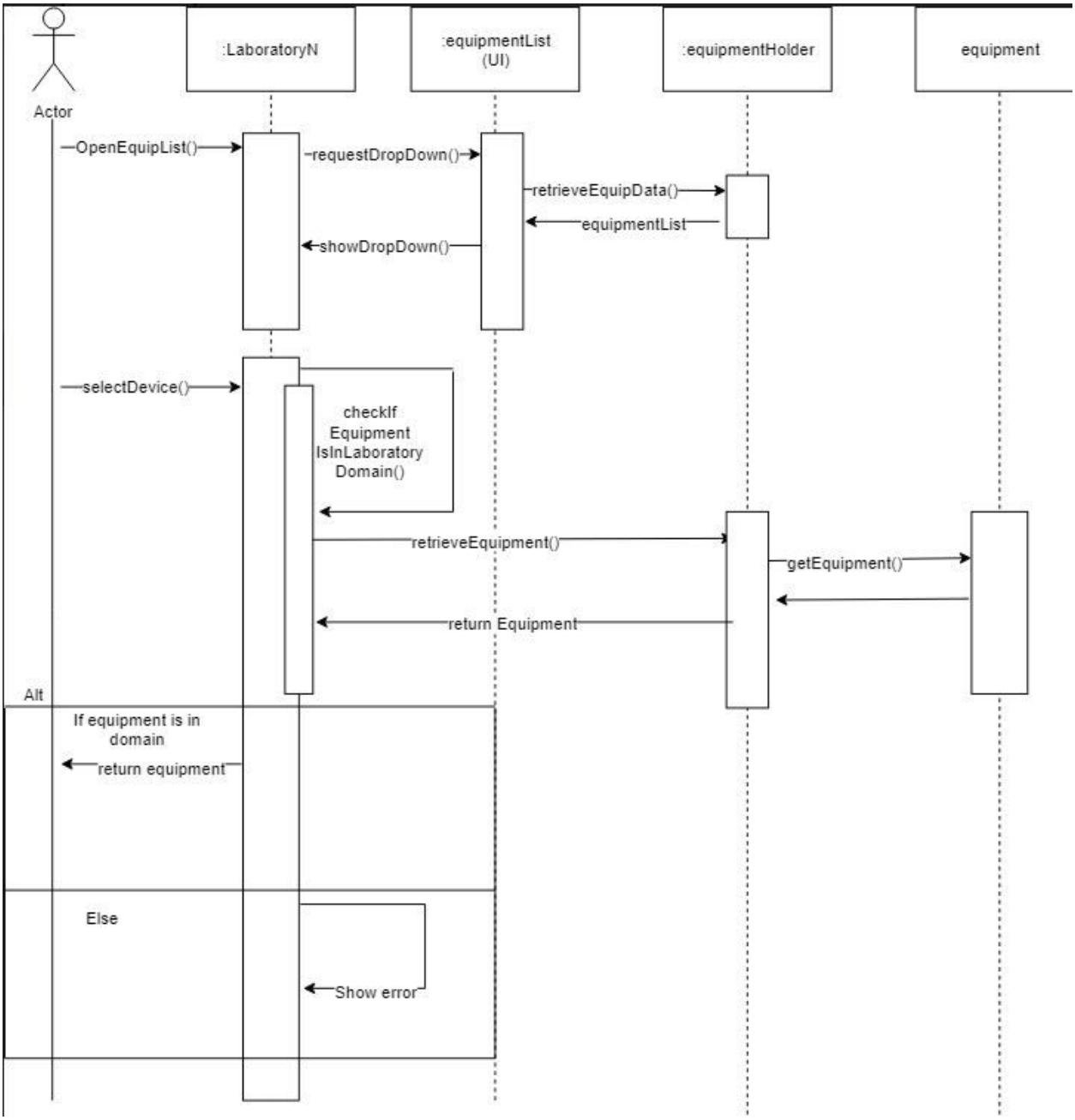
0. Revision from Report 1

Domain Diagram:



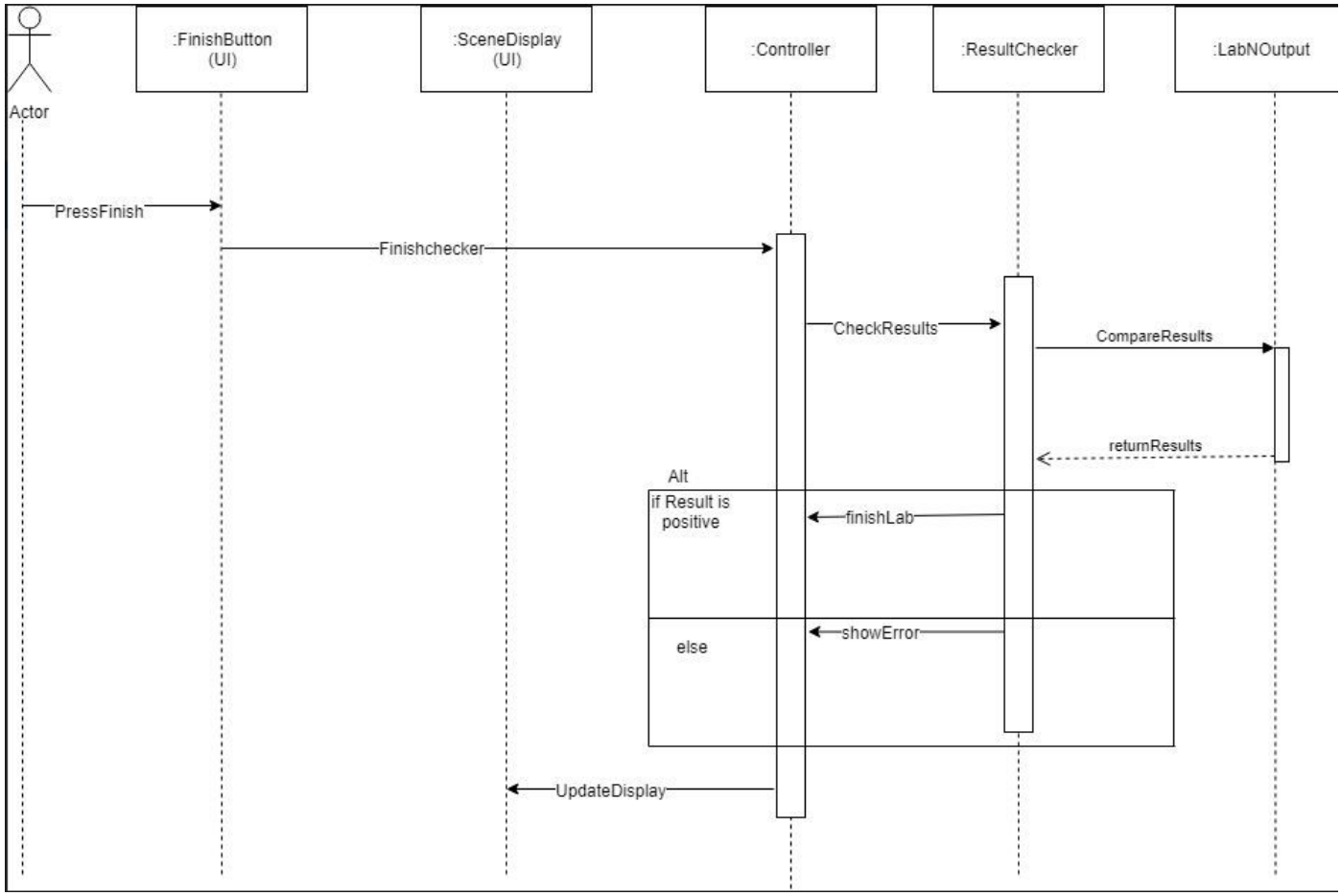
1. Interaction Diagrams

1.1: UC-1 (manageEquipment)



