SYSC 3303 Real-Time Concurrent Systems

Elevator Control System and Simulator

Project Final Report

Carleton University

Instructor

Dr. Gregory Franks, greg@sce.carleton.ca

Group Members

Abdelrahim Karaja (101187105), AbdelrahimKaraja@cmail.carleton.ca
Ben Kittilsen (101101290), benkittilsen@cmail.carleton.ca
Peter Tanyous (101127203), petertanyous@cmail.carleton.ca
Millan Wang (101114457), millanwang@cmail.carleton.ca

April 12, 2021

Table of Contents

List of Figures

Responsibilities & Contributions

Abdelrahim

Iteration 1 – February 5th

-Data analysis of given elevator time measurements to find average times of significance for each elevator
-Designed and created TimeManagementSystem class to generate elevator loading, offloading and movement times
-Created test cases for TimeManagementSystem (JUnit) to ensure operational correctness
-Added UML Class diagrams for TimeManagementSystem
-Updated README information

Iteration 2 – February 19th

-Updated TimeManagementSystem class methods to account for velocity and acceleration of the elevator as opposed to random time generations regardless of state
-Implemented logger calls in TimeManagementSystem to output information about generated times
-Created report on data selection and analysis process to outline the key details and times pulled from the provided elevator timings
-Adjusted program to make use of only one TimeManagementSystem instance for all necessary systems
-Updated UML class diagrams for TimeManagementSystem
-Updated test cases to account for new additions in TimeManagementSystem

Iteration 3 – March 12th

-Updated TimeManagementSystem and Elevator Classes to work together and make use of all generated times when getElevatorTransitTime() is called
-Altered logger class to implement the ServerLogger class for inter-process communication
-Designed and created ServerLogger Class to enable logging of all program events through UDP communication in one location (same console) so running the program using separate processes could be documented and tracked
-Updated JavaDoc for TimeManagementSystem and ServerLogger Classes
-Updated JUnit testing for TimeManagementSystem and Elevator Classes
-Added JUnit testing for new functionality
-Updated README file
-Updated preexisting and added new UML class diagrams

Iteration 4 – March 26th

-Updated TimeManagementSystem methods to function correctly with calls from ElevatorSubsystem
-Updated TimeManagementSystem values and documentation based on assumptions about travel distance and acceleration/deceleration times
-Designed and created a basic GUI class to display all relevant elevator and floor information
-Updated preexisting and added new UML class diagrams for classes worked on
-Debugging/Testing of code areas (JUnit testing)

Iteration 5 (Final Demo) - April 2nd

-Designed and created GUI for FloorSubsystem to display all floor buttons in building, along with a colour coded display of which buttons have been clicked and which are not active
-Designed and created GUI for ElevatorSubsystem to display status and location of each elevator along with the buttons in each elevator. Including a colour coded system to display if the elevator is moving, stationary, open, closed or experiencing any errors
-Updated preexisting and added new UML class diagrams for classes worked on
-Debugging/Testing of code areas (JUnit testing)
-Integration testing

Ben

Iteration 1 – February 5th

- Designed and Implemented the Elevator subsystem and elevator. For this iteration the elevator subsystem only had 1 elevator. The elevator subsystem had an instance of the elevator that the elevator system moved based on next floors given by the scheduler.
- Created test cases for the elevator simulating the elevators movement.
- Integrated the Elevator Door class with the elevator. This was used to simulate the doors opening/closing, by sleeping the thread based on the generate time from the time management system.
- Update UML Diagrams to show code changes.

Iteration 2 – February 19th

- Designed the elevator state machine diagram. See the State machine diagrams for the elevator state machine diagram.
- Implemented the elevator state machine as an Enum class, that had an abstract toString method as well as a next state method, both methods were override by each Enum respectively. This way the next state of the elevator was controlled based on a Direction Enum.
- Created a Direction Enum class used to control the flow of the Elevators State Machine.
- Created test cases for Elevators State Machine such that each possible flow was tested.
- Updated the elevator class to use the elevator state machine instead of just moving up and down floors, like it did in Iteration 1. This used a next state method that would set the current state to the next state based on the current state of the elevator. If the state was “moving up” the elevator would increase its current floor by 1. If the state of the elevator was “moving down” the elevator would decrease its current floor by 1.
- Created test cases to test the moving the elevator between floors using the next state method, which was built to use the elevators state.
- Update UML Diagrams to show code changes.

Iteration 3 – March 12th

- Created a Config class to load configuration variables from a properties file. This includes loading string values, integer values, as well as Boolean values. If a variable does not exist, the config class will cause the system to exit and log the message the variable that was trying to be read from the config file.
- Refactored some of the elevator subsystem functionality to the elevator class so that each elevator can run on its own thread, when expanding to multiple elevators.
- Added an elevator buffer class that communicated between elevator subsystem and each elevator. This was like the Producer/Consumer Model but supported two-way communication (this was split down the road to make debugging easier).
- Setup UDP utils for common UDP communication. Including serialization and deserialization used to encode and decode java objects passed between sockets.
- Modified elevator subsystem class to use UDP communication instead of passing messages between threads.
- Setup a VPN server so we could test our system on multiple computers. As this is outside of the course scope, I will not be going into too much detail about it.
- Updating relevant test cases for elevator and elevator subsystem. As well as add config test cases.
- Update UML Diagrams to show code changes.

Iteration 4 – March 26th

- Modified the way scheduler and elevator communicated such that the scheduler would only send request to the elevator subsystem when there is a next request to serve, once the scheduler sent the packet to elevator subsystem it would wait for the elevator subsystem to reply with the elevator subsystem.
- Fixed issues with the communication buffer, dividing it into two buffers that pass information in 1 direction.
- Fixed other elevator subsystem logic to handle passing errors between elevators. Additionally, once an elevator was removed the elevator count was decreasing such that the buffer would only wait for elevators that were alive.

Iteration 5 (Final Demo) - April 2nd

- Implemented a method to pass information from the elevators to the GUI. It was used to update the GUI's elevators.
- Modified existing process of handling temp errors modified such the elevator thread would sleep for x amount of time (the length of the temp error).
- Update UML Diagrams to show code changes.

Peter

Iteration 1 – February 5th

-Designed & implemented floorSubsystem to read and send elevator requests to scheduler
-Designed & implemented scheduledElevatorRequest to hold elevator request information to be sent to scheduler
-Designed & implemented TextFileReader to read the requests text file and convert them to an array list of scheduledElevatorRequests. This is used by floorSubsystem to get the requests from the text file
-Created JUnit tests for TextFileReader, scheduledElevatorRequests, and floorSubsystem to ensure proper collaboration of all the mentioned classes
-Integration testing
-UML class diagram

Iteration 2 – February 19th

-Designed & implemented logger class to get update messages of the processes running at all subsystems and the times at which these updates happened
-Updated floorSubsystem to have multiple constructor one where inputFile is one of the parameters in the constructor so we can switch between different input text files and the other with no inputFile in the parameter to use the default text file location.
-Updated floorSubsystem to use logger (that is initialized in the constructor) to print out update messages
-Updated scheduledElevatorRequest to have multiple constructors, one for DateTime timestamps for time and the other for milliseconds delay handling all requests in the input text file
-Updated scheduledElevatorRequest to calculate milliseconds delay from Date/Time used in the constructor or to calculate the timeStamp of a milliseconds delay initialized scheduledElevatorRequest
- Created JUnit tests for logger object to ensure that the actual print outs match with the expected
-Updated scheduledElevatorRequest testing to cover the new changes
-Integration testing
-UML class diagram updates

Iteration 3 – March 12th

-Redesigned floorSubsystem to use datagramPackets to communicate with scheduler using the dedicated ports to send the scheduledElevatorRequest and receive acknowledgements and to send messages to the server logger that handles all computer message updates using datagramPackets, and finally to use the Config file in the constructor to read the communication configurations.
-Transferred cmd UI from mainProgramRunner to floorSubsystem
-Updated cmd UI in floorsubsystem to handle error requests and print out guidelines to users
-Updated floorSubsystem to use the cmd UI to ask the user for the inputFile (text file that has the requests) or use the default text file rather than initializing it from the constructor

-Updated floorSubsystem tests to ignore cmd UI interactions and carry with the testing and not wait for user input
-Updated floorSubsystem tests to ensure that datagraPackets are sent to the Scheduler
-Integration testing
-UML class diagram updates
-FloorSubsystem UML sequence diagram

Iteration 4 – March 26th

-Input text file updated to have an additional a column for error type 0-No error, 1-transient error. Temporary for 10 seconds, 2-Permanent error.

-Updated textFileReader to read error column from the input text file

-Updated scheduledElevatorRequest to store, set, and get error type
-Updated cmd UI to handle new error type requests from the user and to print the new guidelines for using the interface

-Updated serverLogger to save the update messages from all subsystems on one logger text file based on the time at which the elevator simulator was run to be able to track all updates that happened after the system's shutdown.

-Integration testing
-UML class diagram updates

Iteration 5 (Final Demo) - April 2nd

-Updated scheduler to manually measure the time taken (in millisecond) to handle requests in the input file
-New inputFile to demonstrate the elevator simulation for a 22-floor building with 4 elevators and different error types with 20 requests
-Minor update to logger to highlight the printouts of time measurements
-Final Report
-Integration testing
-UML class diagram updates

Millan

Iteration 1 – February 5th

-Set up GitHub repository with base Eclipse project
-Supported teammates with Git guide to make pull requests, branches, commits, and to push & pull
-Setup scheduler to act similarly to the box in the consumer/producer model with wait loops
-Designed & implemented initial scheduler's floor request to elevator directions translation (flawed)
-Designed and implemented a scheduler system to delay the incorporation of floor requests based on the input time parameter with a feature flag to enable or instantaneously schedule the request
-Designed the communications object sent between the floor subsystem and scheduler
-Created JUnit tests for the scheduler subsystem that ensure that elevator instructions are received given a collection of floor requests
-Designed and implemented MainProgramRunner to start all subsystems simultaneously for easier integration testing
-Integration testing all subsystems working in unison
-Created first README file
-UML Class diagrams
-UML Sequence diagrams

Iteration 2 – February 19th

-Designed first version of scheduler state machine (unknown at time that it was flawed, edge cases)
-Updated scheduler algorithm to be consistent with state machine
-Designed command line UI state machine
-Designed and implemented command line UI in MainProgramRunner to make testing easier
-Added file selection window to choose which floor request input file to use to make testing easier
-Linked file selection system to pass selected/default file names to floor subsystem
-Integration testing all subsystems working in unison
-UML Class diagrams
-UML Sequence diagrams

Iteration 3 – March 12th

-Designed and implemented PacketReceiver abstract class for easier UDP communication thread implementations
-Designed and implemented packetReceiver extending UDP communication threads for the scheduler to receive direct communications with the floor and elevator subsystems
-Designed state machine for multiple elevator management system
-Designed and implemented the first version of multiple elevator support in the scheduler
-Redesigned scheduler's communication to elevator subsystem
-Set up Scheduler to be run as an individual so that subsystems can be run individually on multiple computers

-Refined MainProgramRunner to simulate running subsystems on different computers locally with UDP communications with one main method to start it
-Updated SchedulerTests
-Integration testing all subsystems working in unison
-UML Class diagrams
-UML Sequence diagrams

