Sprint 1

Requirements Engineering:

Functional Requirements:

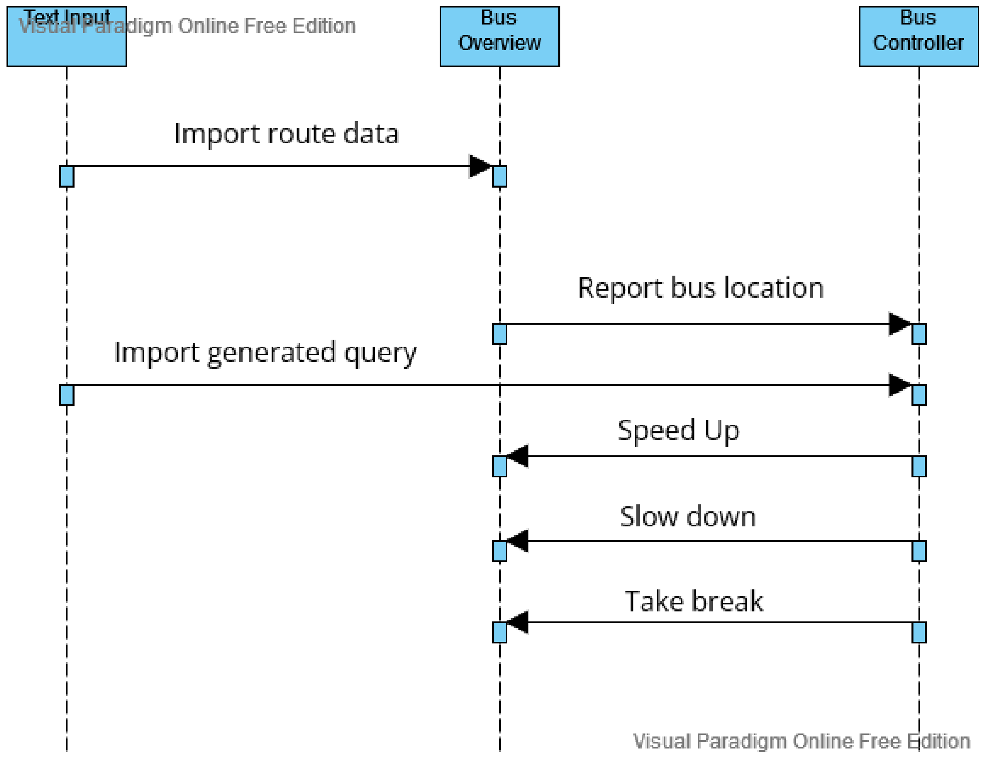
- Create overall object hierarchy
- Setup project structure, with several classes and interactions
- Develop various helper methods to communicate data, calculate bus times and control bus movement
- Create methods to stop, start, speed up or slow down bus routes
- Allow initialization of bus routes using a text file

Non-Functional Requirements:

- Scalability
  - Program must be able to work with varying amounts of bus routes, busses and passengers
- Flexibility
  - The program must be able to handle future changes to requirements, such as varying time constraints, traffic, driver breaks, ect.

System Modeling:

Sequence Diagram:

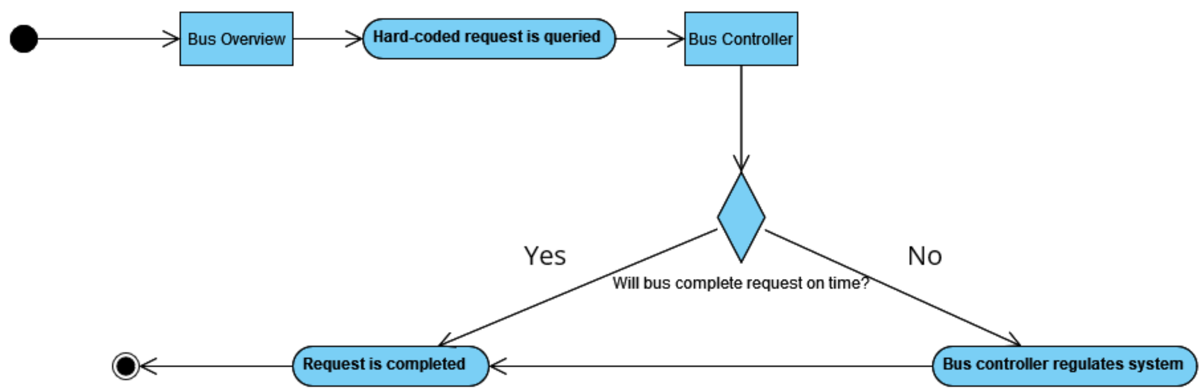


Our original plan for this sprint's sequence diagram includes 3 main interfaces. The first of which is the text input reader, which is responsible for importing bus routes, and executing manually entered queries. Bus routes are sent to the overview for organizational purposes, and queries are given to the controller for the algorithm to compute.

Next, the bus system may report location, and bus status to the bus controller.

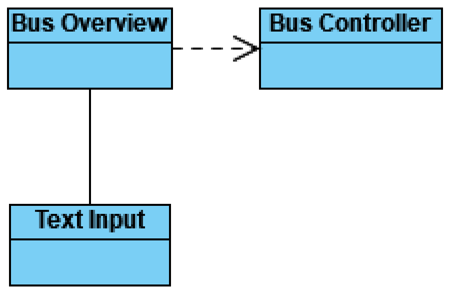
Finally, the database is responsible for making decisions for the bus route to operate correctly. In order to do this, it must take the requests generated by the simulator, and use an algorithm to determine bus activity. It must also take in the location of the buses in order to accurately estimate bus times.