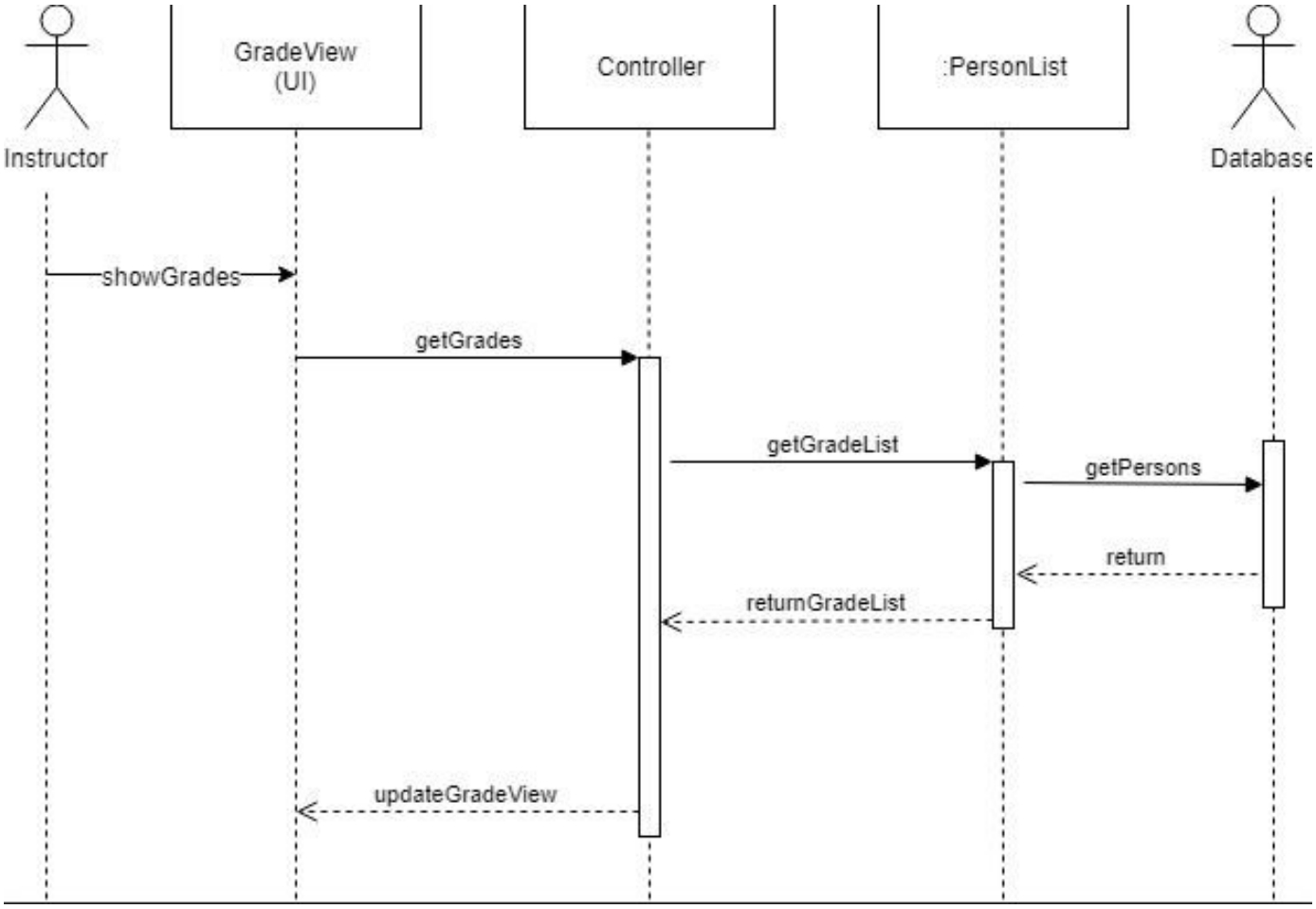
The core of this particular sequence diagram is the equipmentList, equipmentHolder, and equipment objects. EquipmentList is the graphical user interface component that shows the list of equipments available to the user. This is coupled with the equipmentHolder class which holds the list of equipment objects. This allows us to easily change the type and number of equipments without worrying about breaking the functionality of the equipmentList component. LaboratoryN is the ‘controller’ which takes the responsibility of communicating with the equipment related classes.

1.2: UC-2 (finishAndCheck)



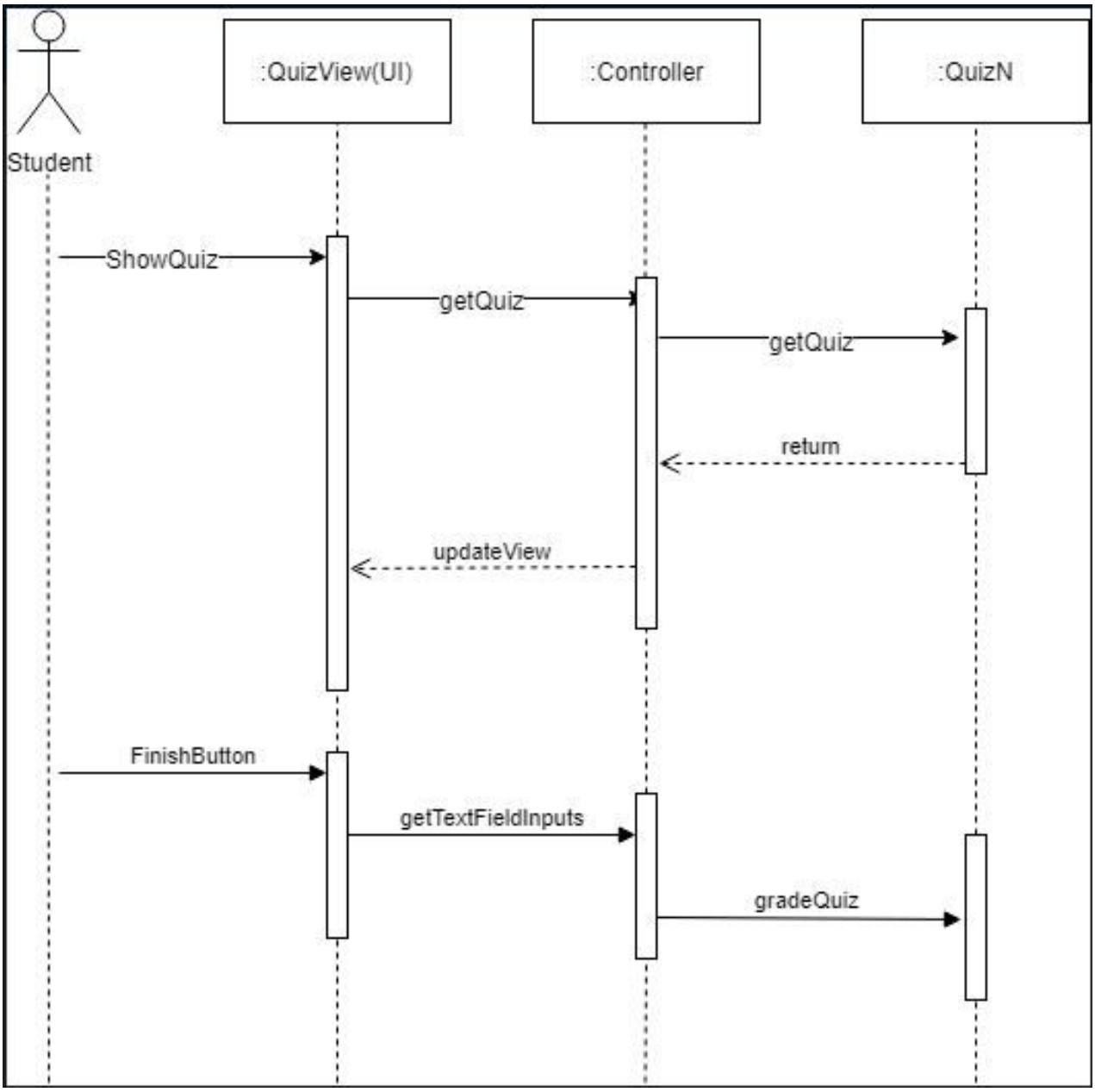
In this scenario, the user (student) needs a method of notifying the system that the student is done with a particular section of the lab. This is done through a button tied to the user interface (FinishButton). After this button is clicked, the system transfers control to the controller. The controller signals the ResultChecker entity to check whether the user's answers to the particular section of the lab are correct. The ResultChecker then signals the lab output (Lab3Output, Lab2Output, or Lab1Output) to compare the student's inputs with the correct inputs, and then return the results. If the student's answers are correct, the ResultChecker notifies the controller that the sequence to finish the lab can be initiated. Otherwise, an error is returned to the controller. Depending on whether the student was successfully able to answer the lab section correctly or incorrectly, and the number of attempts remaining, the controller will then instruct the SceneDisplay user interface to update the GUI accordingly.