Iteration 4 – March 26th

-Created first version of GUI Mockup diagrams (Flawed, too text based, not visual enough)
-Created logical tables to model all possibilities of incoming elevator requests & remaining floors to visit
-Updated elevator specific scheduler state machine to cover all edge cases
-Redesigned and reimplemented elevator specific scheduler system with detailed testing
-Updated elevatorSpecificScheduler to support handling permanent errors
-Updated elevatorSpecificScheduler to support handling temporary errors (Flawed due to misinterpretation of project description)
-Updated elevatorSpecificSchedulerManager to accommodate changes to elevatorSpecificScheduler
-Integration testing all subsystems working in unison
-UML Class diagrams – Designed division into high-level, and subsystem specific
-UML Sequence diagrams

Iteration 5 (Final Demo) - April 2nd

-Updated GUI mockup diagrams to meet visualization requirements
-Integration testing all subsystems working in unison
-Revised implementation of elevatorSpecificScheduler
-Revised implementation of temporary and permanent error handling
-Updated scheduler subsystem to track how floor and elevator buttons would be represented
-Created Util.sendPacket_NoReply(DatagramPacket) static method for easy UDP communications to the GUI given a packet
-Designed and implemented GUI_PacketReceiver to communicate with the GUI subsystem
-Designed base GUI Subsystem to be run like all other subsystems
-Debugging GUI Subsystem
-Integration testing all subsystems working in unison
-UML Class diagrams
-Time Measurements of service, queueing, and transit times
-Recorded demo video, and edited for submission

Diagrams

AUTHORS' NOTE ABOUT DIAGRAMS:
Due to the huge size of these diagrams and the limitations of imported image pixel density in the team's shared document editor, diagram readability may be an issue. For more detailed analysis, please refer to the individual diagram PDFs in the documentation folder of the program's main folder as those diagrams are rendered as lossless vectors instead of lossy images

