

Project Report
on
Elevator System

CISC 675

Fall 2017

Submitted by:

Dixit Bhatta

Emily Evans

Hui Ren

Vinit Veerendraveer Singh

University of Delaware

2017

Abstract

While studying Software Engineering Principles and Practices, we were exposed to cutting-edge technologies of the current software engineering. As a part of our study, we needed a good theoretical understanding, but we also had to apply it to shape our understanding into something useful. So, out of the two offered projects, we decided to create a simulated version of an Elevator system as it allowed us to experiment our understanding with something which we were very familiar. In this project, the skills we learned in the course are implemented to create, test and verify certain properties of an Elevator Simulation System. The main concepts we have exercised are: modular decomposition based on information hiding, agile methodologies, whole test suite generation, and model checking.

Acknowledgement

It gives us immense pleasure to express our deepest sense of gratitude and sincere thanks to our Professor Dr. Stephen Siegel, Department of Computer and Information Sciences, for his valuable guidance, encouragement and help for completing this work. His suggestions for this project and co-operative behavior are sincerely acknowledged.

We are highly grateful to Mr. Yihao Yan, our TA for the course, for whole-hearted support and guidance. A world of Software Engineering including its various technologies which never occurred to us at an advanced level was there for us to explore, learn and most importantly, integrate with what we have learned so far, thanks to our professor and TA.

At the end we would like to express our sincere thanks to all our friends and others who helped us directly or indirectly during this project.

Table of Contents

Introduction.....	2
1. Problem Definition.....	2
2. Scope and Objectives	2
Requirements Analysis	2
1. Functional Requirements	2
2. Non-functional requirements	3
Specifications	3
Design	4
1. System Architecture.....	4
2. Uses Relations.....	5
3. Is Component Of Relations.....	6
4. Method Signatures	6
5. System Flowchart.....	7
6. Traceability	9
Project Management	9
1. Project Team	9
2. Roles and Responsibilities	10
3. Selected Methodology.....	10
4. Tools Used	10
Verification and Testing	11
1. Unit Testing	11
2. Verification using Model Checking.....	11
Result Analysis	11
1. Results.....	12
2. Limitations and future enhancements	14
Conclusion	15

Introduction

1. Problem Definition

Elevators have evolved so much so that multiple elevators can serve multiple floors. But this creates an optimization problem: Which elevator should server the request made by users. There are various factors used to consider this such as the current floor where the elevator is currently at, the direction the elevator is moving in, number of user elevator is already carrying and many more.

Thus, this creates a need for algorithms that can optimize this NP problem. There are various algorithms that promise to solve this problem such as Nearest Car algorithm, dynamic or static Sector algorithm, and many of these algorithms are patented by elevator manufactures. But these algorithms will only get tested when elevators are working in real life. To solve this problem a simulation of elevator system is required that can test these algorithms and provide information on which of them works the best.

2. Scope and Objectives

The aim of the project is to program and simulate an elevator system that has multiple floors and multiple elevators. The system is designed such that various scheduling algorithms can be tested without having to make changes to any other modules of the simulation program. Thus, the simulation must not only conform to the working of the algorithm, but it should also make sure the requests are served in correct logical order.

Requirements Analysis

The project is intended to simulate a real-life Elevator System, which must consider both the hardware and the software aspects. The requirements hence revolve around making the system feel and behave exactly like a set of elevators serving floors of an actual building. In fact, we have this as our very first functional requirement (F0). In that regard, we have listed out our requirements as functional and non-functional requirements below, from which we later derived our specifications. Also, verification of certain properties of the system using *Model Checking* is also a primary requirement of the project, and we have covered in later section dedicated to verification and testing.

1. Functional Requirements

- F0: The system should simulate a real-world elevator system
- F1: The system should continuously and concurrently wait for and serve all user requests: external and internal.