<u>Activity Model:</u>


Bus Overview
Hard-coded request is queried
Bus Controller
Yes
No
Will bus complete request on time?
Request is completed
Bus controller regulates system

Architectural Design:

<u>Box and Line Diagram:</u>

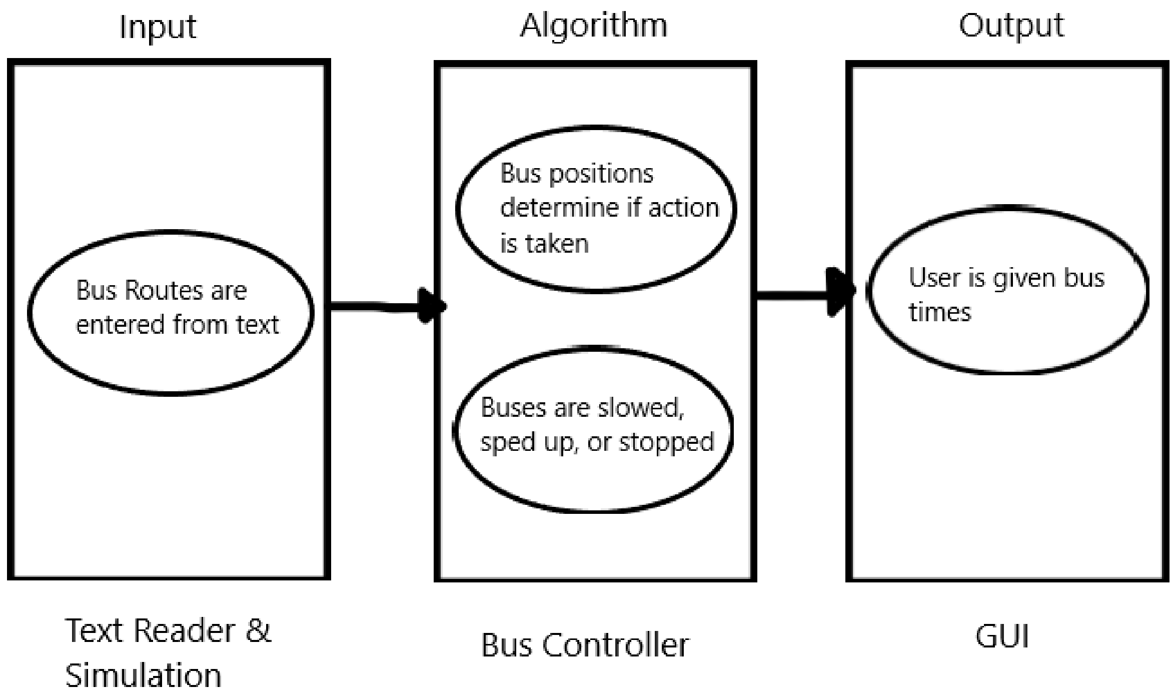


At this stage, there are only two proper classes that dictate the system's operation.

The Bus Route Overview contains the raw data from the buses and routes. For the buses, this means their IDs, operating status, assigned routes, and current locations. For the routes, this is the list of bus stops, as well as their unique IDs and connection times. It is important that this class is made to be compatible with varying sizes of inputs, so that the project maintains proper scalability. This class also includes the text input for the time being, which will be used for testing purposes.

The Bus Controller handles all activities done to manage the bus system. This mainly includes regulating speed and breaks of buses to make sure they can reach any stop in their route in the given amount of time. This class is the 'brains' of the entire system, and contains the important algorithms to correctly implement the software's requirements. It is important to note that this may be the area of greatest issue, as it contains some of the most challenging and potentially problematic code. We must use several techniques to ensure correct operation of this specific class.

Software Architecture Model:

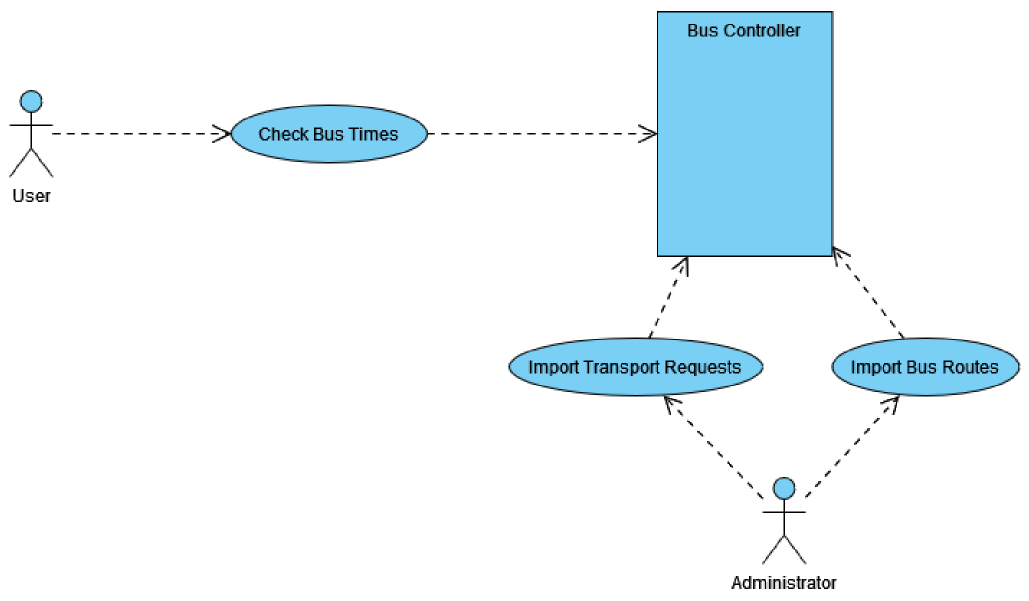

Input
Algorithm
Output
Bus Routes are entered from text
Bus positions determine if action is taken
Buses are slowed, sped up, or stopped
User is given bus times
Text Reader & Simulation
Bus Controller
GUI

Specification and Implementation:

Object Class:

Between the 4 main structures of the system, the bus overview is the object holder class. This class determines the main structure of all of the objects, including buses, routes, and stops. It also generates the distance matrix that handles bus times, and gives unique identifiers to assist the bus controller in its algorithms.

Use Diagram:



Use Case Description Report:

System	**Rutgers Bus Route Scheduler**
Use Case	Transports a passenger from any bus stop to another in under 30 minutes
Actors	User, Administrator
Description	A user may access the database and check when buses will arrive at a stop, while the administrator may configure bus routes and generate transport requests
Stimulus	The administrator submits a request, to which the controller responds
Response	Bus controller regulates pace of bus route to ensure all passengers will reach their destination on the given time
Comments	At this point, only one request may be handled at a time until the simulator has been added

Software Testing:

Testing Process:

- Because there is no user input at this phase of development, testing options are limited. Instead of user testing, we instead use a method of manual input from a text file to test our controller algorithm.
- The Text Input is tested using basic methods, like an output to console. When we import routes to the bus overview, they are then repeated back to the console output to ensure the copying has been done correctly.
- When queries are added through the text input, they are restated in the console to ensure the queries are recognized correctly.
- Instead of showing bus results to a GUI and user, results are printed to the console along with their process in the algorithm. For example, if a query requests a passenger from stop A to B, then the console will output if the query is possible without any interference. If interference is needed, then the process the algorithm used is printed, along with the new results of the bus time.