UML Class Diagrams

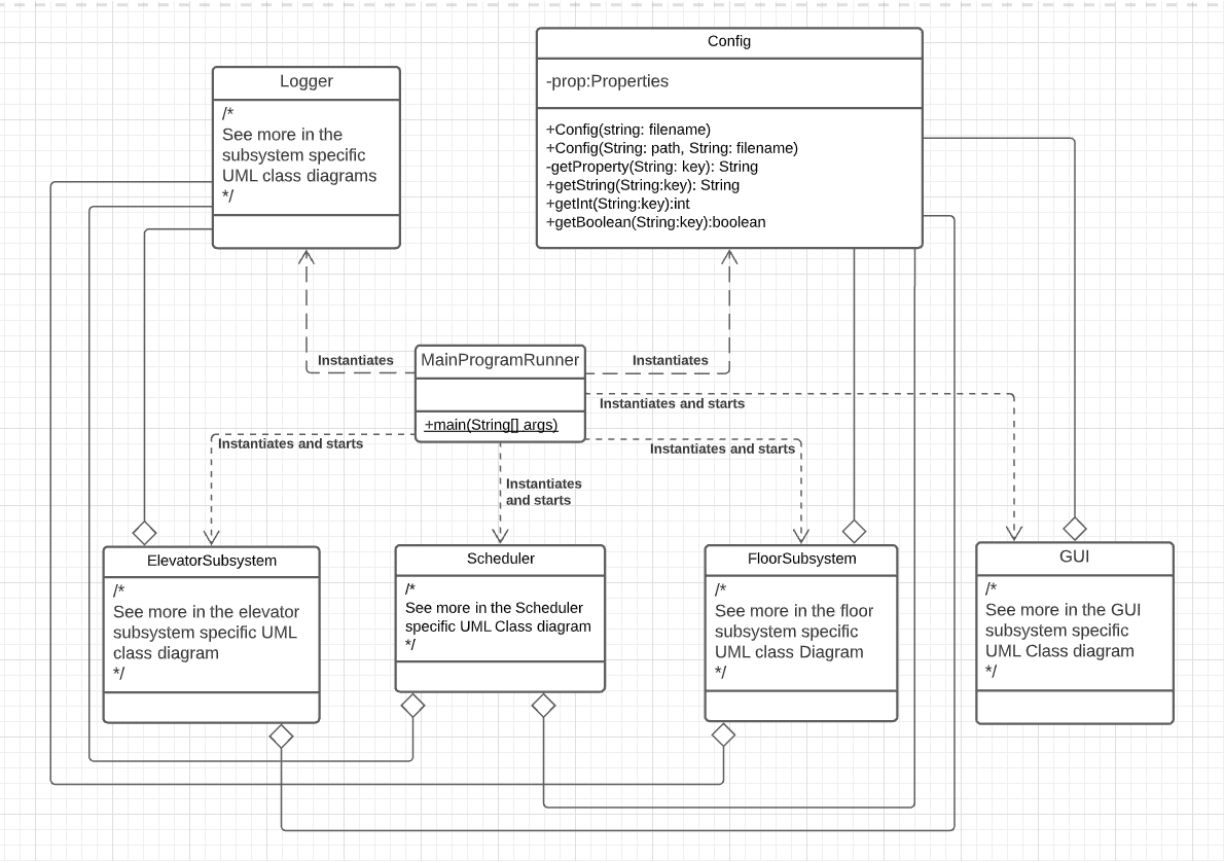

Figure 1: High level subsystem UML class diagram

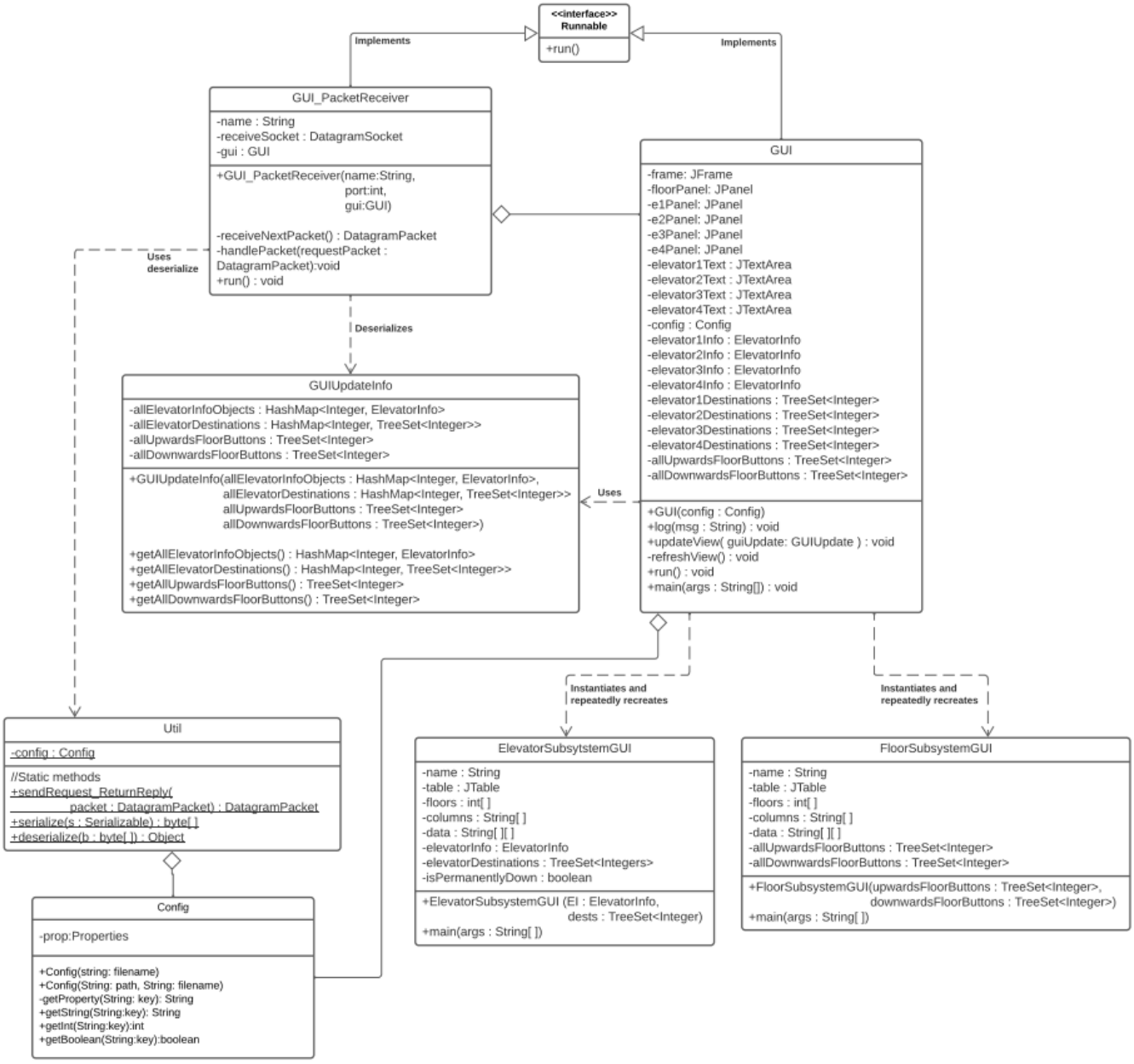

Figure 2: GUI Subsystem UML Class diagram

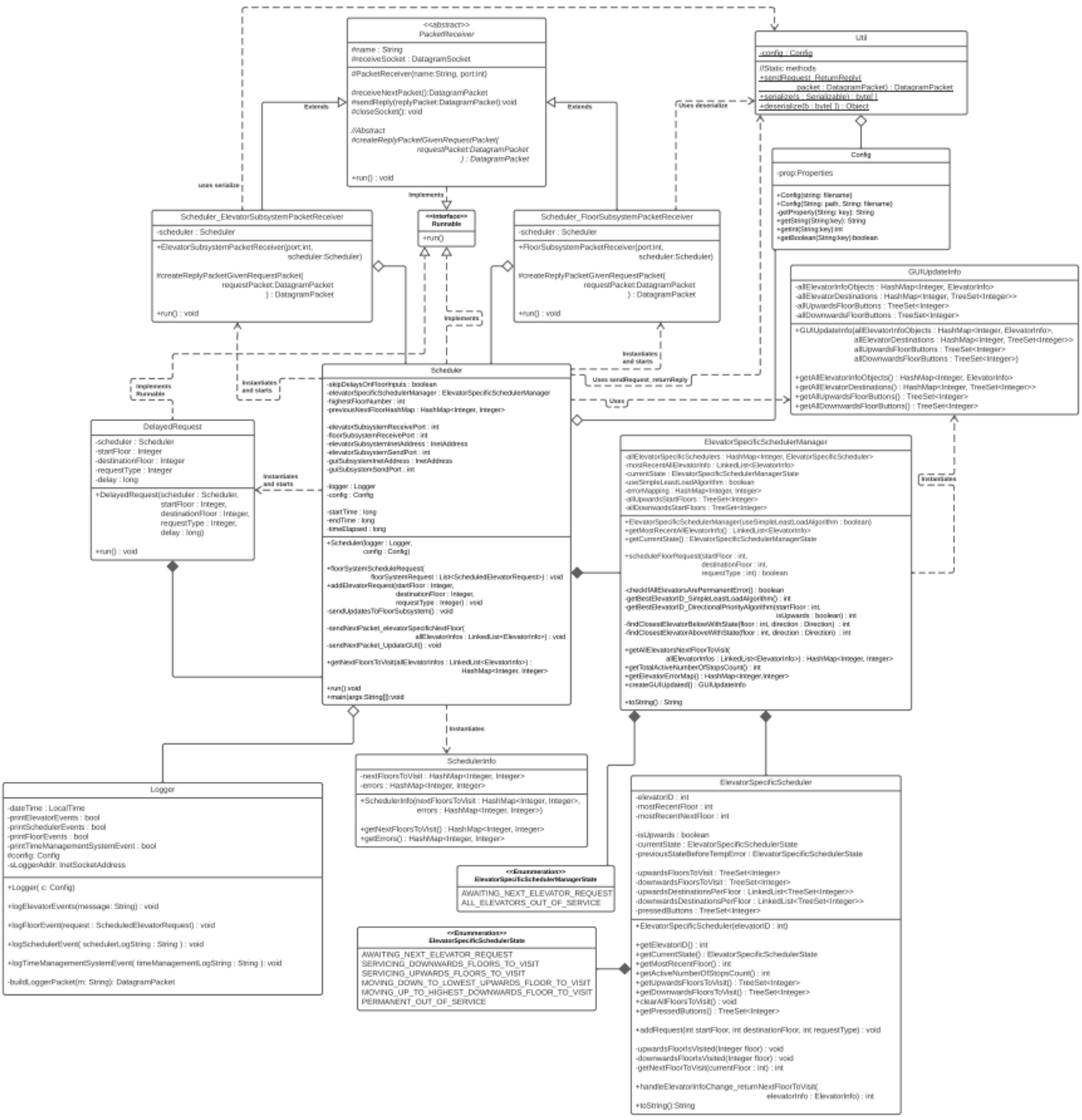

Figure 3: Scheduler subsystem UML Class Diagram

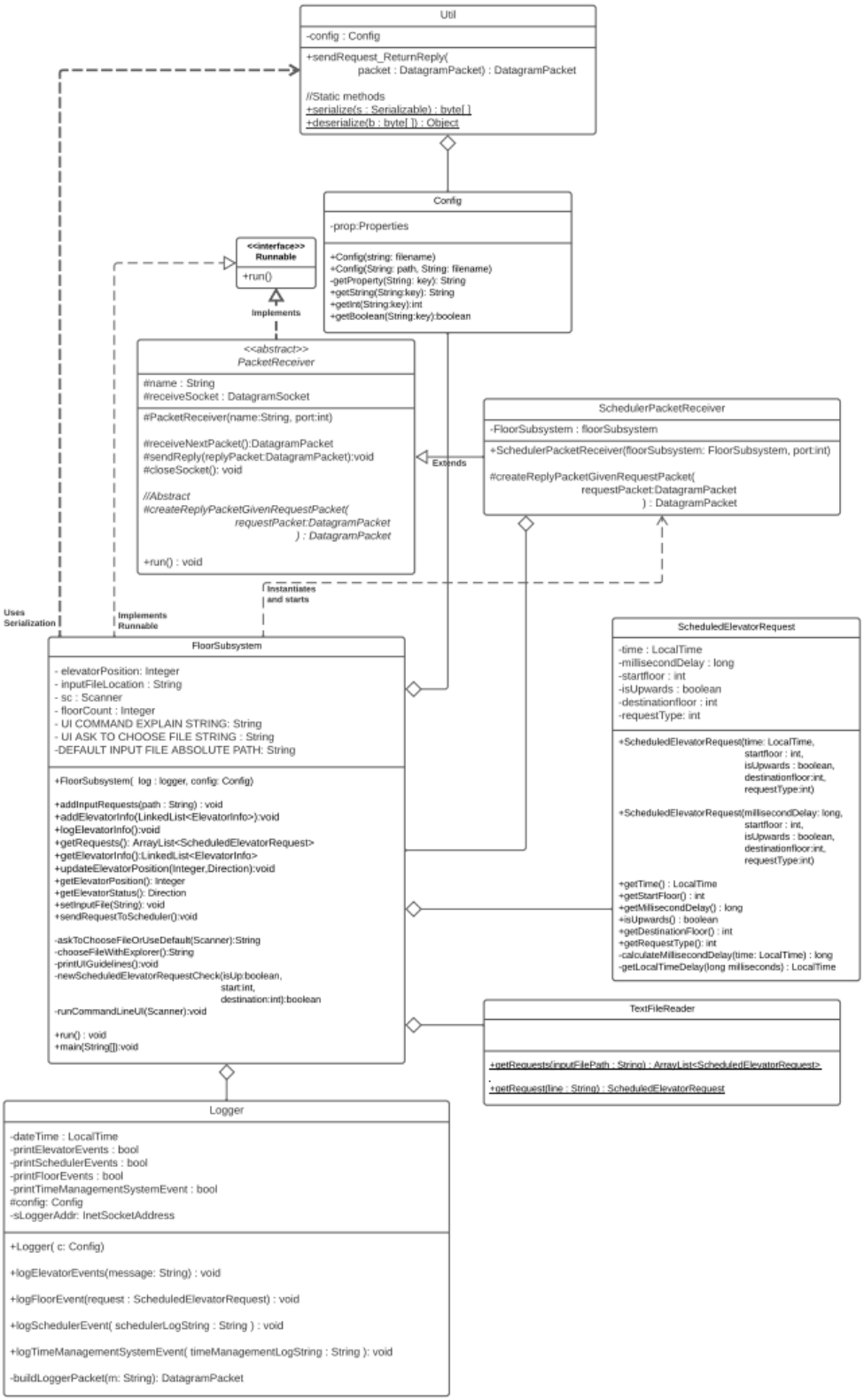

Figure 4: Floor Subsystem UML Class diagram

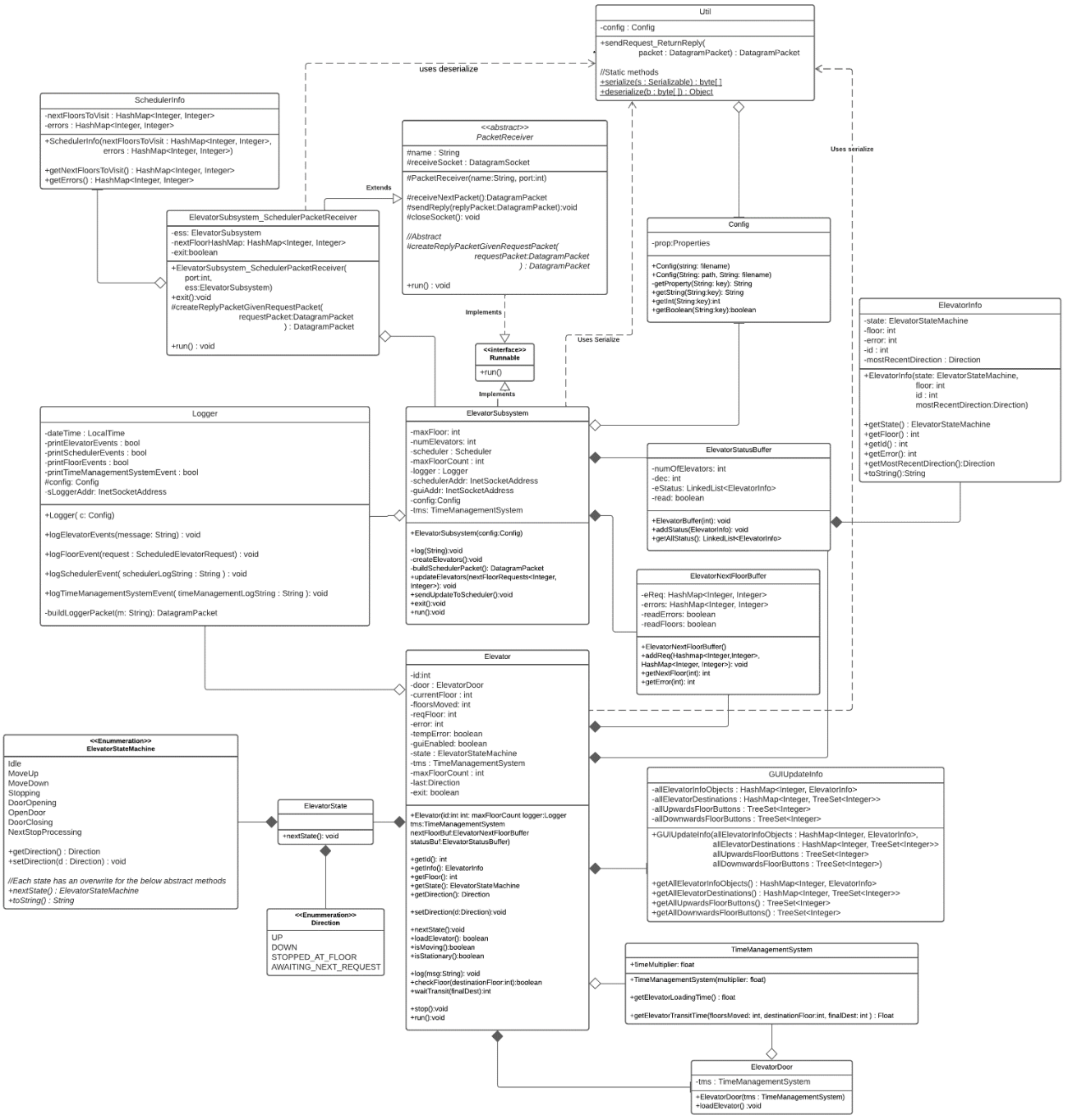

Figure 5: Elevator Subsystem UML Class Diagram