1.3: UC-5 (grades)



This scenario begins with the instructor attempting to look at the grades for a particular student or group of students. The instructor interacts directly with the GradeView user interface, and when the instructor requests grades, the controller is signalled from the interface. The controller then contacts PersonList, which goes through the database containing the names and grades for each student. The database then returns these values to PersonList, which then passes it back to the controller, and in turn to GradeView, which displays the information for the user to see.

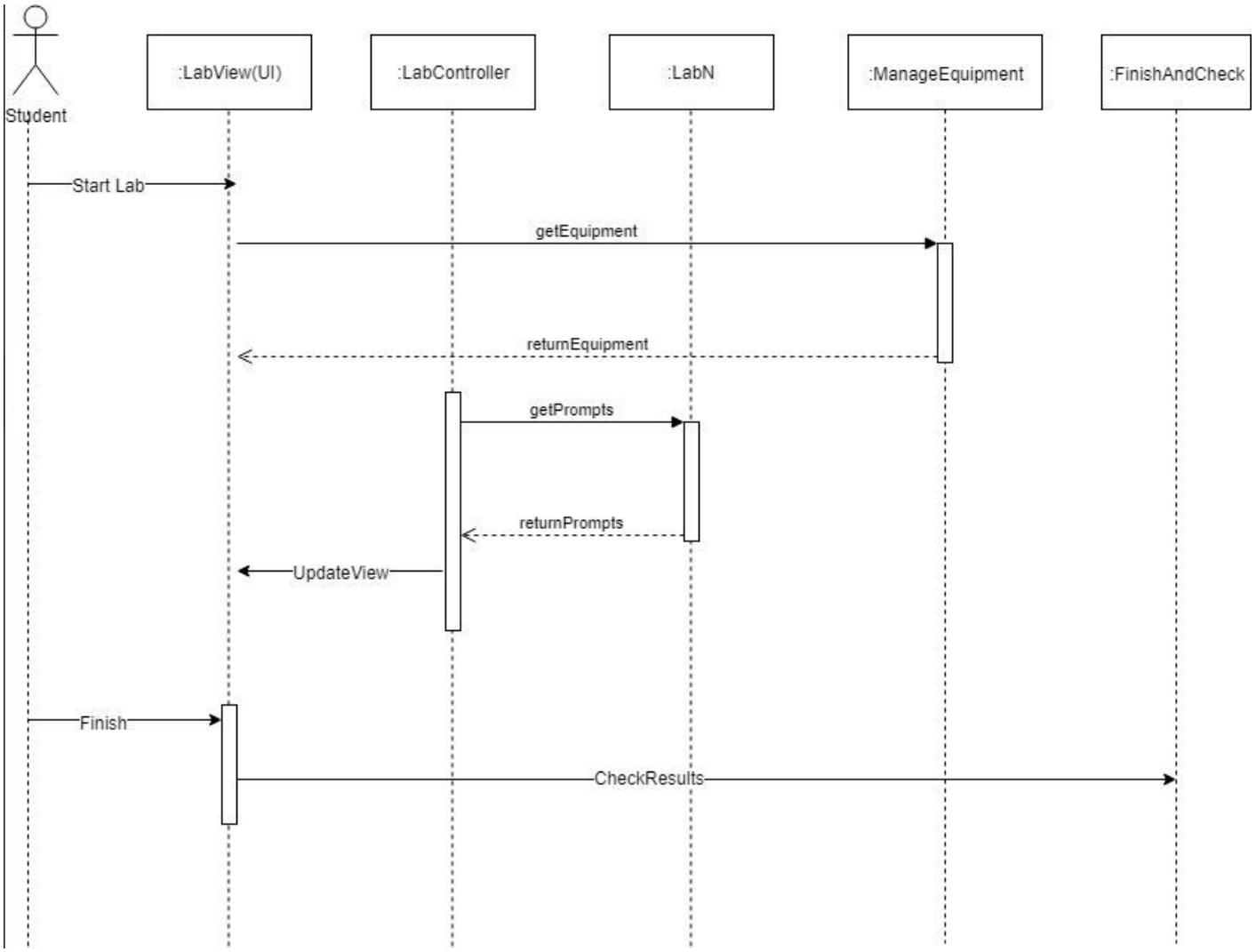
1.4: UC-14 (giveQuiz)



For this scenario, the Single Responsibility Principle was what we attempted to implement. The QuizView is responsible for showing the graphical representation of the quiz, the controller receive information from the specific quiz to control the QuizView's graphical capabilities. The QuizN class has the responsibility of holding the questions and checking the answers entered into the QuizView. The controller delivers the answers the the QuizN class.

1.5: UC-15, UC-16, UC-17 (labOne, labTwo, labThree)

The system sequence diagram below demonstrates the general sequence for all three labs as they follow a very similar pattern:



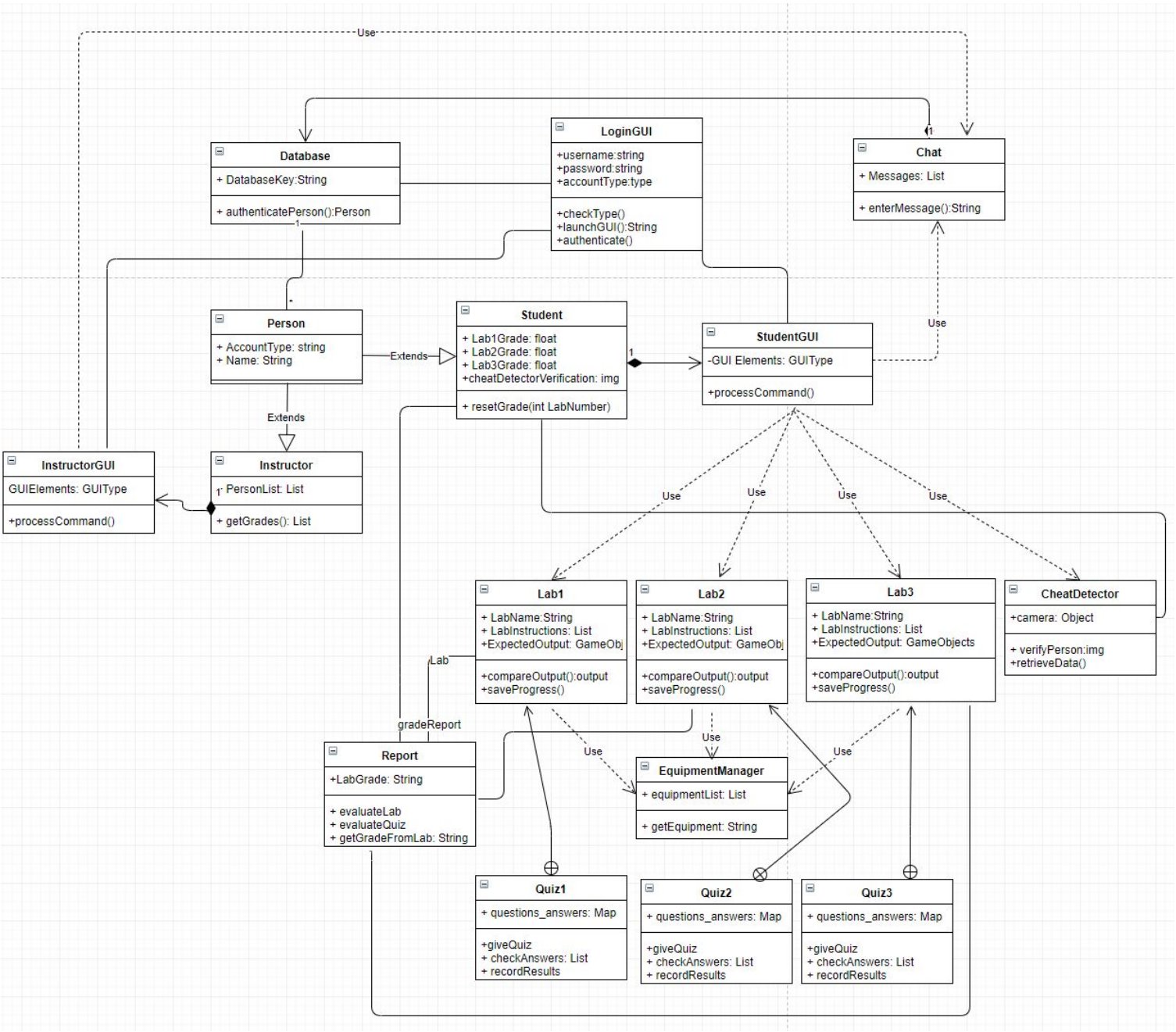
This sequence diagram tries to implement is little coupling as possible between the involved classes.