- F2: The elevators should not change directions until the last request in the committed direction is served.
- F3: The requests should be served in correct logical order i.e. serve the floors consecutively in the committed direction.
- F4: The elevators should move steadily in between floors and stop only at requested floors.
- F5: The system should be able switch between and/or modify scheduling algorithms without changing the processes of other modules.
- F6: The system should be easily extensible to any number of elevators and floors.
- F7: Lights should be turned on as soon as request is initiated and turned off as soon as the request is completed.
- F8: Doors open and close steadily when elevators reach each of the requested floors.

2. Non-functional requirements

- NF1: If an elevator is waiting at the externally or internally requested floor, the doors should simply open and close.
- NF2: There should not be any noticeable delay in the system with the increase in number of requests.
- NF3: Doors should never open or close at intermediate floors.
- NF4: There should be reasonable pause between doors opening and closing.

Specifications

The notion of specifications according to Michael Jackson is that it acts as a bridge between the requirements and the program, which in his words are the phenomenon in the world and the phenomenon in the machine respectively [1]. Hence, once we derive specifications from the requirements, the process deals with just transforming them to the program. After thinking about the way we could implement some of those requirements, we ended up with the following specifications:

- S1: User(s) must be able to continuously and concurrently enter requests to the system.
- S2: All requests (internal and external) should be handled by a single listener.
- S3: Each elevator should maintain its own request set.
- S4: The request set should be structured to allow serving requests in correct logical order.
- S5: Movement of each elevator is independently controlled, irrespective of other elevators.
- S6: Movement of elevators between floors should be continuous and evenly paced.
- S7: Scheduling algorithms should be separate from the core program code.

- S8: User(s) must be allowed to initialize the system with arbitrary number of floors and elevators.
- S9: Lights of internal and external buttons should be instantly responsive.
- S10: Doors should open and close instantly and evenly when a request is completed.

Design

An important phase in any software process is to transform the requirements into design. Since we refined the requirements into specifications in the last section, we will use specifications as the guide to our design. We focus on design to derive an overview of the architecture of the system, the relation between the modules, and understand the overall flow of system. We also check the “traceability” of the design and see how well it aligns with our specifications.

1. System Architecture

Our system architecture was designed with a few initial goals in mind. We wanted to create an architecture that would assist in meeting requirement F5, and the resulting architecture isolates the Scheduling algorithms. In order to switch scheduling algorithms, the only changes that need to be made are in RequestListener. Additionally, we wanted a to design a relatively minimal system that facilitates change and transparency. Our architecture is simple and minimizes dependencies between components.