Evaluation:

- This sprint was completed more or less as expected, with only a few design changes that had to be replanned. First, the way in which distances were calculated had been changed. The original plan consisted of a doubly-linked list of bus times that looped in order to calculate the amount of time needed in a route. This would be true for each route in order to accurately find the total time from any given stop to another. This, after careful consideration, was changed to a distance matrix that held a key of all bus times to any other stop. This would then account for the route needed, by finding a path using the shortest route containing both bus stop IDs. This had the added benefit of changing many smaller route keys, into one, more efficient and centralized object. This way, it may be referenced by any query.
- In the next sprint, we aim to add the simulator which will allow generation of passenger requests to accurately handle a constant load of passengers needing to reach a destination. Through this sprint, user input will still be limited, but now a request may be inputted either manually, or generated automatically. The user is still only capable of viewing bus times.
- Software Testing went smoothly in this sprint, with few issues. The bus break method needed more configuration after testing, to make sure buses would space more evenly if getting congested on a route. Other than this issue, there was some debate over how much a route should be sped up or slowed down,

with an eventual conclusion that led to consistent results while still being reasonable.

Sprint 2

Requirements Engineering:

Functional Requirements:

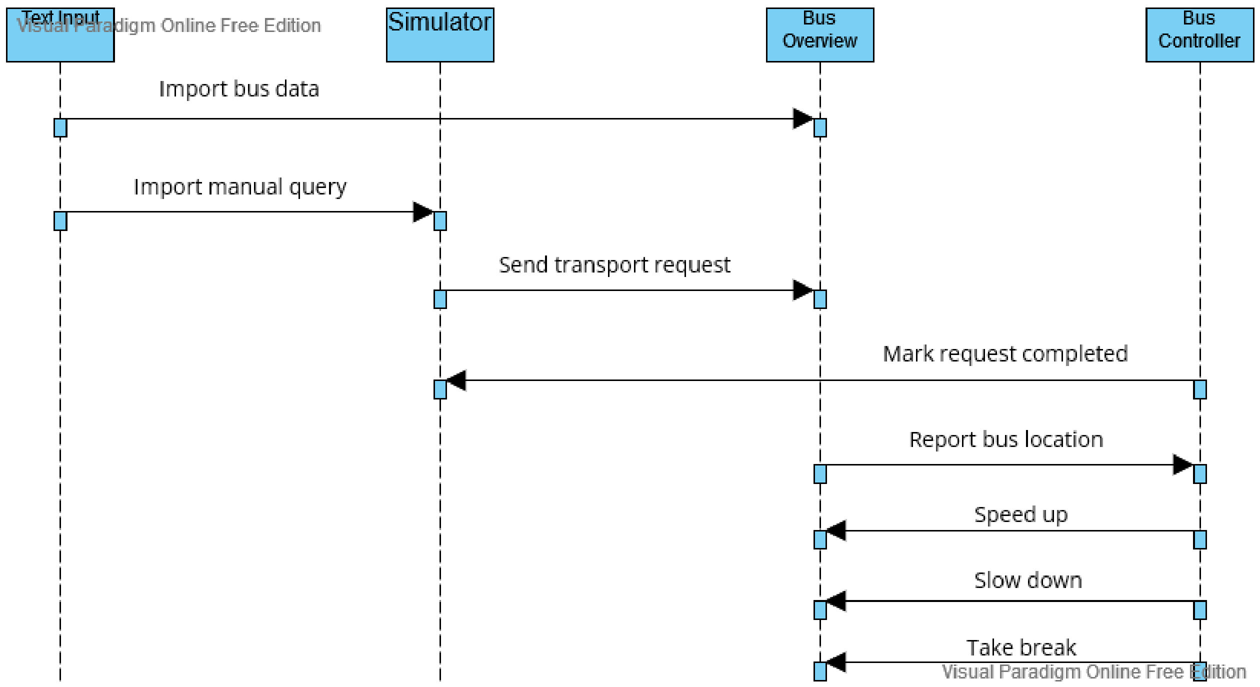
- Create bus simulation class
- Create test methods to simulate a load of passengers requiring movement
- Use progress bar to show the movement of buses between stops
- Use worst case scenarios to make sure all passengers can move in given time

Non-Functional Requirements:

- Scalability
  - Simulation must be able to be scaled, with more passengers adding strain to the system
- Efficiency
  - Simulations should not take up too many resources, performing minimal calculations and only refreshing data when needed
- Documentation
  - Simulation should be well documented
    - Allows causes of error to be identified more easily
    - Allows changes to be made quicker
    - Helps new members trying to understand previously written code

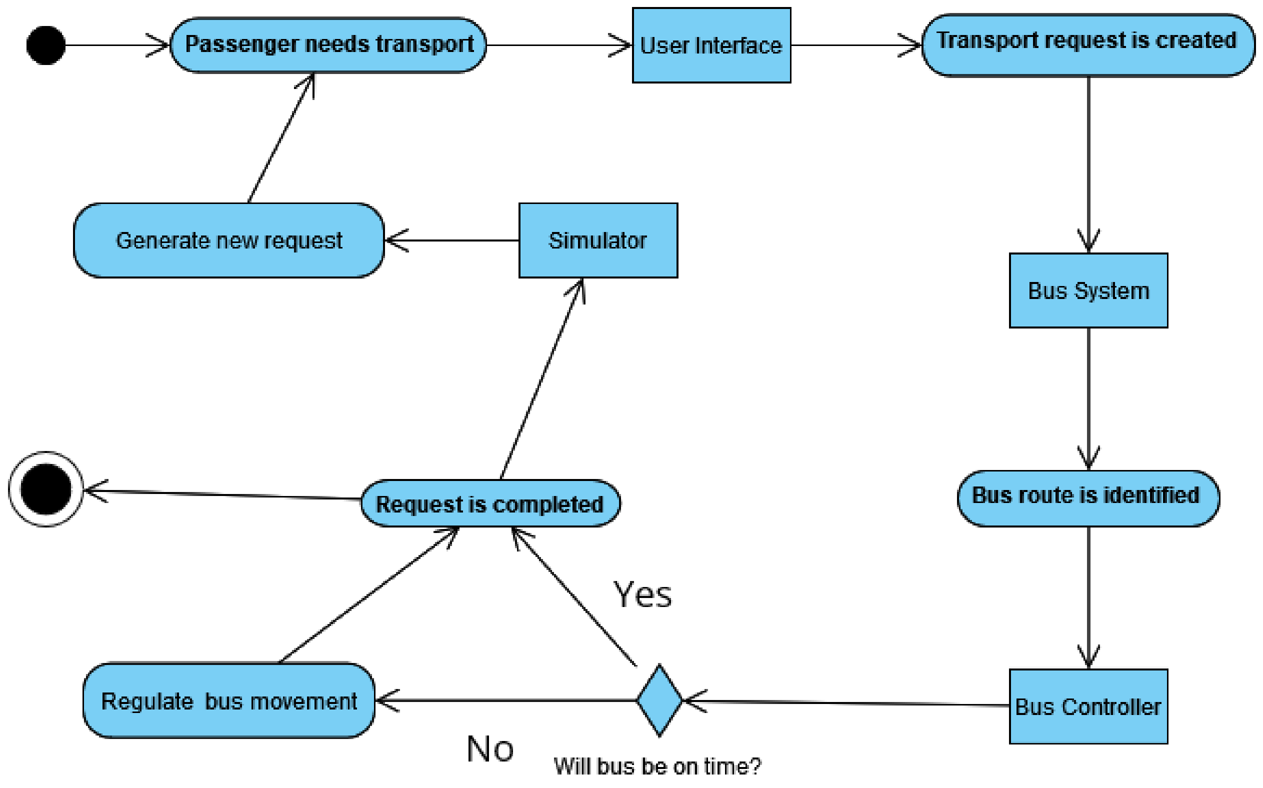
System Modeling:

<u>Sequence Diagram:</u>



In this sprint, the simulator is added in order to create passive bus movement, as well as generate transport requests instead of users. This allows more thorough testing of the program, as well as brings the project one step closer to operational in the real world. The main purpose of the simulator is only to handle the problems associated with not actually being a deployed product. Because the program doesn't actually have any busses reporting location, the simulator instead uses deltatime to change their location based on their speed. Also, user requests would not actually be generated by the program itself, but instead done automatically through users of the application.

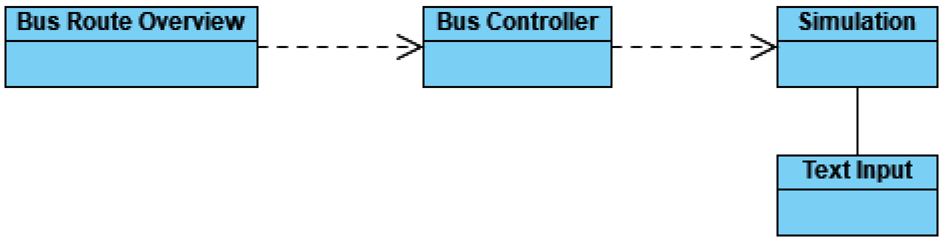
Activity Diagram:



This updated activity diagram shows the role of the simulator in how stimuli are handled. The basic logic behind the operation remains unchanged within the bus controller. However, now generation of requests is handled by the simulator rather than being manually inputted. Also, the system is capable of running multiple requests in a row, through having buses passively move through their routes in downtime.

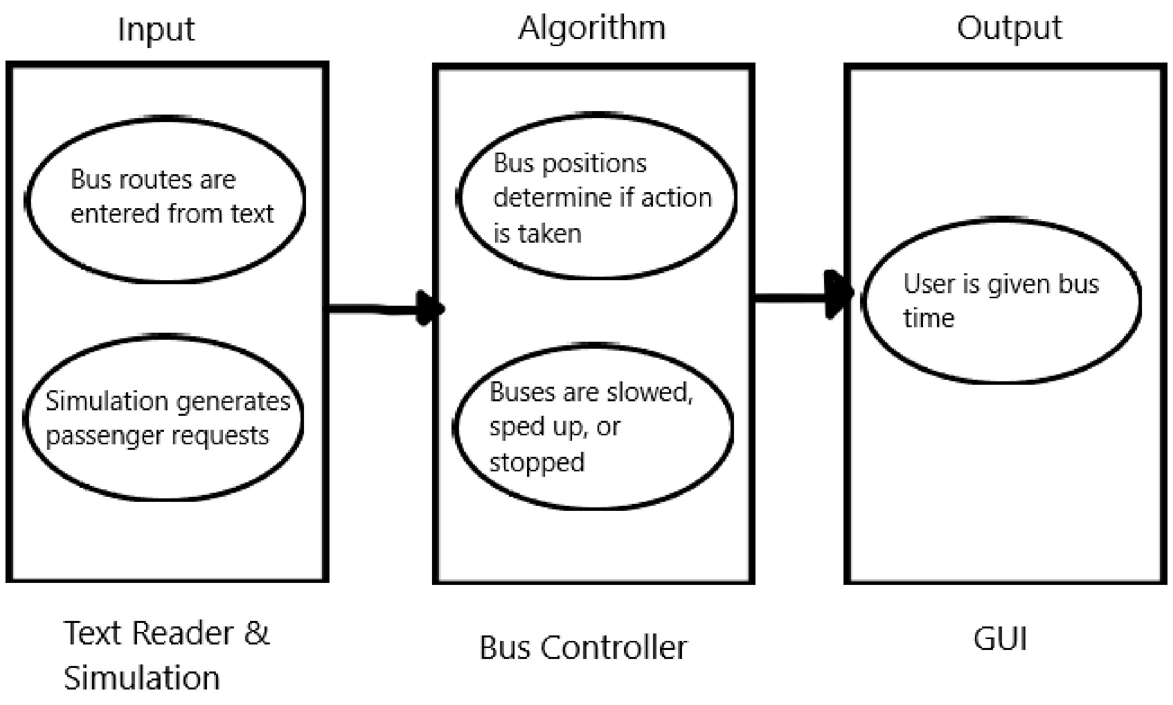
Architectural Design:

Box and Line Diagram:



With the addition of the simulator class, the text subsystem has been changed to report information directly there instead. The bus route data is still sent to the bus overview, but the information regarding bus locations and requests are handled through the simulator instead.