Note: The ManageEquipment class is a representation of the UC-1 Sequence Diagram shown above. We attempted to separate as many of the functions as possible to ensure there is opportunity to dynamically change either the actions taken in when selecting equipments, checking the results of the laboratory, and the prompts presented by the particular lab. We tried to ensure that the Single Responsibility Principle is kept, where the LabView is only responsible for rendering the graphics to the screen, the LabController is responsible for controlling the view, and LabN is in charge of the lab specific instructions needed for the user.

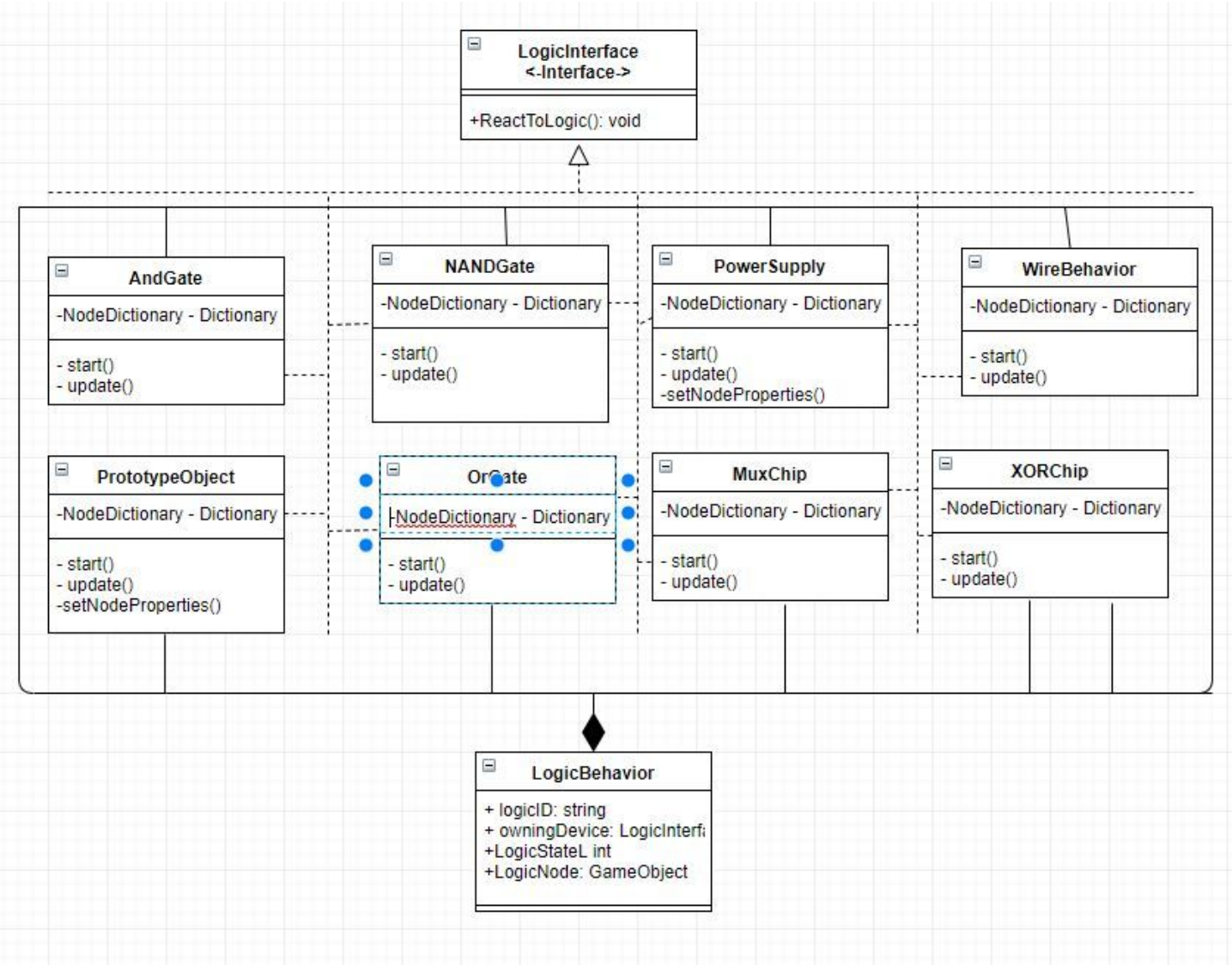
2. Class Diagram and Interface Specification

2.1: Class Diagram

2.1.1: General Class Diagram



2.1.2: Equipment Manager Inner Structure



2.2: Data Types and Operation Signatures

Class Name:	LoginGUI - Provides interface for login	
Attributes:	-username:string -password:string -accountType:type	Stores username Stores password Stores account type (student/instructor)
Operations:	+checkType() +launchGUI():String +authenticate()	Checks the type (student/instructor) Initiates login GUI for users Tests if username and password are correct

Class Name:	Database - stores users, passwords, grades	
Attributes:	-DatabaseKey:String	Stores key for database
Operations:	+authenticatePerson():Person	Tests if user is in database

Class Name:	Chat - Allows students to chat with each other and instructors	
Attributes:	-Messages:List	Stores the chat messages
Operations:	+enterMessage():String	Allows student/instructor to input a new chat message

Class Name:	Person - Responsible for keeping track of individuals	
Attributes:	AccountType:String Name:String	Holds type of account (student/instructor) Stores name of user
Operations:	-	-

Class Name:	Student - Keeps track of a particular student's grades and other info	
Attributes:	-Lab1Grade:float -Lab2Grade:float -Lab3Grade:float -cheatDetectorVerification:img	Holds lab 1 grade Holds lab 2 grade Holds lab 3 grade Stores image of user's face to identify cheating
Operations:	+resetGrade(int labNumber)	Resets a grade for a specific lab (designated by input labNumber)

Class Name:	StudentGUI - interface for students	
Attributes:	-GUI Elements: GUIType	Contains the buttons, textfields, and other GUI attributes that are used in the StudentGUI.
Operations:	+processCommand()	Processes user commands

Class Name:	Instructor - keeps track of instructor information	
Attributes:	-personListGUI Elements: -GUIType:List	Contains the buttons, textfields, and other GUI attributes that are used in the personListGUI and other Instructor GUI elements.
Operations:	+getGrades(): List	Gets student grades for instructor

Class Name:	InstructorGUI - interface for instructors	
Attributes:	-GUI Elements: GUIType	Contains type of GUI (instructor)
Operations:	+processCommand()	Processes user commands

Class Name:	Report - delivers grade reports to students and instructors	
Attributes:	-LabGrade: string	Contains grades for lab
Operations:	+evaluateLab() +evaluateQuiz() +getGradeFromLab(): string	Compares lab performance to correct solutions Compare quiz solutions to correct solutions Obtains lab grades

Class Name:	Lab1, Lab2, Lab3 - control the operations of the individual labs	
Attributes:	-LabName: string -LabInstructions: List -ExpectedOutput: GameObj	Contains name of lab Contains lab instructions Contains correct results
Operations:	+compareOutput():output +saveProgress()	Compares student answers to correct output Saves student lab progress