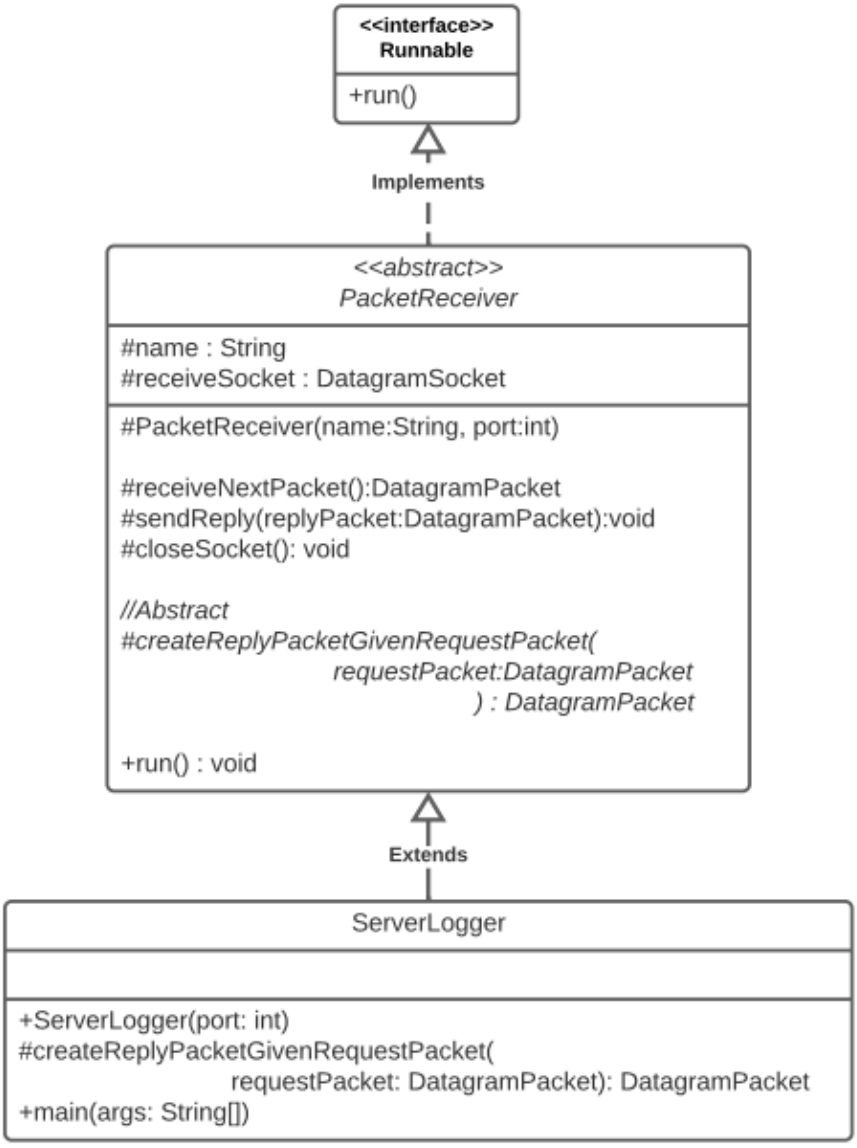

Figure 6: ServerLogger subsystem UML Class diagram

State Machine Diagrams

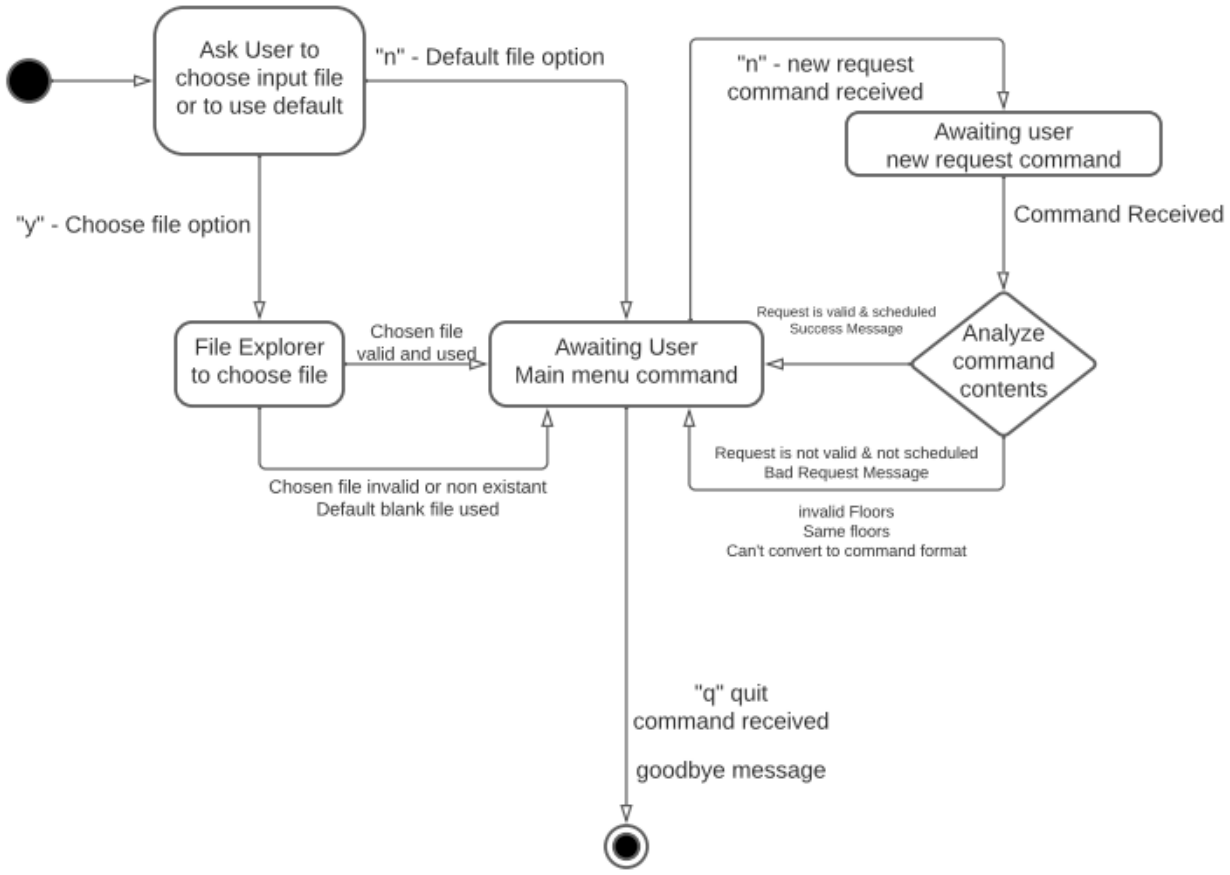

Figure 7: Floor subsystem command line user interface - state machine diagram

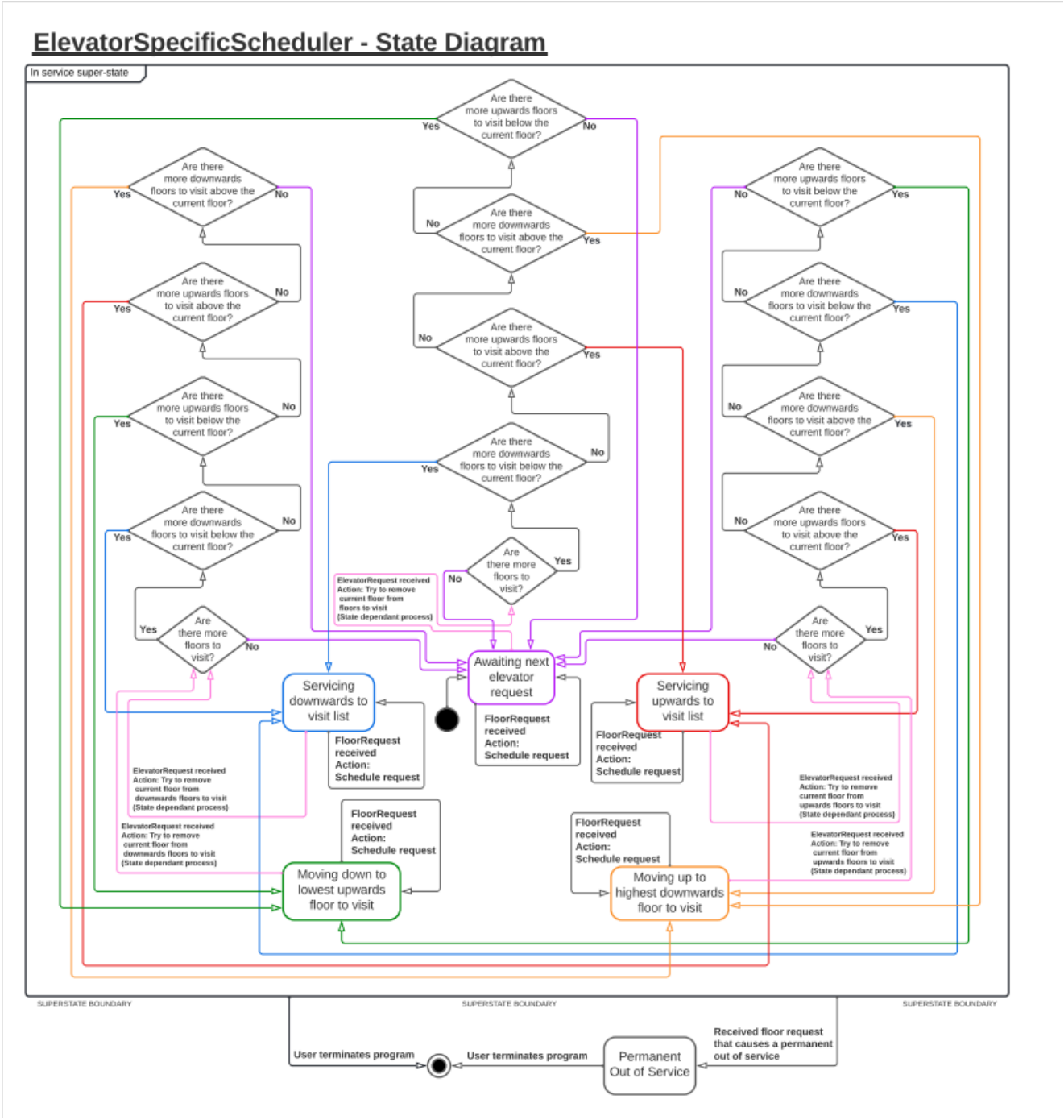

Figure 8: Elevator Specific Scheduler – State Machine Diagram
This state machine is based on the NextFloorSelection tables found in the scheduler documentation area