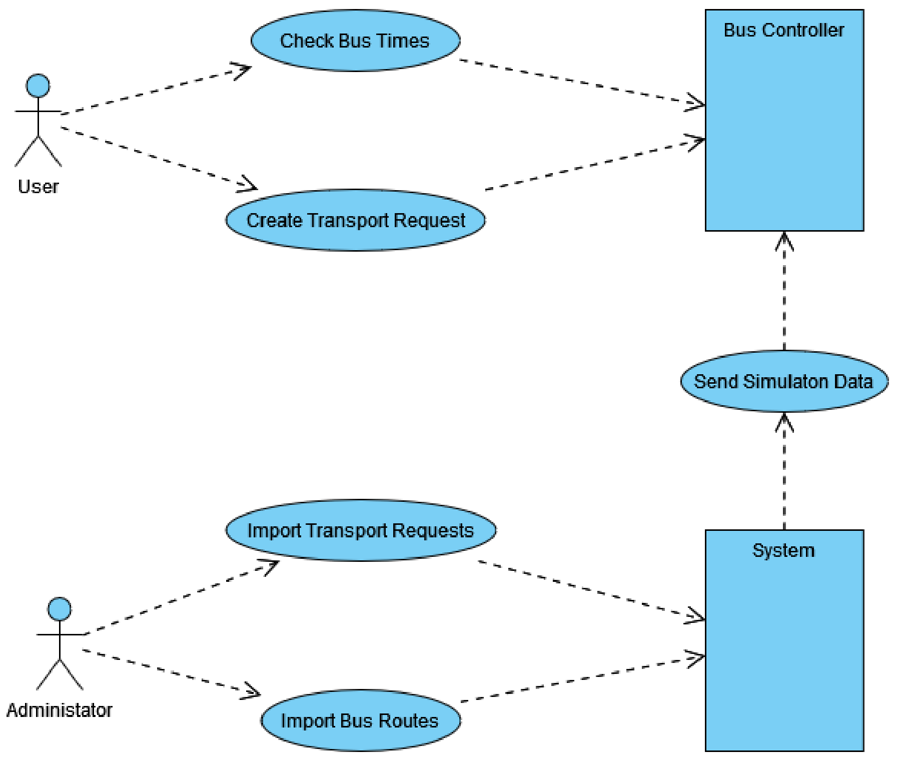
Software Architectural Model:


Input
Algorithm
Output
Bus routes are entered from text
Simulation generates passenger requests
Bus positions determine if action is taken
Buses are slowed, sped up, or stopped
User is given bus time
Text Reader & Simulation
Bus Controller
GUI

The main architectural change in this sprint is that input is now generated from the simulator, rather than roley handled by the text input. This has no effect on the actual calculations done by the program, or the output of the system as that is only dependent on the bus overview and controller.

Specification and Implementation:

<u>Use Diagram:</u>



The box marked ‘system’ in this diagram represents the simulator, which now can send information regarding requests and bus locations to the bus controller. The rest of the use diagram remains unchanged. Although the simulator is not actually human, it still acts as an actor as it represents both the bus drivers and potential passengers.

Use Case Description Report:

System	**Rutgers Bus Route Scheduler**
Use Case	Transports a simulated passenger to any bus stop, from any bus stop, in under 30 minutes.
Actors	User, Administrator, Simulator
Description	A user can only view the bus data through the console at this point in development. An administrator may add bus data to the text file, as well as add queries through text for testing. The simulator portrays bus divers through passage of time, as well as generating automatic requests instead of the user.
Stimulus	The simulator generates and sends a request to the bus controller, along with current bus data.
Response	Bus controller uses given data to calculate times needed for bus to fulfill request in under 30 minutes
Comments	Although the program is not fully functional with input and output at this point, it is now possible to test requests in succession with the addition of the simulator and passage of time

Software Testing:

Testing Process:

- The testing process for this sprint was pretty straightforward after completing the simulator class. The first step was to import routes from the text file, and verify their existence in the program through the console. This part had not changed since sprint 1, and there were no issues. Afterwards, instead of reading a manual query from the text file, we would allow the simulator to generate a random request, while observing how it was handled. This was done by checking the bus times and locations through the console at several points in the process. We decided to represent the progression of time using a constant refresh rate, that increments all buses by two minutes upon refresh. After a request was completed, we let the simulator generate a new request, and repeated the process to verify the functionality of repeated queries.

Evaluation:

- The implementation of the simulator went smoothly and with little error. The only issue that became relevant was the usage of real-time simulation. Since the requirements specified a bus being able to make a call in 30 minutes, observing a bus make the entire route could take up to that amount of time. In testing, this was an issue since one request one be extremely time consuming. We decided to put the simulator on a time scale to prevent tests from taking 30 minutes. After some thought about the scale being slow enough to observe, but fast enough to not take too much time, we decided that each refresh would represent 2 minutes. This way, the bus information could still be observed when refreshed, but also could be tested quite quickly.

Sprint 3

Requirements Engineering:

Functional Requirements:

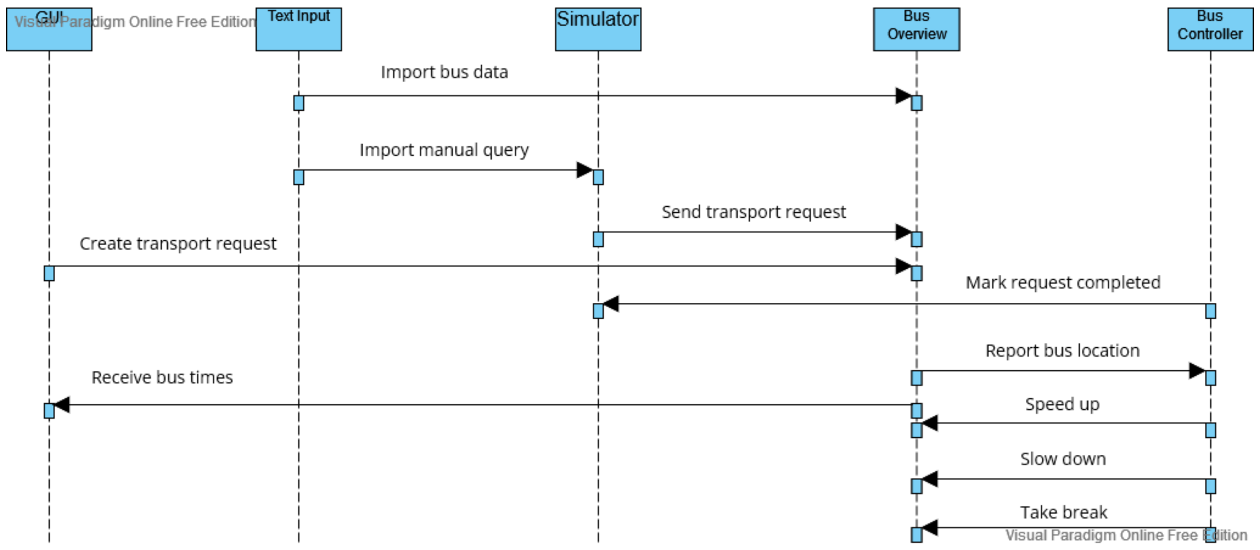
- Implement Exception Handling
- Implement Minimal GUI to display and refresh results
- Handle incorrect text file input
- Give error if passenger takes longer than 30 minutes to reach destination