Class Name:	Quiz1, Quiz2, Quiz3 - controls operations of the quizzes	
Attributes:	-questions_answers:Map	Holds answers to questions
Operations:	+giveQuiz() +checkAnswers(): List +recordResults()	Gives student quiz Checks for correct answers Records student grade on quiz

Class Name:	EquipmentManager - handles all equipment for the labs	
Attributes:	-equipmentList: List	Contains pieces of lab equipment
Operations:	+getEquipment(): string	Gets selected equipment piece

Class Name:	CheatDetector - identifies if students are cheating	
Attributes:	-camera: object	Computer camera is used to monitor students
Operations:	+verifyPerson(): img +retrieveData()	Verifies that the student matches his/her stored image Retrieves stored images of students to compare to camera image

Class Name:	LogicBehavior - is a component (A subclass of a GameObject) of a GameObject called LogicNode which sets the voltage states of a particular node in the Electrical Device.	
Attributes:	+LogicID: String +LogicState: int -LogicNode: GameObject -OwningDevice: GameObject	The unique ID for each LogicNode gameobject. The state that emulates the circuit voltage of the particular node (Low, High, or Invalid). OwningDevice is the device that owns the particular logicNode.
Operations:	-Start() -Update()	Start initializes the conditions for the component script. Update allows the logicBehavior script to react to any changing input from the user every frame.

Class Name:	ProtoboardObject - Object that holds the logic nodes for a protoboard gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeDictionary	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	PowerSupply - Object that holds the logic nodes for a PowerSupply gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	AndGate - Object that holds the logic nodes for a AndGate gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	OrGate - Object that holds the logic nodes for a OrGate gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	XORGate - Object that holds the logic nodes for a XORGate gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	MUXChip - Object that holds the logic nodes for a MUXChip gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	NandGate - Object that holds the logic nodes for a NandGate gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

Class Name:	WireBehavior - Object that holds the logic nodes for a WireBehavior gameobject and changes the state of its nodes accordingly.	
Attributes:	-NodeHashMap	Dictionary (Hash Table) of LogicID to Node GameObject mappings.
Operations:	-start() -update() -setNodeProperties()	Start initializes the conditions for the component script. Update allows the device script to react to any changing input from the user every frame. SetNodeProperties is a helper function to help the protoboard initialize the logic nodes inside it.

2.3: Traceability Matrix

Domain Concepts	**Classes**							
	LoginGUI	**Database**	**Person**	**Student**	**StudentGUI**	**Chat**	**InstructorGUI**	**Report**
Login	**X**							
DatabaseConnection		**X**						
Admin Subsystem							**X**	
Student Subsystem					**X**			
GradesList							**X**	**X**
Chat						**X**		
Grades								**X**

Domain Concepts	**Classes**							
	Lab 1	Lab 2	Lab 3	Equipment Manager	Quiz1	Quiz2	Quiz3	CheatDetector
Equipment	**X**	**X**	**X**	**X**				
LabSelection	**X**	**X**	**X**					
Lab1	**X**				**X**			
Lab2		**X**				**X**		
Lab3			**X**				**X**	
CheatDetector								**X**

Domain Concepts	**Classes**								
	Logic Interface	Logic Behavior	Protoboard Object	Wire Behavior	Power Supply	AND gate	OR gate	NAND gate	XOR gate

Equipment	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**

Most of our domain concepts mapped very similarly over to our class diagrams with a few exceptions. The Login, StudentSubsystem, and AdminSubsystem domain concepts were changed by adding a postfix ‘GUI’ at the end of them as much of their functionality are GUI operations. We changed the name DatabaseConnection to Database as we expect that class to handle everything from the initial connection to database management methods. We changed the name of the concept Grades to Report as we felt we wanted more information within that class than just the grades, such as the specific areas where the students failed in the Virtual Lab to diagnose design issues. We changed the Lab domain concepts to contain both a Lab class and a Quiz class as they are designed to be decoupled different systems. The equipment underwent the biggest change as during development of the equipments, we found uses for many classes that specifying different equipments.

3. System Architecture and System Design

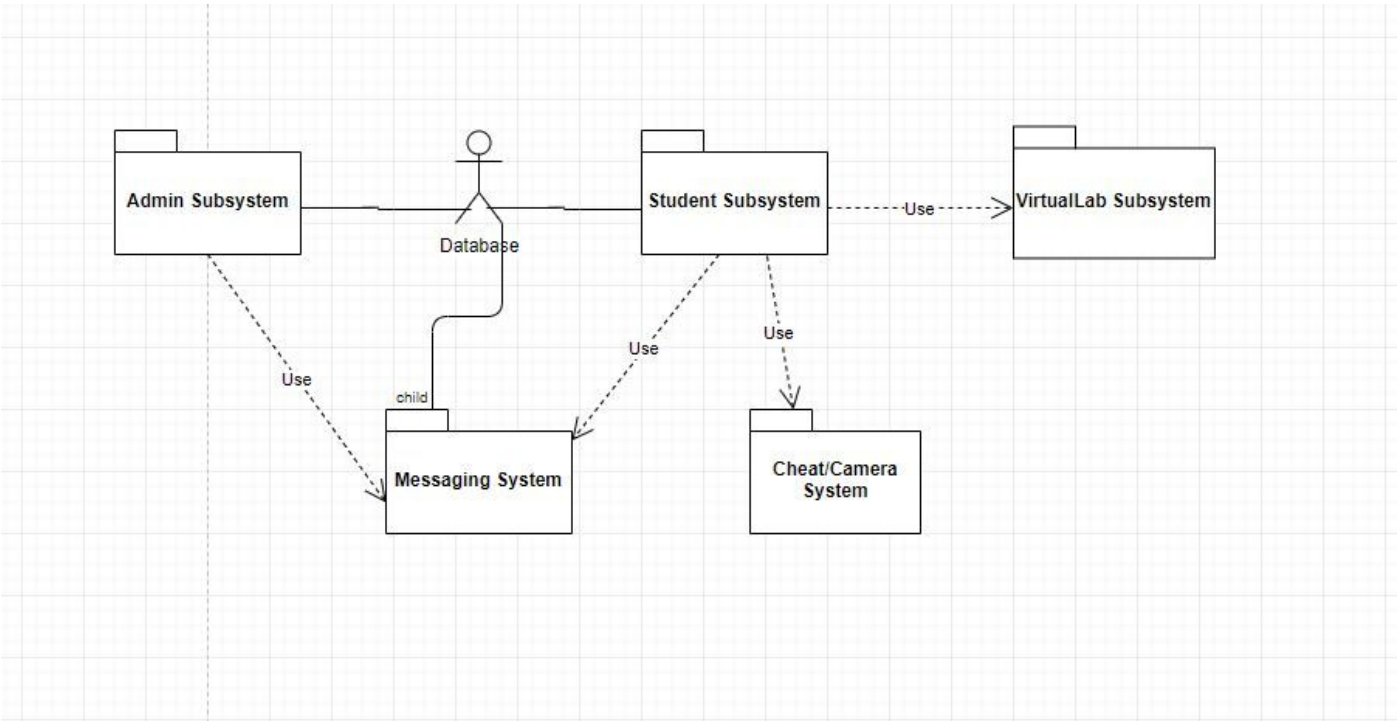
3.1: Architectural Styles

Event-driven architecture - This architectural style relies heavily on detection of, and reaction to events, or changes in state. As our system will be an interactive one, much of it will rely on the actions of the users, and must respond accordingly. Our system must be able to respond appropriately to each button click, as well as the click-and-drag features installed in each lab. The systems developments and procedures are almost exclusively reactions to the changes in state brought about by the user. The Unity Game Engine (Framework) is based on an Event driven architecture where each script is initialized in a Start method and the event loop executes the Update method. There are special cases where a callback is necessary, such has when the user clicks the mouse, a mouse operation callback is executed which can be handled in the “scripts” or “classes”.