Elevator Specific Scheduler Manager - State Diagram

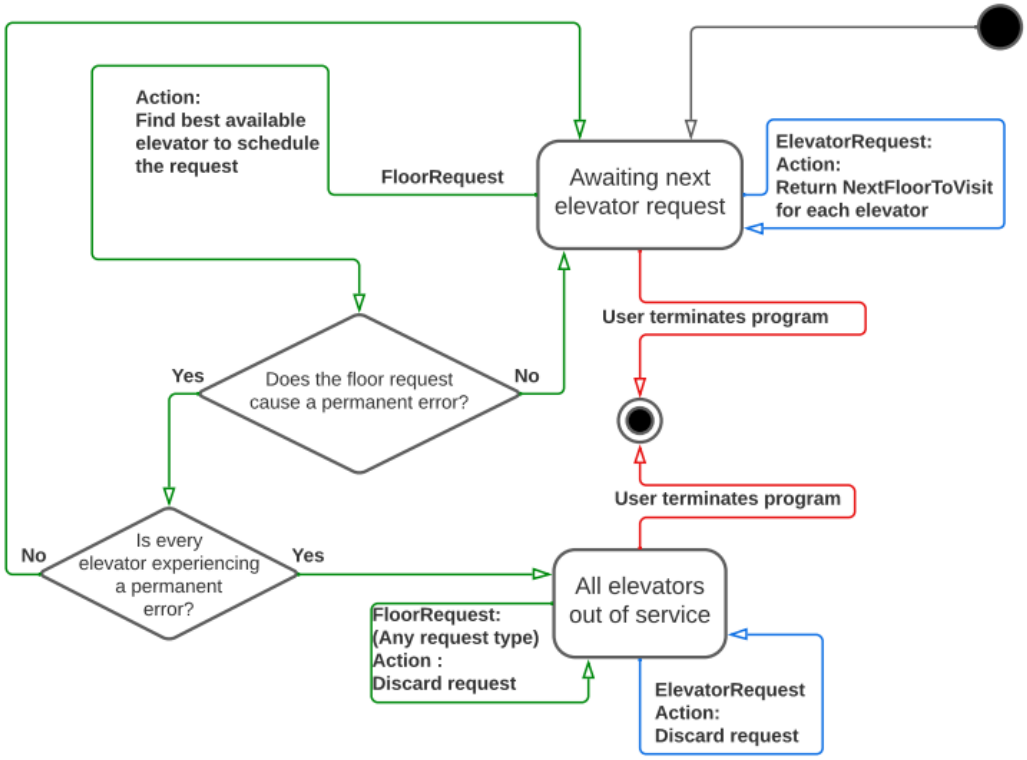

Figure 9:Elevator Specific Scheduler Manager – State Machine Diagram

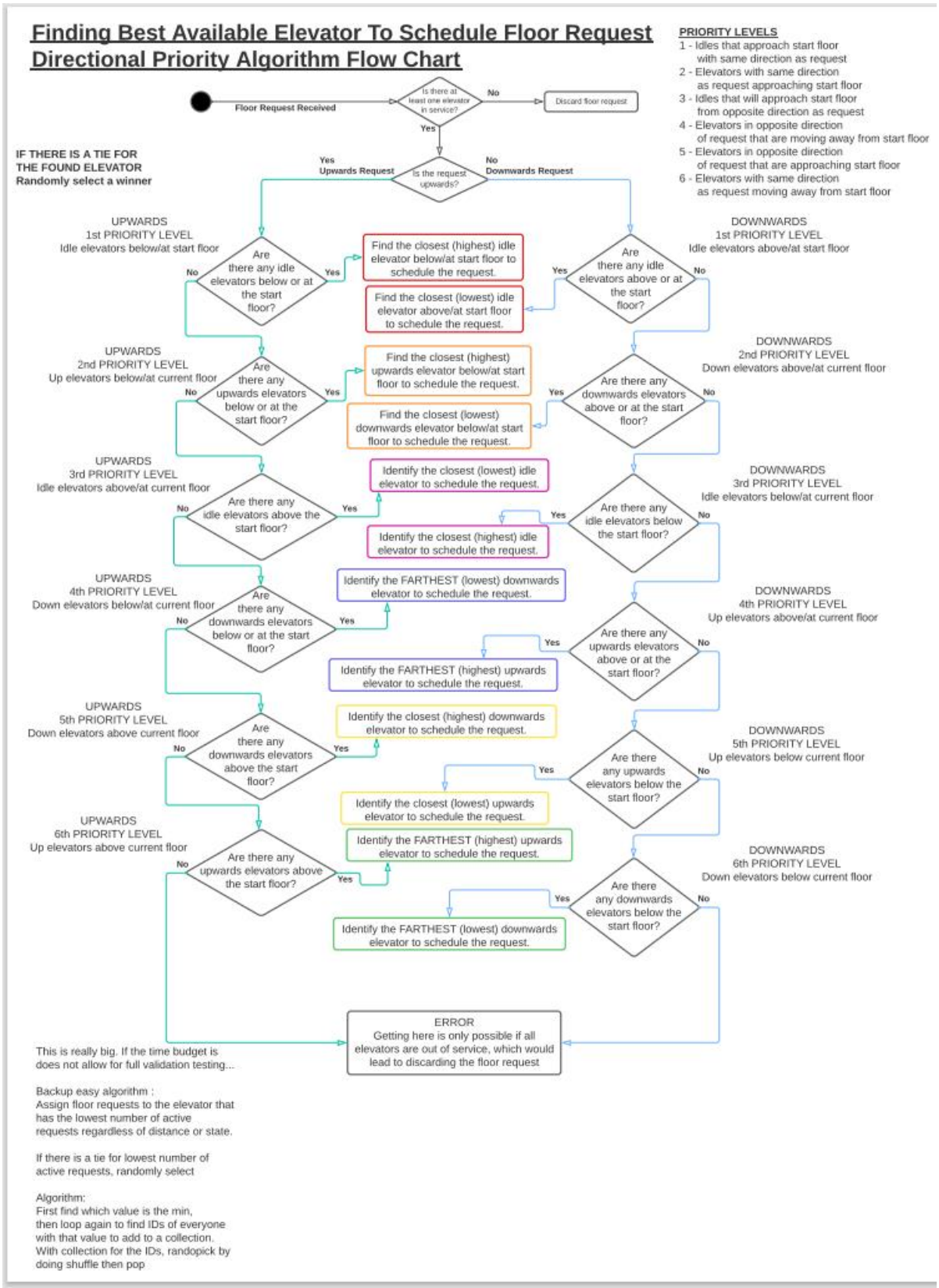

Figure 10: Floor request distribution algorithm "State" diagram and backup algorithm description