Non-Functional Requirements:

- Usability
    - Program must be easy for a user to interact with, and check estimated time to their stop
- Effectiveness
    - Program must be able to produce results for the amount of operations being executed
    - e.g. if the time constraint is lowered, then the program should operate less intensively
- Reliability
    - Program should be well optimized with little to no faults in the system on average
- Fault Tolerance
    - In the event of a fault, the program must be able to identify the error, and correct it with minimal effect on operation

System Modeling:

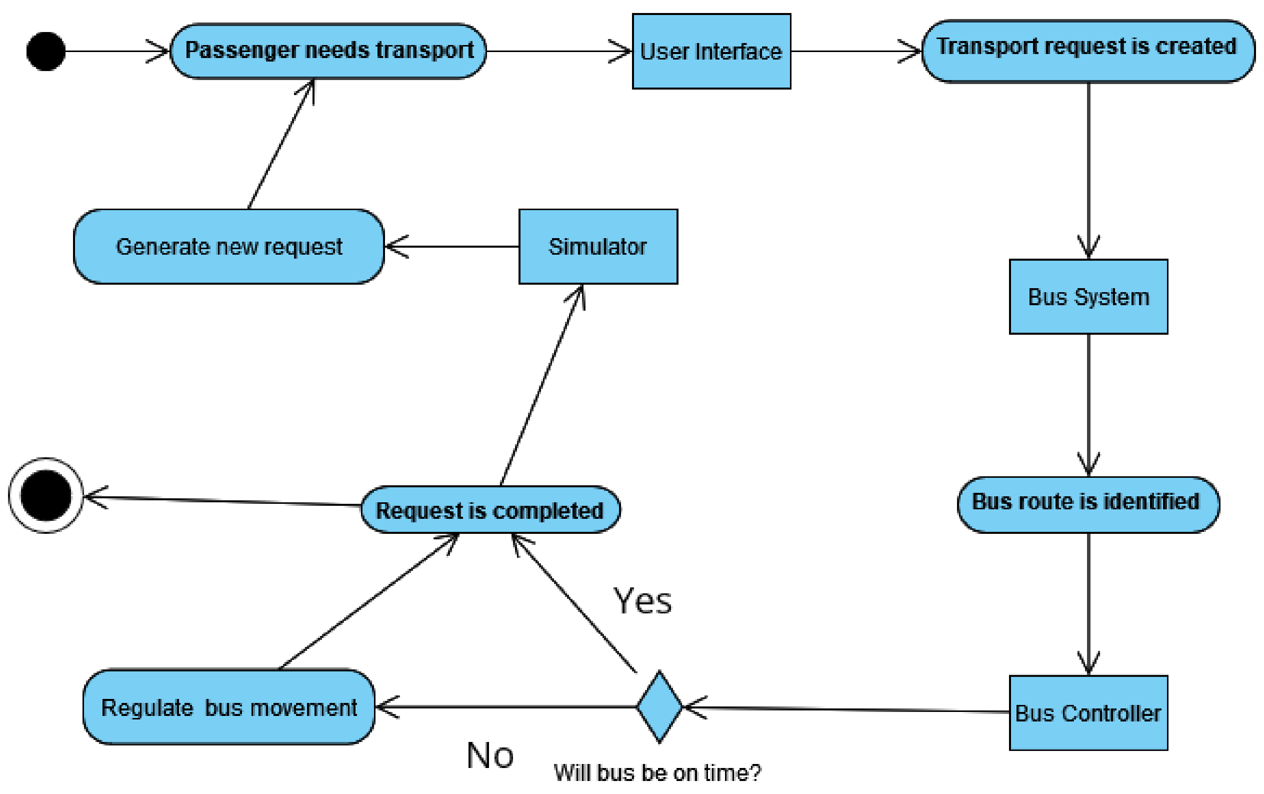
Sequence Diagram:



This sequence diagram is updated for the addition of the GUI, which allows the user to interact with the bus controller by adding their own request. In addition to this, the user may refresh, and view the bus data including bus times, locations and general information.

Not marked on this diagram, but exception handling has also been added within the bus controller, which can identify and attempt to correct faults in programming. If the error was not solvable, then an error message will be displayed in the console for an administrator to see.

Activity Diagram:



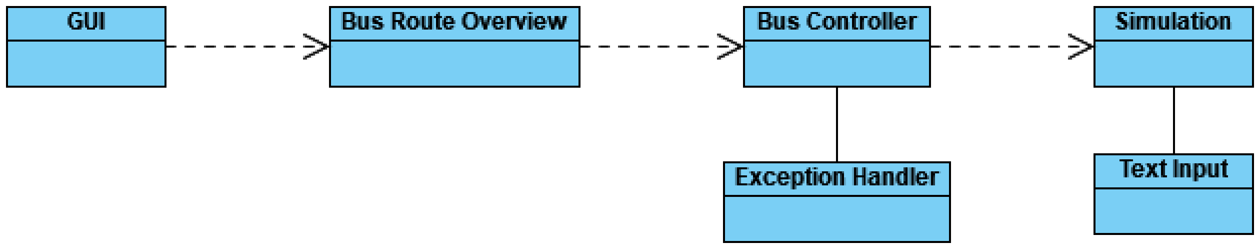
The activity model does not change between sprint 2 and 3, since all functionality is roughly the same. The only main difference that is not shown here is the addition of error messages, which do not significantly change the flow of the program.

If an error is found, then the program will output a message to the administrator to notify of the issue, as well as be prepared for future occurrences.

Other than this, we see that the simulator method has been completed to handle the generation of new transport requests, as well as the possibility for a user to create their own request. This now makes the concept of adding manual inputs from the text file obsolete, and the functionality can be discarded in order to mitigate issues in the future. When code is obsolete, it is best to eliminate it from the program to simplify code for understanding, as well as eliminate possibility for error.