Database-centric architecture - This type of architecture typically relies on a standard relational database management system rather than in-memory or file-based data structures. We will be relying on databases for a good portion of this project, as a means to store information about each student and instructor, lab grades, quiz grades, usernames, and passwords

3.2: Identifying Subsystems

3.2.1: UML Package Diagram



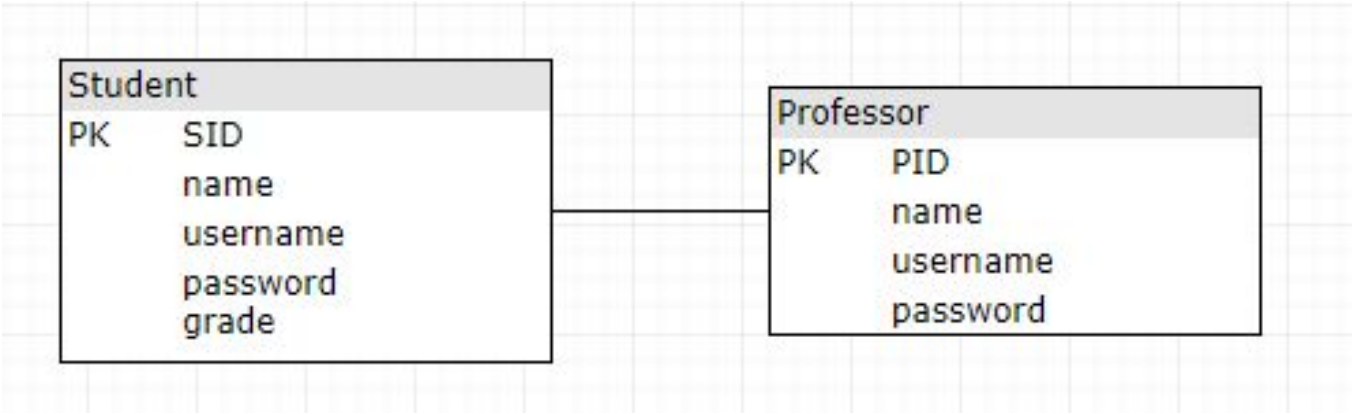
The reason the packages are assembled the given way is that they are being developed independently. We can develop each group of classes independently, and test them in such a way, such as developing

the Virtual Lab subsystem, and Student Subsystem independently without affecting the other. Each package however contains its own Model-View-Controller structure that allow each subsystem to work. For example:
The VirtualLab Subsystem has its own model, which represent the equipment available to the lab, how the equipments react to each other, and what the correct output is for a particular lab. The view is run by the Unity Engine Framework that shows the sprite components of a particular equipment. The controller controls how user input causes the equipments to react and passes the appropriate actions to the model.

3.3: Mapping Subsystems to Hardware

Our system does run on multiple hardwares. We have a client - server interaction between our application and a cloud based relational database. Our database keeps track of the student's data (Student Subsystem) for the administrator to manage (Admin Subsystem). Our server also facilitates the messaging system. When a user, be it the student or the administrator (Instructor/Professor), types in the messaging system, the message is sent to a database and relayed along all other open messaging systems that query it.

3.4: Persistent Data Storage



Our lab system as an event-driven system so users can do whatever they want in the real lab. Each event will change the associated value in the database. The system tracks how far each user goes in each lab so that their progress is saved on cloud. We use amazon web service to store the data, which is reliable and efficient. Here we have two roles: student and professor. They have their own id, name, username and password. Student has a extra grade item to keep their scores.

3.5: Network Protocol

We will be using the Firebase SDK to help us set up a connection to the firebase server. The SDK will facilitate setting up the connection in C#, which is the programming language of choice for the Unity Engine Framework.

3.6: Global Control Flow

3.6.1: Execution Orderness

Our system is event-driven, as once users enter each individual lab, they are responsible for the circuits that are created, and they can use any combination of materials that they choose. It is likely that many students will have circuits that consist of different pieces, or are presented in a different layout. Every action that the user makes while constructing the circuit affects the outcome and the functioning of the circuit, which can impact the grade that they receive. In the end, multiple users may end up with the same results, but they will not have necessarily taken the exact same steps, as their circuits can differ in chip placement, or the way that they went about constructing the circuit. Along with the virtual laboratory section, the user interface is also loop based - event driven. It waits for a particular action from the user, and updates the UI accordingly.

3.6.2: Time Dependency

There is a timer in a system, but it is only kept to decrease to user's grade if the user (Student) is taking far too long. However, the timer does not force the user to exit the program or any type of execution at any time. In fact, we allow the user to save their progress to come back to it later.

3.6.3: Concurrency

At this point, we do not have a concurrency model. However, it is likely that for some of the networking systems (messaging, and retrieving student data from the database), we will need several threads to ensure that the main User Interface thread is not interrupted.

3.7: Hardware Requirements

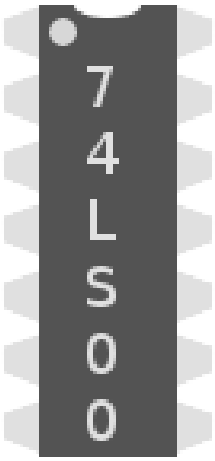
Listed below are specific requirements for our system:

- OS: Windows XP SP2+, Mac OS X 10.9+, Ubuntu 12.04+, SteamOS+
- Graphics card: DX9 (shader model 3.0) or DX11 with feature level 9.3 capabilities
- CPU: SSE2 instruction set support
- Screen: minimum resolution of 720p
- Minimum network bandwidth: 56k
- Minimum hard drive space: 100 megabytes

4. Algorithms and Data Structures

4.1: Algorithms

Much of our complex algorithms reside in getting the Virtual Laboratory section of the application working. What this means is the logic between the circuit equipments, such as Breadboard, NAND Chips, Wires, and other similar components are computed in conjunction with each other. Each digital logic equipment is filled with objects that we call “Logic Nodes” positioned in critical locations in the chip, such as the output pins in the following 74LS00 Double NAND chip:



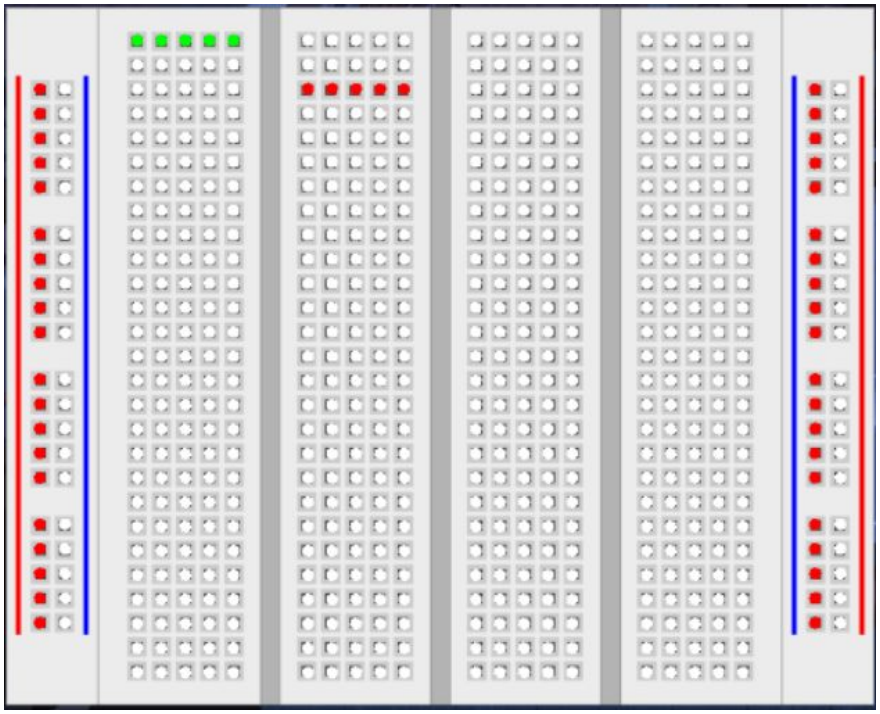
The interaction between the Logic Nodes of this NAND gate and other digital circuit equipment is facilitated by the Unity Engine’s collision detection system (when one Logic Node object is positionally overlapping another Logic Node object). When the device is ‘set’, meaning the user has stopped moving the device via the mouse, the collided nodes are detected and the device is forced to evaluate the ‘Logic States’ (Logic LOW, Logic HIGH, or INVALID) of any external Logic Nodes. Based on the particular device specification (a NAND device object would follow the logic of the NAND gate configurations), the device will set its own Logic Node in reaction to the inputs. Some devices may have outputs, which will force the external collided node’s Logic State to change.