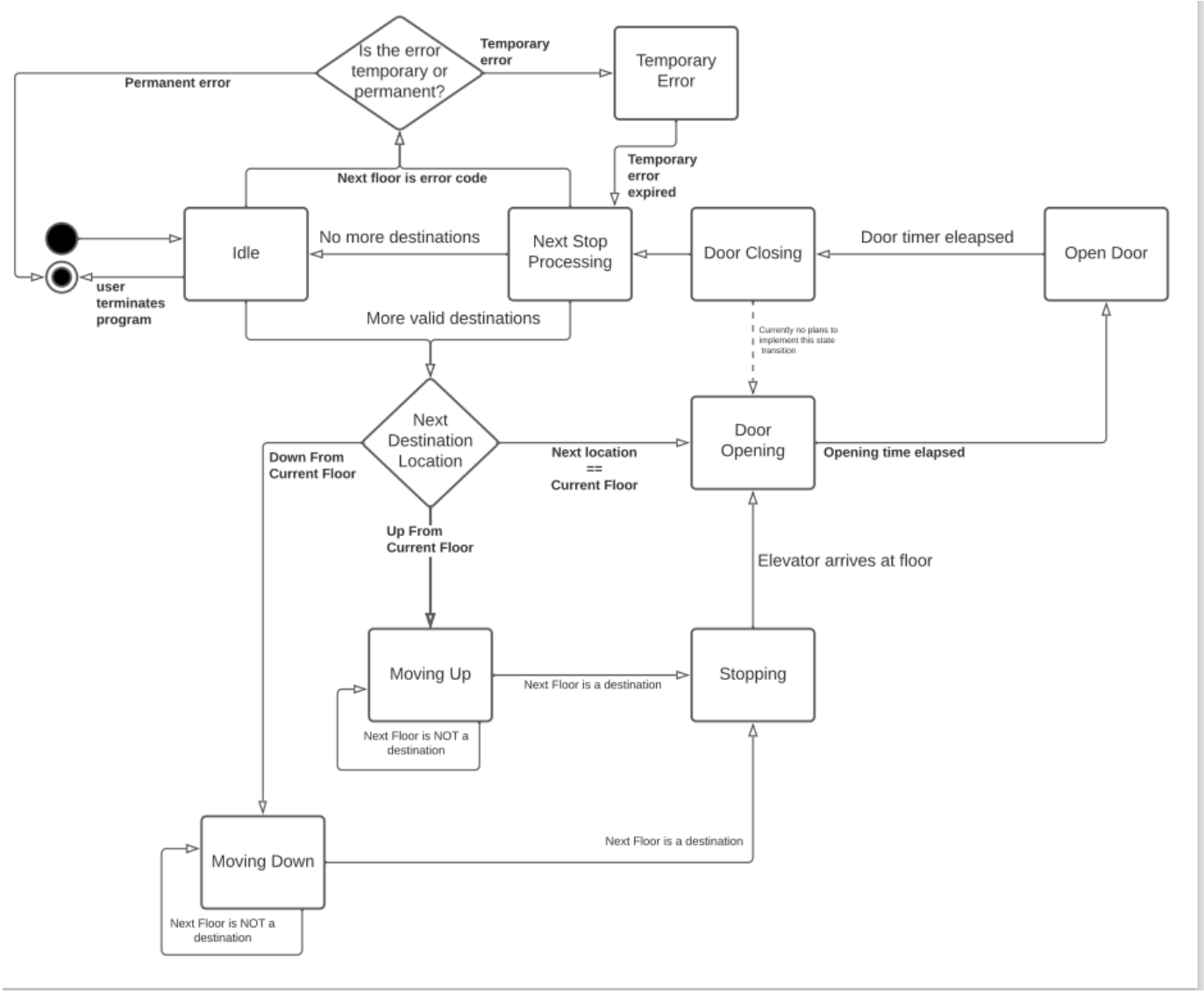

Figure 11: Elevator - State Machine Diagram

UML Sequence Diagrams

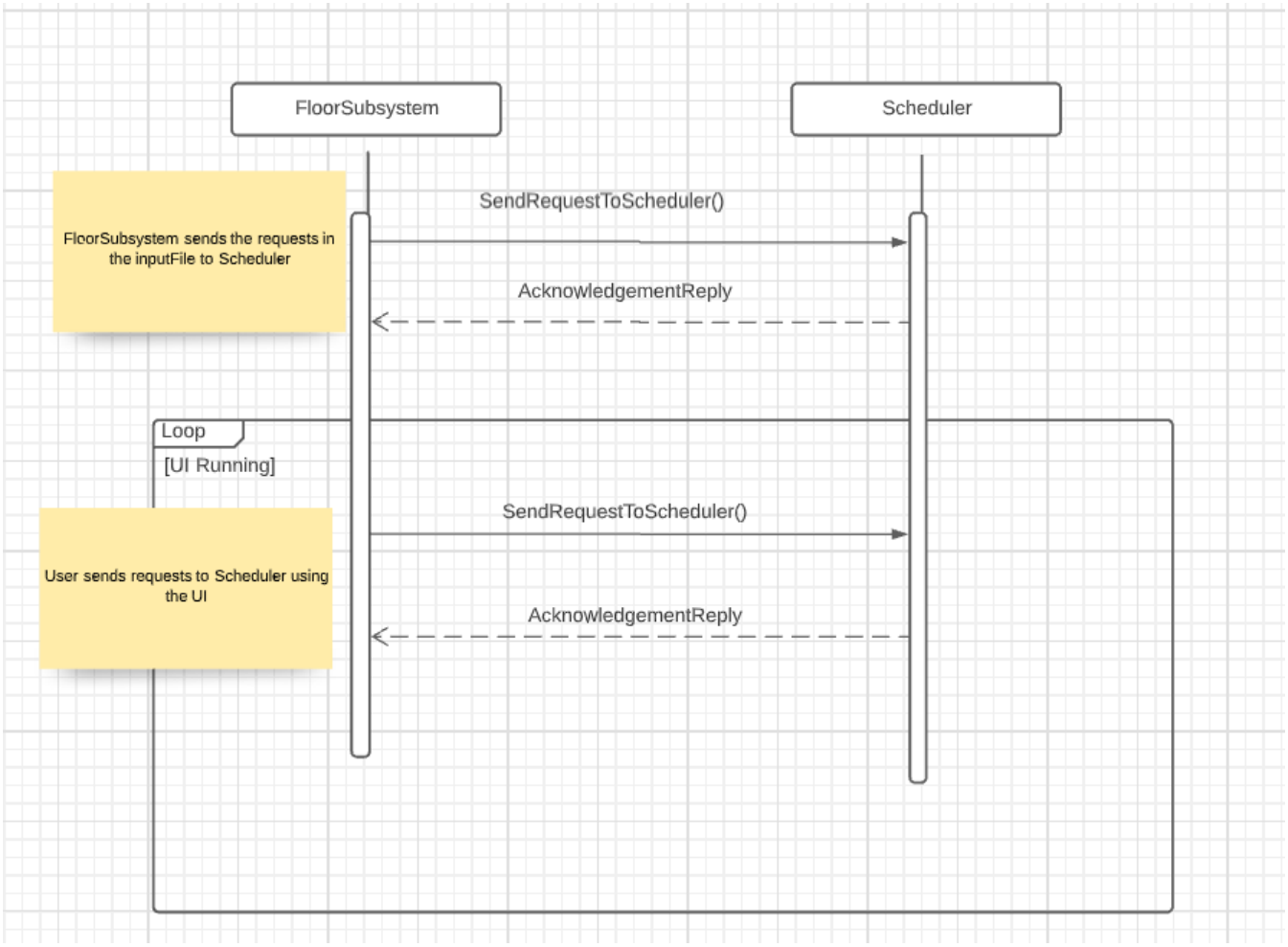

Figure 12: floorSubsystem Sequence Diagram

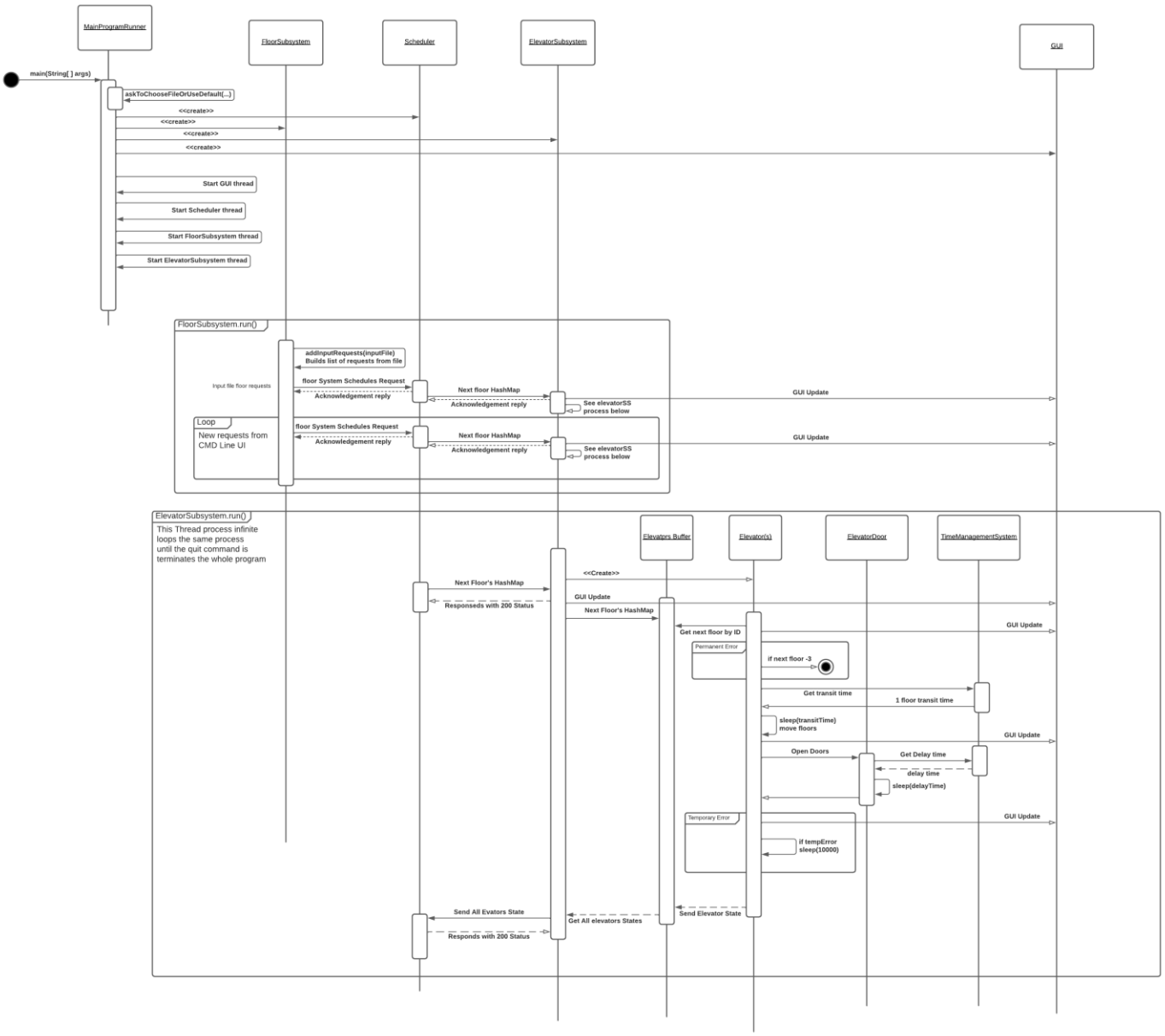

Figure 13: Main multi-subsystem Sequence Diagram

Timing Diagrams

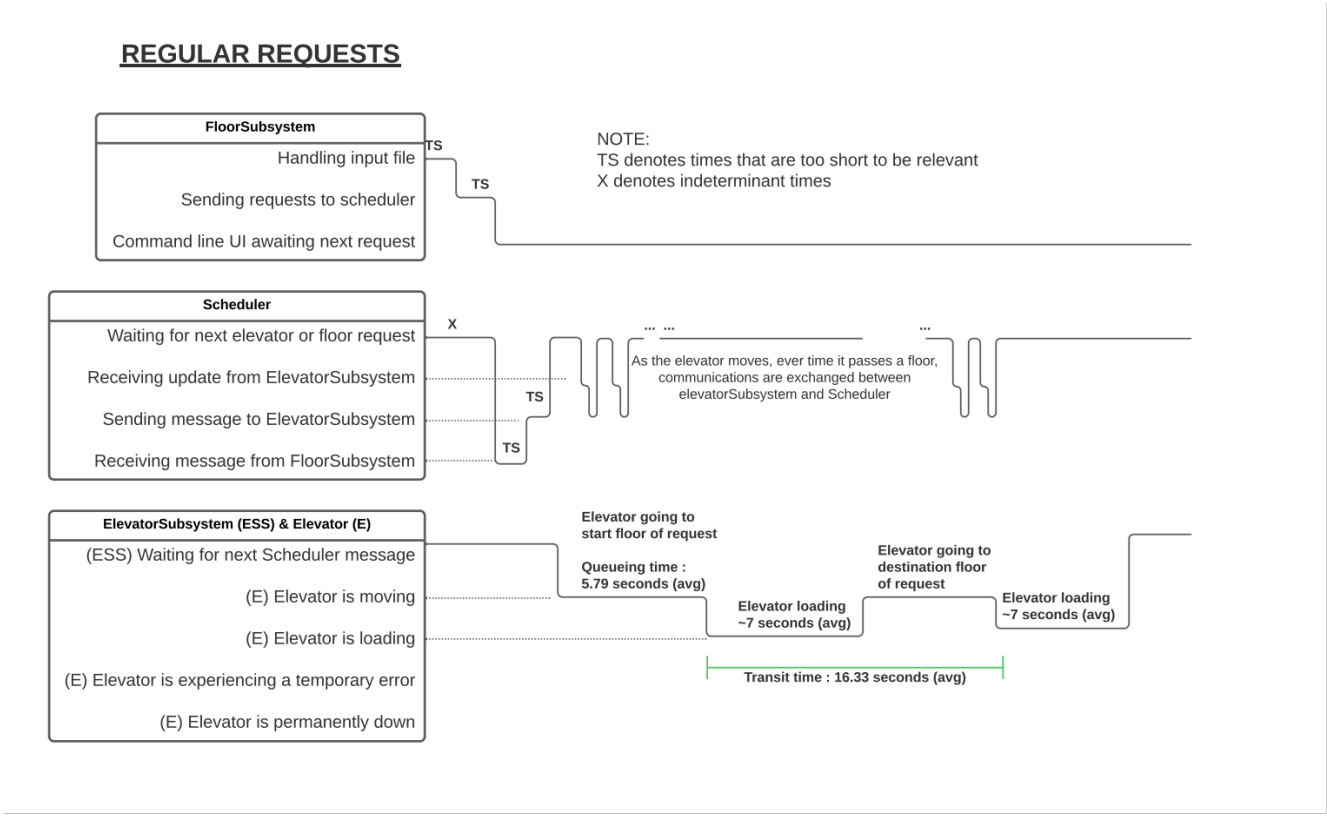

Figure 14: Regular Request Timing Diagram

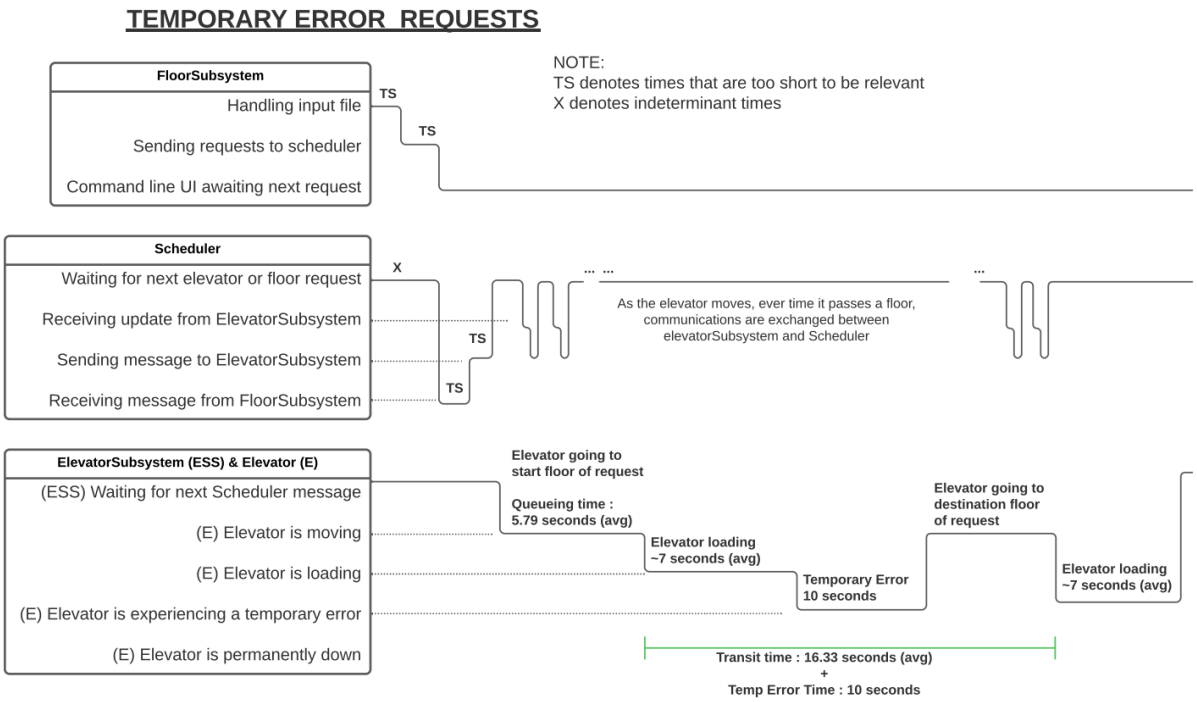

Figure : Temporary Error Request Timing Diagram

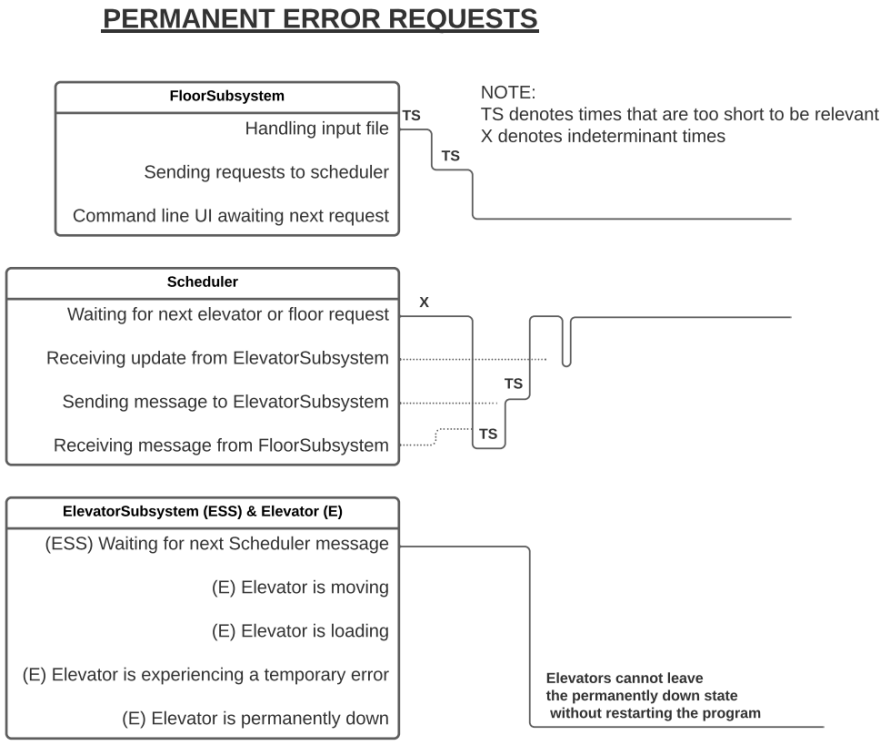

Figure : Permanent Error Request Timing Diagram

Time Generation Process

For the calculations of travel time from floor to floor, a couple of assumptions were made about the elevator's behaviour:

I) The acceleration of the elevator would be 0.50m/s^2
II) The elevator would reach maximum velocity within the one floors travel length (4 meters)

- Using these assumptions and the data provided, the maximum velocity was calculated to be 2.0m/s, this meant that the time it would take for the elevator to move from floor to floor would be **2.0s** at maximum velocity.
- It was also necessary to calculate the time it would take for the elevator to reach maximum velocity (which is also the time it would take to slow down to a stop). This value was calculated at **4.0s**.
- Comparing these times to the calculated times with times with the average times found from the data presented shows an > 80% resemblance which will be accounted for using a confidence interval.
- Calculated velocities with an assumption of acceleration = 1.5m/s^2 or 1.0m/s^2 proved to be too fast in comparison to the data and were therefore ruled out as possibilities.