Architectural Design:

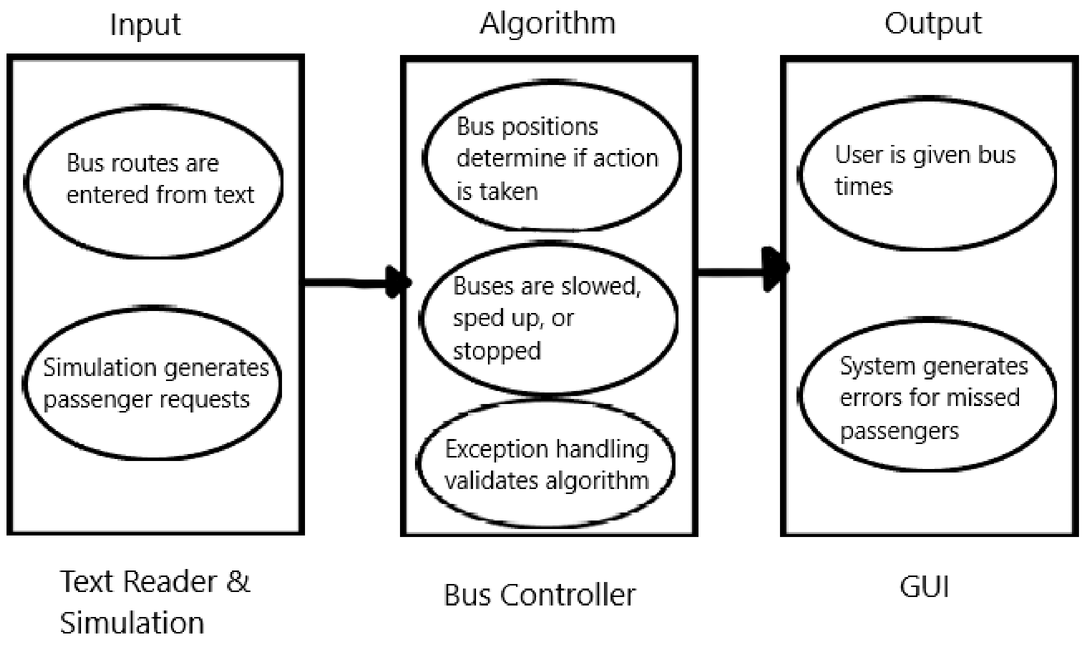
Box and Line Diagram:



Here we see the final additions to the system: the GUI and Exception handler. The GUI is a stand-alone system that allows a user to refresh and request data, as well as push a new transport request to the queue. This eliminates the need for testing using the text input, which is now only used for route configuration.

The exception handler is a subsystem of the bus controller, which has the capability of diagnosing non-fatal errors in the system. If the fault cannot be fixed through exception handling, then an error will be sent to the console instead as a mitigation effort to keep the system operational.

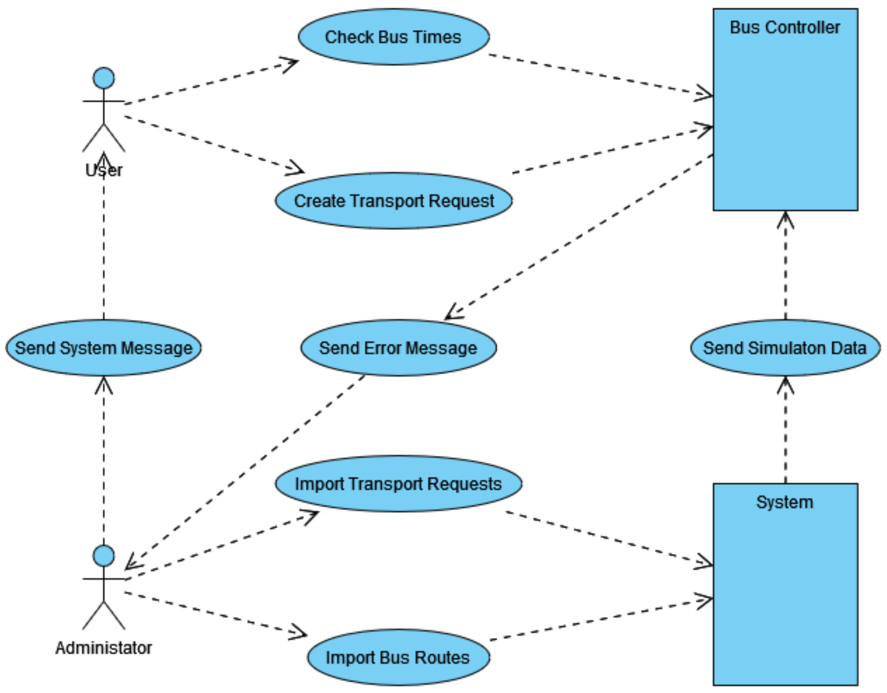
Software Architectural Model:


Input
Algorithm
Output
Bus routes are entered from text
Simulation generates passenger requests
Bus positions determine if action is taken
Buses are slowed, sped up, or stopped
Exception handling validates algorithm
User is given bus times
System generates errors for missed passengers
Text Reader & Simulation
Bus Controller
GUI

The main change to the architecture of the system comes in the addition of exception handling. Due to this functionality added in the bus controller system, errors can be identified and mitigated before they cause an issue. If this is not possible, the program is not capable of generating error messages to the console, which will allow administrators to address the issue going forward.

Specification and Implementation:

Use Diagram:



This Use Diagram has only two main differences from the previous versions of the system. With the implementation of the GUI, the user will now check bus times and create transport requests through the GUI, rather than operating the console directly. Since in practice the user is unable to actually access the console, this step finally makes the system an operational, marketable product.

Also, the exception handling subsystem now adds two new connections. First, when the bus controller encounters an error, it will be caught and sent to the administrator in order to mitigate the effects. The main point of this is an unexpected input, or an incorrect usage of bus stops that do not correct. In the event of these erors, the system will reject the transport request and continue with an error message. In the event of a fatal error, the system will be rendered unoperational, but minor errors may be diagnosed and fixed in the future. Also, the administrator will now have the ability to communicate directly with the users if needed, through the addition of the GUI.

Use Case Description Report:

System	**Rutgers Bus Route Scheduler**
Use Case	Passenger may send a transport request through a GUI to get to any bus stop, from any bus stop, in under 30 minutes
Actors	User, Administrator, Simulator
Description	Users may create a transport query to reach their destination, as well as request bus information to check the status of their ride. The administrator can manage the system by adding manual inputs of bus stops, as well as handle exceptions caught by the bus controller. Finally, although not a human actor, the simulator is used to create random requests to test the program, as well as keep buses moving and handle the passage of time.
Stimulus	User sends a request, which the bus controller will fulfill. If there is no user request active, the simulator may generate a request instead to keep the bus controller active.
Response	The bus controller will regulate the movement of the buses, first prioritizing user requests, then only after working on simulated requests.
Comments	With the addition of exception handling and the GUI, the program must be able to handle a variety of inputs and outputs to operate smoothly. This means rigorous final testing must be taken to ensure all operations can work together.