To facilitate this algorithm, we force the equipment objects to implement an interface that has the method ReactToLogic(). When a state in the Logic Node changes, the ReactToLogic() method is called for the respective devices. What this means is that we don’t have to continuously check the states of all Logic Nodes in the system, but efficiently compute any changes for any devices that may be affected by any new collisions, or state changes.

4.2: Data Structures

Our application makes use of many data structures, ranging from Hash Tables (called Dictionaries in C#), and Lists. Many of our data structures are implemented in the Device objects that hold Logic Nodes. The data structures are chosen based on the quantity of information needed to store, and efficient method of getting such objects. Digital Logic devices such as the Protoboard requires hundreds Logic Nodes to serve its purpose whilst devices such as the 74LS00 (NAND Chip) require exactly 14.

For cases such as the protoboard, we decided to store all the nodes in a Hash Table with a designated key. The rows of the Protoboard all represent the same logic state, as do the columns of the far left and far right of the protoboard. We can store key-value pair based on Lists of Logic Nodes that have the same Logic ID (manually assigned based on the row and or column). In addition to the numerous number of Logic Nodes, the Get operation on the Hash Table usually has a complexity of O(1) which allowed us to quickly retrieve any set of Logic Nodes incredibly fast.

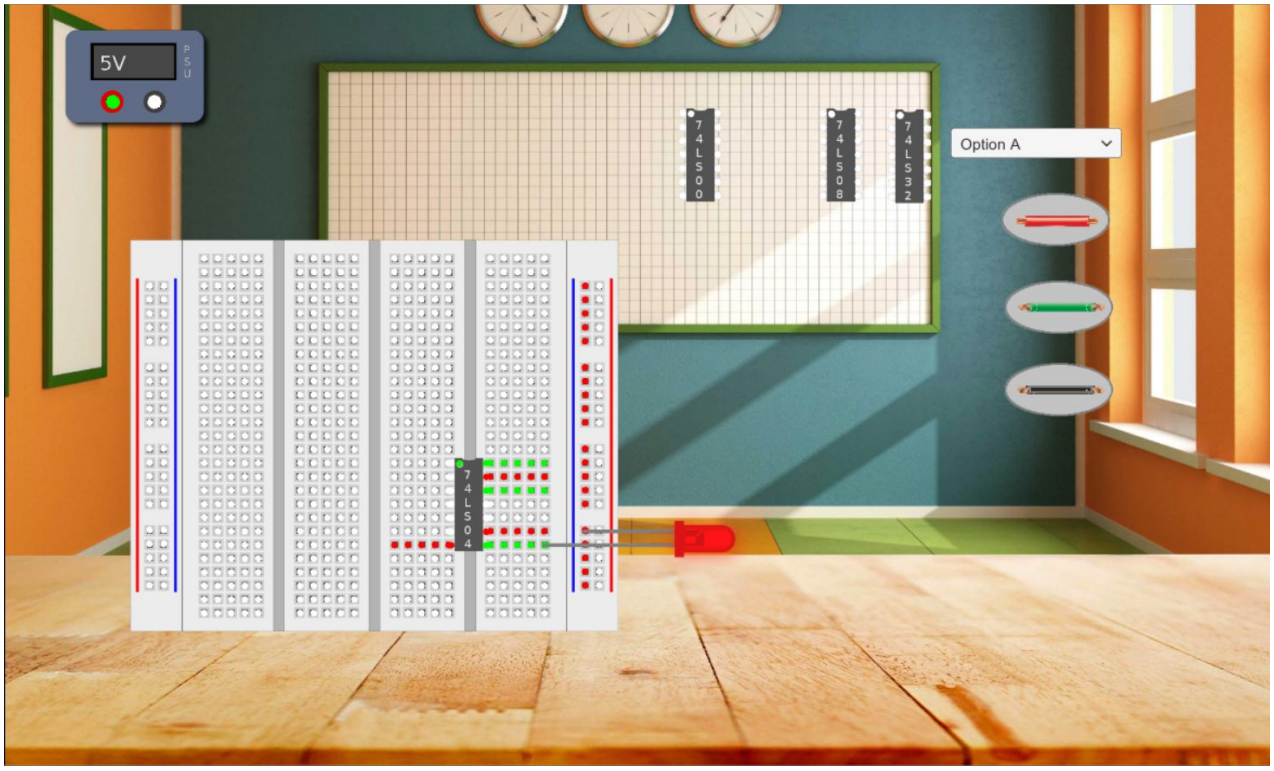


Above, a protoboard with Logic States randomly based on row or column

Other devices, such as the NAND chip, also use Hash Tables, but it is mainly for the convenience of the Get operation with a key input.

5. User Interface Design and Implementation

There have not been any significant user interface design changes from the original preliminary design specified in section 4 of report 1. The initial UI design was carefully developed in order to minimize user effort and maximize ease of use. However, current design calls for a magnifying glass object in order to zoom in on ICs and other electronic components. Keeping the goal of minimizing user effort and design effort in mind, the magnifying glass function will be replaced by a double click command. This decision was made due to the realization that a double click zoom requires less effort than clicking and dragging and object. Another benefit is that this decision reduces the number of components available on the screen, making the user interface screen less cluttered and less overwhelming to the user.



The above picture is a currently underdevelopment view of a particular lab. The lab contains several different equipments, all of which comes from a dropdown menu from the top right. The wires are not yet implemented. The green represents a logic level of HIGH on the protoboard, and the red represents a logic level of LOW on the protoboard. We removed the magnifying glass (that shows the user the device's inner structure and additional information) and adopted to use a double click mechanic instead.

Outside of this change, the rest of the UI specifications are still in development, but have not changed in structure from report 1.

6. Design of Tests

6.1: Test Cases

Because our system is an interactive one, many of our tests will involve working with the circuits and GUIs, making sure that everything executes as expected. The most basic unit tests that we will complete are tests of the equipment, making sure that each piece looks like it should, reacts to clicking and dragging properly, connects to the protoboard, and functions the way that it should. We will also need to complete tests to see that the system correctly creates and deletes users, and allows them to log in to their accounts. These are the most basic functions of our system, so these are the things that we must test rigorously in order to ensure that there are no faults with them. Once our basic components are properly implemented and tested, we can begin to construct and test the more complex functions of our system. The labs will require a large amount of testing as well, as we will need to make sure that the system reacts properly to every circuit that is submitted.

Element testing:

Action	Output
Clicks and drags circuit element	SUCCESS: Circuit element moves along with user's click FAILURE: Circuit element does not move or moves independently of user's click
Places circuit element on protoboard	SUCCESS: Element connects with protoboard, and collision of nodes is detected FAILURE: Element will not attach to protoboard, or collision of nodes in not detected, impairing functioning of the circuit
Test circuit interactions by connecting power source, chips, and wires	SUCCESS: Logic "HI" and "LOW" appear where expected, based on circuit configuration FAILURE: Logic "HI" and "LOW" do not appear where expected, based on circuit configuration

Lab Testing

Action	Output
User completes lab according to the desired requirements and clicks the “Finish” button	SUCCESS: The system analyzes the user’s circuit and determines if the correct output is received FAILURE: The system fails to, or incorrectly analyzes the user’s circuit

Quiz Testing

Action	Output
User completes given quiz and clicks the “Finish” button	SUCCESS: The system correctly grades the user’s quiz by comparing his/her answers to the correct version FAILURE:The system incorrectly grades the user’s quiz, mistaking wrong answers for correct ones, or vice versa

UI Testing

Action	Output
The user clicks on the button to take them to the desired lab	SUCCESS: The user is taken to the lab, where they are given instructions and allowed to complete the tasks FAILURE: The system does not respond to the click, or takes the user to the incorrect place
The user clicks on a button to view grades	SUCCESS: The user is given access to view grades received on previous labs and quizzes FAILURE: The user is unable to view previous grades, or the grades that they are able to view are incorrect

User Management/Login Testing

Action	Output
User is created	SUCCESS: User is created and entered into the database, along with password FAILURE: User is not created successfully, or is not associated with the correct password
User enters username/password to log in	SUCCESS: The user is admitted into the system after entering correct credentials, or denied access if incorrect FAILURE: The user is denied access after entering the correct credentials, or admitted access after entering incorrectly

6.2: Test Coverage of Test Cases

The system testing is a very crucial part of our project, as it serves the purpose of letting us know when a particular piece of code is not acting as expected, or where we may have made faults in design or implementation. Through the use of a wide range of testing, we can ensure that each and every fault is met early on and corrected, and that every invalid input given by a user is handled efficiently by the system, rather than having it react in an unpredicted manner. Our test coverage must not just include inputting the correct answers, but also requires that we input every combination of invalid and incorrect inputs possible, to ensure that the system can handle them and react appropriately.