Along with full distance travel, the data also gave floor to floor travel but included the time for doors opening and closing on each floor. The average speed here would be far slower than the continuous travel option as every floor involved acceleration on launch and slowed down to a stop on each floor. Identifying the amount of time it took on each floor for a door to open and close gave much more data about loading/offloading times.

- The mean of the onloading/offloading time and a 95% confidence interval were used to find a range of **9.53s ± 1.61s** used in the getElevatorLoadingTime() function.
- In contrast to the travel times, these times will likely not have to be revised as there is no need to account for acceleration or any other factors.

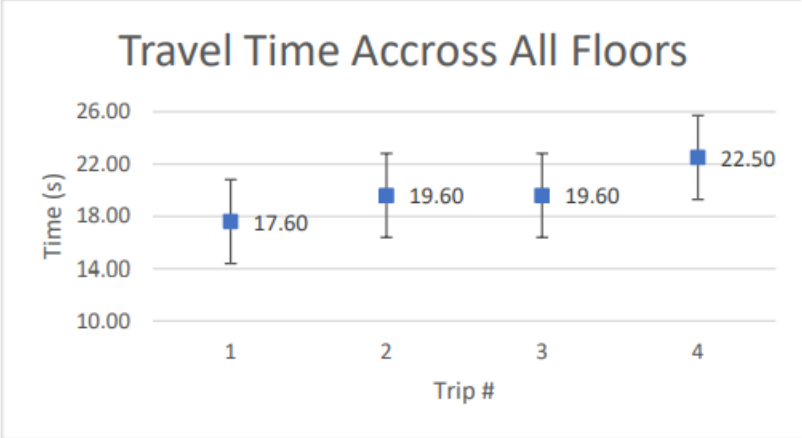

Figure 15: Travel time across all floors

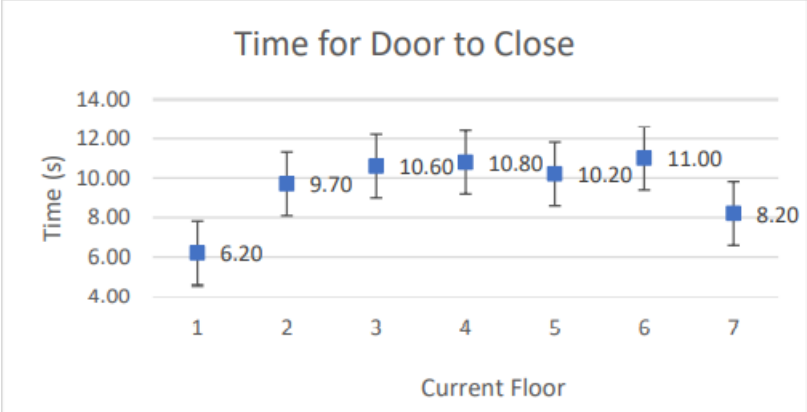

Figure 16: Time for door to close

Velocity/time model graphs of how movement would look when accounting for acceleration, deceleration, and max velocity:

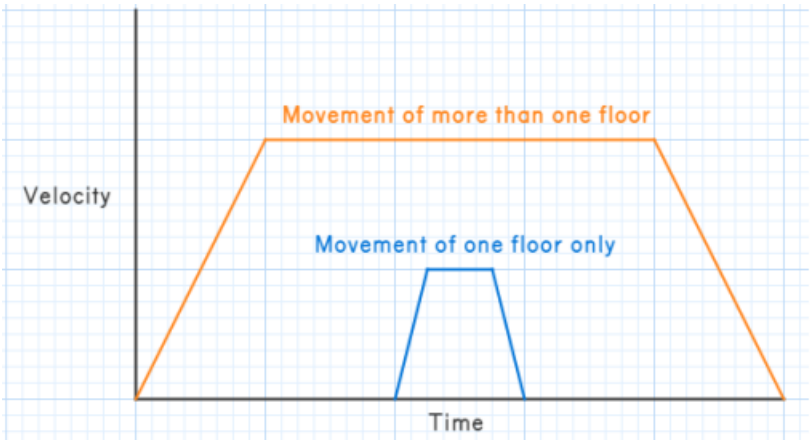

Figure 17: Velocity/time graph

This graph models what the elevators movement will generally look like, it will accelerate initially and plateau at a maximum velocity until it is 1 floor before its destination where it will decelerate to a stop.

- Since moving only one floor will not cause the elevator to reach peak velocity, it will plateau at a lower velocity for a much shorter time

Time Measurement Results

The current implementation of the program has detailed logging with timestamping so that temporal performance metrics can be determined. Note that we have implemented a system that allows us to accelerate time by applying a multiplier to all time values for easier testing, however, the following measurements were conducted with a time multiplier of 1. Note that during time measurements, the time formatting was changed to show milliseconds, but changed back before submission so that the logging would not be as cluttered.

Average service time

The average service time will be calculated by determining the time taken to handle all floor requests in the default input file, and then dividing that time by the number of requests. Service time represents the time between when the floor request is scheduled, and when the passenger arrives at their destination. The true value of this average will have a strong dependency on the number of floors in the system, the number of active elevators, the distance traveled by requests, the correlation between subsequent requests, the rate at which floor requests arrive, and the distribution of floors across the various requests. Given that there are so many variables associated with the true average of this value, and the limited time budget, only an estimate based on the demo input file will be used for the calculations, however, it would be feasible to implement a system that does more thorough statical tracking dynamically as the system is used, similar to a machine learning system training itself.

Measurements

Number of requests: 20

Time to complete all of them: 257779 ms = 258 seconds = 4m18s

Average service Time: 12889 ms = 13 seconds

Average queueing time

The average queuing time will be determined by analyzing the console log to determine when requests are received by the scheduler, and when an elevator arrives at the start floor of the request. This time measurement will depend on the density of floor requests, and if/how other requests are currently being serviced. These factors will not be considered in measurement for simplicity, and because these measurements will only be used to provide estimates for the timing diagrams

Measurements

Number of requests: 20

Queueing time measurements: [9.86230, 6.07370, 4.02570, 7.32800, 2.26430, 0.92823, 2.98910, 4.58230, 2.42610, 6.92480, 12.1950, 1.25340, 9.17590, 8.59010, 3.71990, 4.13970, 4.03250, 12.2390, 10.1420, 2.92500] seconds

Average service Time: 5.79 seconds

Average transit time

The average transit time will be determined by analyzing the console log to determine when passengers are picked up and when passengers are delivered to their final destination. This time measurement will depend on if other requests have paths that overlap with the currently measured one, however, this will not be considered in the measurement for simplicity, and because these measurements will only be used as estimates for the timing diagrams.

Measurements

Number of requests: 20

Transit time measurements: [19.1940, 11.8260, 12.8200, 26.9300, 24.3150, 13.2870, 14.4230, 13.7640, 11.2904, 16.3471, 16.6060, 11.9770, 25.7150, 15.5763, 16.7880, 19.6870, 9.65100, 13.9390, 14.1930, 18.3038]

Average Transit Time: 16.33 seconds

Note about the relationship between measurements

The reason the individually measured transit times added with the individually measured do not add together to become the average service time is because the average service time has 4 elevators handling around 5/20 requests each, and the time measurement for average service time is based on multiple elevators with its associated temporal acceleration. The time measurements for the average transit time and average queueing time are with respect to a single elevator.

Configuration & Setup

To make life simpler our group created a Config class that would load a properties file into memory. This config class is passed to the following classes logger, scheduler, floor subsystem, elevator subsystem, and GUI. If any of the following properties are missing, then the system will exit resulting in an error message being logged to user explain the variable that is missing. The table below shows all the required properties as well as the type it expects.

Important Config Properties

Properties	Description
time.multiplier (int)	
time.log (boolean)	If true will log time management related messages
scheduler.address (string)	The ip address of the scheduler
Scheduler.elevatorReceiverPort (int)	Port the scheduler will receive messages from elevator subsystem
scheduler.floorReceiverPort (int)	Port the scheduler will receive messages from floor subsystem
scheduler.log (boolean)	If true scheduler events will be logged
scheduler.skipDelaysOnFloorInputs (int)	If 1, then the delay for received floor requests will be skipped. If 0, then delayed floor requests will be handled normally
floor.address (string)	The address of the floor subsystem
floor.schedulerReceivePort (int)	The port the floor system will receive messages from the scheduler
floor.highestFloorNumber (int)	Number of floors
floor.log (boolean)	If true the floor subsystem events will be logged
elevator.total.number (int)	Total number of elevators
elevator.log (boolean)	If true the elevator subsystem (and elevators) will be
elevator.gui (boolean)	If true the elevator will send updates to the gui
elevator.address (string)	The ip address the elevator is running on
elevator.port (int)	The port the elevator subsystem receives messages on
logger.address (string)	The ip address of the logger
logger.port (int)	The port the logger receives messages on
gui.address (string)	The ip address of the GUI
Gui.port (int)	The port the GUI receives messages on
udp.buffer.size (int)	The buffer size for messages datagram packets

Running

To run on multiple computers, it may be required to disable the windows firewall so packets can be passed from ports of 1 machine to another, this may also be required for running locally.

To run the system locally the best option is to use the MainProgramRunner.java file located in the app package. This file will use a local config file (note you may need to change some of the port variables depending on the ports you have available, additionally you can change parameters as needed). The MainProgramRunner.java will create 5 instances. The instances are as follows Logger, Scheduler, FloorSubsystem, ElevatorSubsystem, and GUI.

To run on system on multiple computers each of the 5 instances mention above, have a main method that can be used to run that instance on a different machine. This will require environment specific updates to the configuration file passed into the constructor for each of the 5 instances accordingly (or change the local.properties configuration file).

GUI Colour Legend: This legend identifies what each colour present on an elevator or floor button represents.

FloorSubsystemGUI – Displays all buttons in building	
Colour	**Indicating**
	Button Unpressed
	Button Pressed

ElevatorSubsystemGUI– Displays all buttons in elevator and current elevator position	
Colour	**Indicating**
On Right Hand Side (Elevator Buttons)	
	Button Unpressed
	Button Pressed
On Left Hand Side (Elevator Position)	
	Elevator Not At Floor
	Idle at this floor
	Doors Opening at this Floor
	Doors Open at this Floor
	Doors Closing at this Floor
	Temp Error at this Floor (Doors stuck)
	Perma. Error at this Floor (Elevator Shut Down)
	Elevator Moving

Testing

Our group used JUnit as our testing framework. Each member of the group was responsible for testing their own code. A list of test files that were included can be seen below. A common feature throughout the testing is implementing mock systems that simulate responses for packet communication. This is needed so that starting and running subsystems individually would not get stuck waiting for a UDP packet reply. It is worth pointing out that the ElevatorSubsystemTest.java is designed to test end to end behavior, and within the test there is a FakeScheduler class that is used to send fake requests to the elevator and get the elevators response. This end to end testing leads to some issues when running the elevator subsystem tests, as the tests do pass when run individually, but sometimes the combined execution of all tests will cause the test to freeze.