Software Testing:

Testing Process:

- In order to test our program, we made use of the built-in simulator to generate requests for the bus controller to fulfill. Then, by carefully observing the operation status through both the console and GUI, we were able to validate the smooth operation of the program. To test exception handling, we were able to add faulty code that would intentionally trigger an error within a try-catch statement to validate that errors would be successfully handled. Lastly, to verify the operation of the GUI, we were able to contrast the displayed information with the known correct information written into the text file at startup. Then, we generated transport requests through the GUI, and carefully monitored the process of the bus movement to validate that it was handled correctly. Then when finished, the program would continue handling simulator requests as expected.

Final Evaluation:

- Overall, the system worked as originally planned with few detours.
- With the addition of exception handling, we are able to verify that the program will run even after encountering common, non-fatal errors. This was done simply through mitigation techniques, try-catch statements, and prevention through error messages. After the deployment of the program, all errors would be noticed by an administrator, and corrected in routine maintenance. The main implementation of this feature has to do with invalid routes, where invalid input of stops will be rejected.
- The simulator handles the passive movement of buses, as well as the passage of time that would originally be signified by the actual change in position of the buses. Instead, we use an ingrained timer to move the buses in accordance with their speed and break times.
- The GUI was implemented with fewer functionality than desired in a final result. Due to the nature of the project, there is no way for a user to have an application to work in tandem with the database. Instead, we have a built-in GUI that will allow a user to refresh time, updating the results. The code for the GUI was reused from a previous project, in order to save time and resources. Also, transport requests are generated through this built-in GUI, which would instead be on a separate app after the actual deployment of the application.
- Despite the diverse range of commands that the program must handle, the system is able to operate perfectly under load. This is important as code must be tested for a variety of scenarios in order to mitigate changes after the initial deployment.

Incorporate at least one dependable attribute:

The dimensions to dependability identified in the textbook are Availability, Reliability, Safety, Security, and Resilience.

Early on, we paved the road for a smooth project development, avoiding the introduction of accidental errors into the system during software specification and development.

Whilst undergoing the software development process, we kept in mind reliability and incorporated dependability properties at each and every step of the way, making sure that the program will not fail in normal use.

At the end, we designed verification and validation processes effective in discovering residual errors that affect the dependability of the system. Along with proper documentation and comments, we wrote the code up using well organized objects and methods. Thus, we designed the system to be fault tolerant so that it can continue working if things went wrong. Relating to bug fixes, handling, and errors undergoing each sprint, we made sure that whenever incorporating a major change to the program, we documented the updated code and tested incrementally, thus allowing us to return to our previous version and perform regression testing if anything went wrong or gave us trouble.

Reliability through proper response time was a dependable attribute that was incorporated and heavily stressed when developing this project. We wanted to produce something that was dependable, with high reliability. The bus route management system we were tasked with creating required that no student should have to wait more than 30 minutes to get to a class on any campus from any other point on the New Brunswick campus. With our finished product, this was achieved, and the program was stable and ran whenever demanded when we did user tests with the GUI we implemented. Obviously, it is not possible to create a system that is 100% completely dependable, but through our rigorous planning and leaving days to do testing after completing our project, we were able to incorporate dependability into our overarching developmental process.

Incorporate at least one Project Management concept:

In terms of project management, risk management is one of the most important aspects of successful project development. Risk management involves anticipating risks that may impact the project schedule or quality of software.

The types of risk identified in the textbook are Estimation, Organizational, People, Requirements, Technology, and Tools.

As a project manager, the last minor project that we produced, gave important insight into how this group works as a team and what we need to improve on. Of the above types of risk we recognized which risks were the most detrimental to our development process, and thus this project we focused mainly on **lowering estimation, organizational and people risks**.

In terms of estimation risk, the estimated time required to develop the previous minor project was underestimated quite a bit. Our group started by agreeing to meet once a week once the minor project was announced, but towards the end, we realized that this work schedule didn’t give us enough time to properly complete the project. In order to maintain overall morale and efficiency of work, we discussed and determined how long we were going to work on this project every week. With a newfound understanding of what it takes to work together and deliver a project like this, we met up and worked on the project multiple times a week for a little bit each time instead of saving it all for the end only to find out that there isn’t enough time.

In terms of organizational and people risk these concepts come hand in hand when it comes to project development. Dividing up the project and making sure everyone had equal roles and responsibilities was something that we struggled with for the previous minor project. We each came in with different skill sets, and sometimes group members were simply unavailable and canceled during days we were supposed to meet up and work together on the project for long hours due to other classes.To minimize this kind of risk again in our project development process, we decided to create a Discord team group chat to communicate more efficiently and constantly check in with each other with updates and responsibilities. We also outlined who was in charge of what part of the project better and set boundaries early in the development process to divide the work. With this, we could better communicate with one another to do work online without having to meet physically even if not everyone is there. This risk was relatively easy to solve with a little bit of project management and discussion, since my group now knows each other better and we're more familiar with each other's skills after the completion of the minor project.

With project management and these concepts of risk in mind, we discussed and employed various tactics and strategies in our development process that would allow our group to work better together and produce a proper finished product. From risk identification to risk analysis, into risk planning and monitoring, employing these project management concepts allowed our group to work together safely, and more effectively when compared to our previous project with minimal project management.

Incorporate at least one concept from Advanced SWE topics:

Amongst all of the Advanced SWE topics and concepts that are covered in the textbook, software reuse is arguably one of the most practical and efficient strategies employed in project development. By building off certain aspects of our code from our preexisting minor project components and systems, we can make our project development processes not only more efficient but also less risky. By using old code that we know works well, we can take advantage of software reuse and modify existing code as needed and integrate pieces into our new system. Although there is a wide variety of approaches to software reuse, we mainly focused on reusing a stand-alone application from our old minor project system, which refreshed data and let the user test the system as much as he/she would like.

The main example of our new system's incorporation of software reuse would be the implementation of a basic graphical user interface. This simple GUI allows the user to interact with our program, simulating our student bus route management system. The GUI was also a feature of our last project, in which we created a pop-up window displaying information with buttons to interact with the system. This simple GUI serves as a way to simply refresh the data of our system and display it so that we can test it numerous times to see if anything seems incorrect or suspicious in our system. Since the code for the GUI was pretty much a stand-alone application system, the general purpose system could be adapted for use in this specific application with relative ease. With this incorporation of software reuse into our project development, the cost of the time spent adapting and configuring the reusable software components is without a doubt, lesser than the cost of writing new code. All in all, software reuse is a core concept covered in the advanced SWE topics, and incorporating it into our project was invaluable, accelerating development, increasing dependability, reducing process risk, and more.