6.3: Integration Testing Strategy

Our integration strategy will be top-down approach. Because our project is so large and consists of so many different sections, there will often be pieces that are ready for integration, while others are still being modified. Using a top-down approach will allow us to test the interactions of the specific sections that have been integrated, and this will allow us to find faults in the communication between classes, which is heavily relied upon. Errors in communication between classes would not be as simple to see if we were taking an approach that focused more on smaller, individual sections of the the system.

An example of the top-down testing method that we use is when we test the equipment. The protoboard is an example of an upper integrated module, and once that is integrated, we can test the branches (in this case, the wires and chips) step by step, until they too are integrated. After we verify that the protoboard is functioning properly, we test the other objects (the branches in this case) to see if they interact correctly with the protoboard (the main module). Once this has been deemed correct and properly integrated, we can test the functioning of even smaller modules that branch from the chips, such as whether they output the correct logic states in certain scenarios.

7. Project Management and Plan of Work

7.1: Merging the Contributions from Individual Team Members

Merging the contributions was without a doubt a challenging part of this Report. It was very difficult ensuring that all the work had unit formatting, consistency, and a clean appearance. To do so, 2 people were assigned the job of merging contributions- Sagar and Joe. We worked together and created a new document where all the contributions were merged. When merging it, the main challenge we faced was ensuring flow. In our previous Report 2 documents (of each part), different people were assigned different responsibilities. Because of that, there was a lack of flow in the overall document. Once both parts of report 2 were added to this document, we (Sagar and Joe) proofread it and changed up the flow as necessary. Additionally, we added proper headings and the other missing parts to ensure that the report was complete. After we finished the document, we made the rest of the group go over it to ensure that no errors were made.

7.2: Project Coordination and Progress Report

Currently, UC-1 manageEquipment, UC-3 login, UC-4 manageUsers, UC-6 selectLab, and UC-14 giveQuiz are the main focuses. Although none of them have been fully completed yet, we have made significant progress in each one. Some pieces of equipment are functional at this point, such as the protoboard, while chips and wires are being developed. Work is also being done on the database, which will hold each user and his/her password and will be significant in the login process.

Because the labs cannot be fully developed until implementation of the circuit elements are complete, we have shifted our focus to the quizzes, which are currently being constructed. Along with the quizzes, we have also begun developing both the student and instructor GUIs, attempting to ensure that the function of the buttons is correct , and presents the user with the correct interface when prompted.

7.3: Plan of Work

At the time of writing this report, 80% of the Equipment system for the Virtual Lab section of the application is implemented. Equipments such as Protoboard, NAND Chips, OR Chips, AND Chips, Inverter Chips, Power Supply, and LED Lights are finished. From now on to the first demo, the User Interface, including the Student subsystem GUI, and the Administrator subsystem GUI. In addition to the GUI, the sequence for facilitating the Virtual Labs will also need to be implemented. All team members have their respective assignments (shown in the next section), and are expected to finish them before the first demo is presented. By the first demo, we expect the GUI to be completely implemented and at least Lab 1 of our plan to be fully working.

7.4: Breakdown of Responsibilities

The main infrastructure that will facilitate this project is the Unity game engine which is capable of easily rendering 2-Dimensional graphics in a developer friendly manner. The main programming language of choice will be C# with possible integration of others on smaller details. We also plan on using an open source and free database such as LiteDB or RavenDB to upload our grading reports, user data, and chat service.

The most important part of this project is to implement the individual labs to facilitate student learning. To do this, a smaller subset of our group members will implement the “game” logic and create assets needed to create the laboratory experience. This is likely to be the greater of the technical challenge. However, after the initial hurdle of designing a base game logic, creating the actual labs should be fairly straight forward as all the labs use the same concepts of putting together digital circuits.

The second part is to implement a graphical user interface that the administrators and students will interact with to create new users, login, take labs, chat, and view grades in. Parts of this need focus from individual team members but will be implemented by another small subgroup of team members. This part requires the additional specialization of database management which some of our team members possess.

The following table displays the internal deadlines for the project:

Student GUI	3/25
Administrator GUI	3/25
Equipment for Virtual Lab	3/20
Laboratory 1	3/27
Laboratory 2	4/15
Laboratory 3	4/20

7.4.1: Sagar Phanda

- Project Management:
  Responsible for making sure that the project schedule is followed and that deadlines are met (mainly external, but also internal)
- Documentation:
  Responsible for making sure that all documentation necessities are completed, such as: merging all the group work, ensuring proper formatting, and report submission
- Programming:
  Working on the subsystem [with Yiwen] that will allow for lab selection / progress viewing and for the subsystem that will verify academic integrity (UC-6, UC-13)

7.4.2: Khalid Akash

- Duty Assignment:
  Responsible for assigning programming duties to each team member and ensuring that internal programming deadlines are met
- Programming:
  Working on coding [with Vikas] the equipment building portion of the lab and ensuring that the student subsystem is functional (UC-1, UC-7, UC-15, UC-16, UC-17)

7.4.3: Vikas Khan

- Programming:
  Working on coding [with Khalid] the equipment building portion of the lab and ensuring that the student subsystem is functional (UC-1, UC-7, UC-15, UC-16, UC-17)

7.4.4: Joseph Cella

- Programming:
  Responsible for coding the administrator substem / database [with Yiwen], which will be used to allow the instructor to edit / reset user passwords or entirely delete users (UC-4).

7.4.5: Dhruvik Patel

- Programming:
  Responsible for coding the first lab, Lab 1, which will include a tutorial and a pre/post quiz to help the users get acquainted with the system

7.4.6: Yiwen Tao

- Programming:
  Responsible for coding the administrator substem / database [with Joseph], which will be used to allow the instructor to edit / reset user passwords or entirely delete users (UC-4).
  Additionally, working on the subsystem [with Sagar] that will allow for lab selection / progress viewing and for the subsystem that will verify academic integrity (UC-6, UC-13)

8. References

1. Professor Marsic's website, detailing the requirements for report 2
   http://www.ece.rutgers.edu/~marsic/Teaching/SE/report2.html

2. A past project of the same topic, that was used as a guide
   http://www.ece.rutgers.edu/~marsic/Teaching/SE/report2.html

3. Used to detail the interaction diagrams:
   https://medium.com/omarelgabrys-blog/object-oriented-analysis-and-design-design-principles-part-6-b78e2b9da023

4. A good overview of top-down testing
   http://extremesoftwaretesting.com/Techniques/TopDownTesting.html

5. A site that was used to develop an understanding of Unity
   https://unity3d.com/learn

6. Used to gather information about the SN74LS00 chip that we will be simulating
   http://www.ti.com/product/SN74LS00/technicaldocuments

7. Provided information about several gates that will be used in our simulations
   http://www.futurlec.com/74LS/74LS04.shtml
   http://www.futurlec.com/74LS/74LS08.shtml
   http://www.futurlec.com/74LS/74LS32.shtml

8. Provided information about data structures that will be used in this project
   https://www.geeksforgeeks.org/data-structures/