JUnit Test Files

- ConfigTests.java
- ElevatorSubsystemTest.java
- ElevatorTests.java
- ElevatorStateMachineTests.java
- FloorSubsystemTests.java
- LoggerTest.java
- SchedulerElevatorRequestTest.java
- ElevatorSpecificSchedulerManagerTests.java
- ElevatorSpecificSchedulerTests.java
- SchedulerTests.java
- TimeManagementTests.java

Reflections

Abdelrahim

Looking back at the development process for this project, I was able to identify areas that my group and I did well, along with areas that could have been approached differently to provide a better result or more efficient production speeds.

One approach that I feel was extremely helpful towards the completion of this project was the organizational tools and methods we used prior to working on any iteration. The typical organizational meeting would consist of a breakdown of the iteration requirements, then identifying what code needs to be written and how it will communicate with other classes/portions of the program, and then finally a person-by-person division of work to close off. This structured approach allowed us to spend less time planning and more time in the development process, which was crucial as this project required a lot for each deliverable.

Another part of our process that I felt was crucial to the success of our group was the area-focused approach we took towards each portion of the project. Although assistance in every area of the code was part of everyone's responsibilities (testing, debugging etc.), each group member would usually work on the specific area of the code which they created and focused on. Having this distinction of code areas not only allowed us to speed up the planning process, made creating new sections of code easier and allowed for the project to be maintained throughout, but it also made communication between group members smoothers. Since this program requires a lot of inter-process communication, many of our classes would communicate and knowing who coded what made integration flow smoothly.

In terms of changes I would have made to the project approach knowing what I now know, the first approach I would alter would be the framing of the project starting at Iteration 0. Approaching the project as a JavaFX project from the start would have made the final stages of this project much less problematic. Knowing what information would have been needed to create the GUI from the start of the project would have allowed me to create specific variables and methods designed to be used by the GUI as the project progressed, as opposed to having to do all of that in a much more complex version of the program near completion.

I would also have liked to research more options for displaying the information in the GUI. Using JTables as a form of display proved to be problematic and took longer than it should have as updating them as the program progressed showed a few flaws that needed to be resolved. If I were to have more time, I would have also allowed for the dynamic creation of elevator displays as the program can currently run with a different number of elevators, but the GUI will only show 4 elevators.

The final area of improvement would be the time measurements provided which were used to calculate velocity and acceleration. It would have been much more accurate and simpler if more measurements were taken for the elevator, specifying maximum velocity motion and time to speed up or slow down as it comes to a stop. However, with the assumptions made during the analysis process, the results are usable and do accurately depict elevator motion.

Despite the areas of improvement that have been mentioned, the final submission of this project is of high quality and delivers an accurate depiction of an elevator system. The project fulfils all

the documented requirements in a well-organized and understandable manner, producing a simple and functional elevator system.

Ben

For this project, one of the features I liked the most was the way the Config class read the configuration file to be used in our system. I found having a class that loaded all the properties made it easy to change the configuration for our system without having to go through the code to update each variable that needs changes. The design of having get methods for several types of variable types made life easier as well. If I could have changed one thing about the config class, I would have adjusted the config file name to be a flag that was passed as an argument, therefore eliminating the need to update the config file name in 5 separate locations.

In terms of the communication between the elevator and elevator subsystem using the producer/consumer model I think that this could have been better implemented by removing the elevator subsystem and have the scheduler communicating with each elevator individually using UDP. Ideally, I think that each elevator would run on its own thread and accept packets from the scheduler, where the scheduler would send updates for each elevator assigned to that elevator next floor as well as any errors. I think this would have solved some issues we were having with threads locking during development as well as during testing.

Peter

The development of the Elevator Control System and Simulator program was based on detailed plannings and tasks delegation that led to a high-quality program. However, due to the limited time budget, some tasks and classes could have been designed and implemented better. The change of requirements through the different iterations of this project has caused major changes in the initial structure and design. For example, switching all communications between subsystems to use the Internet protocol (Datagram Packets and Sockets) in iteration 3 rather than using instances of subsystems in communication in iterations 1&2. The floor subsystem base functions that included reading the input requests from the text file using the textFileReader and ScheduledElevatorRequest to store the request information has followed a good design in the implementation that made the new changes that came into the project in the latest iterations easier to implement, such as adding the new error configurations in the elevator requests. Nevertheless, adding the command-line user interface (cmd UI) to floorSubsystem class was not the best solution as it caused many errors in the testing phase and has made the floorSubsystem not just responsible for reading, and sending elevator requests to Scheduler but also handling an entire UI which was out of its scope in the initial design. The cmd UI should have had its own class that handles all user interactions just like the graphical user interface (GUI). A User Interface folder or subsystem that holds both the cmd UI and GUI would have made the design a bit more understandable and uncomplicated. Additionally, the delaying of floor requests might have made more sense on the floor side than the scheduler side. This would better reflect how real-life elevator systems work and would make the UDP comms thread between floor and scheduler more optimally utilized and would also allow the packet buffer size to be reduced.

Overall, some aspects of the design could have been done a little better, but the current implementation is still of high quality and fulfills all the requirements of the project to simulate a simple and understandable real-time elevator system.

Millan

Throughout the development of this project, there are many areas which I thought were done well, and many areas that would be redone given a larger time budget. The first thing that I would have done differently would be to revise the high-level design methodology to start with compiling a collection of all requirements for the final deliverable and then associate each requirement to an iteration number. The approach that we took instead was to look at the requirements of the current iteration to inspire design which led to design choices that had to be nearly completely redone on later iterations. I believe that the time taken to redo functionalities that changed between iterations with a start-to-finish design methodology was more costly to the overall time budget than investing more time into design with a finish-to-start design methodology would be. This is most relevant to the transition from a single elevator system to a multiple elevator system as the design used in iterations 1&2 was eventually found to be so incompatible with multi-elevator operation that it was easier to just redo it all completely. Related to this, I also overestimated the requirements for iteration 1 and started to implement a scheduling system before even having a state machine diagram to guide its implementation. This was due to me not thinking that the main requirement to make a communications link between the subsystems was way too simple for this course and I didn't want to have a system that had communication links with bad communication data. This became a problem as this felt like this task overwhelmed me a little bit on the first iteration, and this work was done in vain as most of it would need to be redone. This issue could have been avoided by reaching out to the TAs more before the first iteration to get a clearer understanding of the requirements.

The next area of discussion is the UDP communication systems that were used. I believe that making an abstract PacketReceiver class was a good design decision because there were many areas which needed to have parallel threads that received UDP packets, and given that they would have nearly identical implementations, it is better to have all their shared functionality in a shared superclass so there are less areas in which bugs may occur, and fixes would only need to be done in one location. However, one issue with this area is that I designed this immediately after doing assignment 3 which focused on remote procedure calls where packet replies are ***always*** expected, while the replies are only sent under certain conditions. I spent a lot of time on this assignment and given that this assignment was fresh in my mind when I designed the PacketReceiver class, I implemented it in the project such that all UDP communications would require replies to proceed, and replies would only be sent under certain conditions. Often, the replies would not contain any useful information beyond a 200 OK message to signify that the packet was received and handled properly, and the requirement for replies lead to a lot of problems during integration testing with threads getting stuck waiting for replies. These bugs were very difficult to resolve as the learning curve for effective usage of multithreaded debug tools is steep & not covered in the lectures. These issues could have been more easily dealt with if communications were done with one-way asynchronous packet communication, and when it came time to set up GUI communications, I set it up so that only one-way asynchronous communications were expected, leading to much easier communications. The lesson learned from this UDP communications development experience is that design decisions should be made independently of other recently done work, as I feel that my familiarity with the assignment 3 communication system had too much of an influence on the design decisions made in this area of the project.

Another area of discussion is the process in which ElevatorSpecificSchedulerManager has different algorithm options for how to distribute floor requests for elevators. I think having interchangeable algorithms was a good design choice as this would make it easier to audition different

ideas, however I wish that there was a longer time budget to allow for more experimentation with different algorithms. I think I got too comfortable with the simple algorithm and decided not to invest too much time into alternate options, but I'm proud of the design that left that option available, even if unused.

The next area of discussion is the managerial organization of the team. During the first meeting discussing what was needed for the first iteration and how to divide the labor, we decided to assign roles based on each of the floor, scheduler, elevator, and time management subsystems. At the time, this felt like an easy way to divide labor with everyone working in mostly separate jurisdictions which significantly reduced the likelihood of merge conflicts or overlapping work. This worked out for the first few iterations where the division of labor in the different jurisdictions was mostly even, however, iterations 3, 4, &5 had more significant changes to the scheduler & elevator subsystems and given that our division of labor was still dependent on each team member's original jurisdiction, this led to an imbalanced distribution of responsibilities. My jurisdiction was the scheduler subsystem, and looking back, I feel that I didn't initially see the parallelizability of internal operations such as UDP communications, assigning floor requests to elevators, and elevator specific movement control, which led to me addressing all those functionality areas while other team members occasionally had idle hands due to everything on their own to-do list being done. Also, having longer to-do lists can increase productive efficiency as it will reduce the impact of Parkinson's law, aka, the tendency for tasks to take longer to complete if the deadline is farther away.

Another managerial operation that I wish I had done better was the instructions for the GUI subsystem design and to-do list. Initially, it was thought that using a table would be an easy way to represent the movement of the elevators, and with this, I recommended that Java's built in JTable component would be a good system to implement this. When I made this recommendation, I had zero experience with using JTable components, and did not realize that dynamically updating JTable cells would be so problematic. I wish I had instead made a list of GUI component requirements and then as part of the GUI assignment, make a task to determine which GUI components should be used and which would be avoided. This would have disqualified the usage of JTables early on and would not have led to the person with the GUI assignment spending so much time with the problematic component and would have prevented fighting its' inconsistencies until 4AM on the day of the presentation. In retrospect, a GUI implementation with **MANY** simple components would have been much better than our current implementation with a handful complex components, and this may have also increased the parallelizability of GUI development.

The final managerial point of discussion is that I wish I had put more pressure on the utilization of Git during the first two iterations. It's a shame that version control systems were not covered in this course, or in any prerequisites of this course, as collaborating on code is impractical without it and most professional software development roles depend on it. Luckily, half of the group was already fluent in using Git before starting this course, but this was only by coincidence from experience gained in co-op work terms. For the other half of the group that was not familiar with Git, I wish I applied more pressure on making their first contributions through git early on to avoid the problems with contributing code as the deadline approached. When I initially made the repository 1 day after the project group was made, I made a general request to the group to make a pull request to prove that everyone will be able to contribute, and I also mentioned that I'd be able to help anyone encountering issues with it. I only started to apply pressure on this topic 3 days before the deadline, and I wish I started to apply pressure

once per day after I made the repository as the proper usage of Git was mission critical to proceeding with the project.

Overall, I think that the project was sufficiently well done with respect to the requirements of the project and course, however, many improvements could have been made with a longer time budget, and without the distractions of other courses & searching for co-op work terms to cover tuition costs. I am proud of what myself and the team accomplished, and this project taught me many valuable lessons in software concurrency, UDP communication, software development team management, and perseverance.