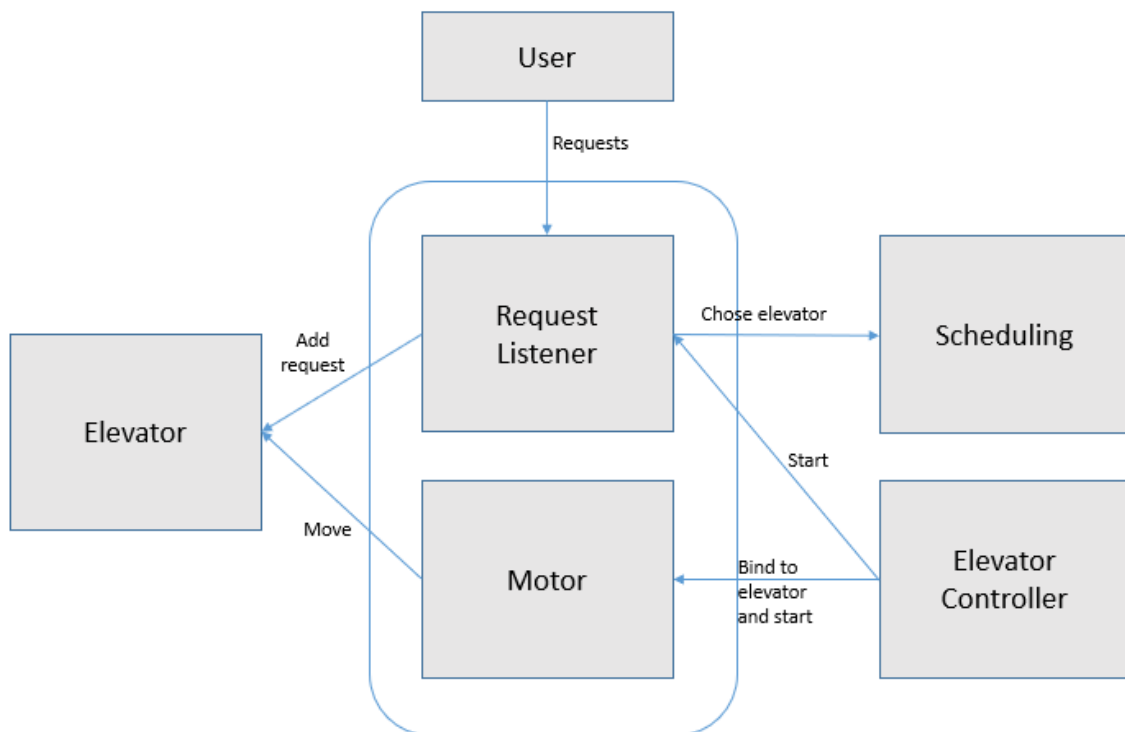


Fig: System Architecture

2. Uses Relations

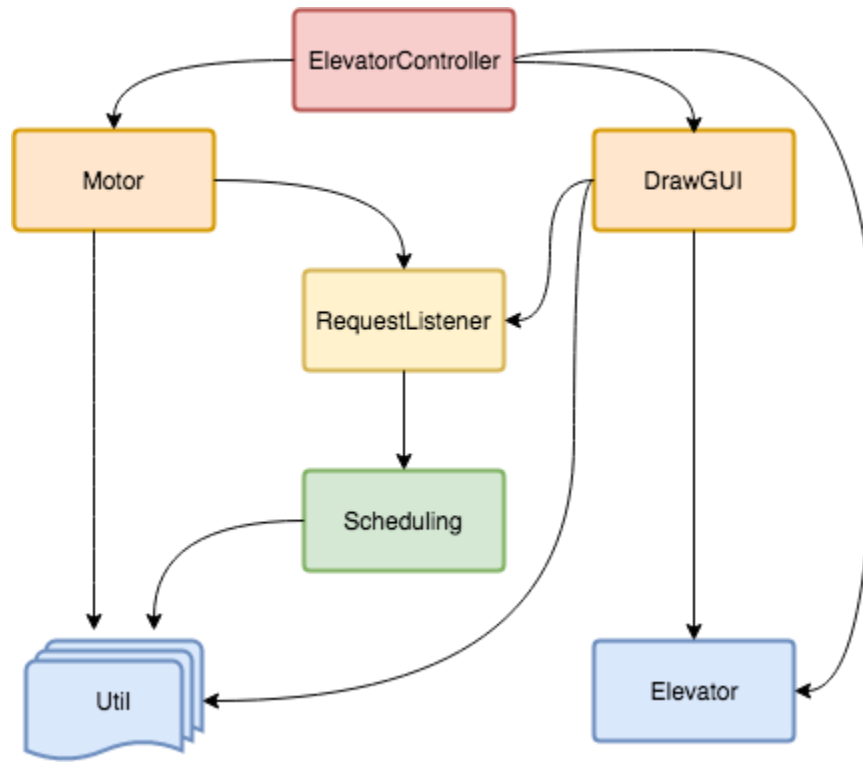


Fig: USES Relations

Like our architecture, we refined our uses relation until it met our goals. We ensured that it is cycle-free (and therefore, a hierarchy) which minimizes dependencies between modules. The level of each module is obvious from the color-coding of the diagram. In summary, ElevatorController is used by nothing, and Util and Elevator each use nothing.

3. Is Component Of Relations

When it came to designing the IS-COMPONENT-OF relation, our minimalism is reflected in the relatively low “height” of the hierarchy. We have few levels in this relation and it is strictly a hierarchy.

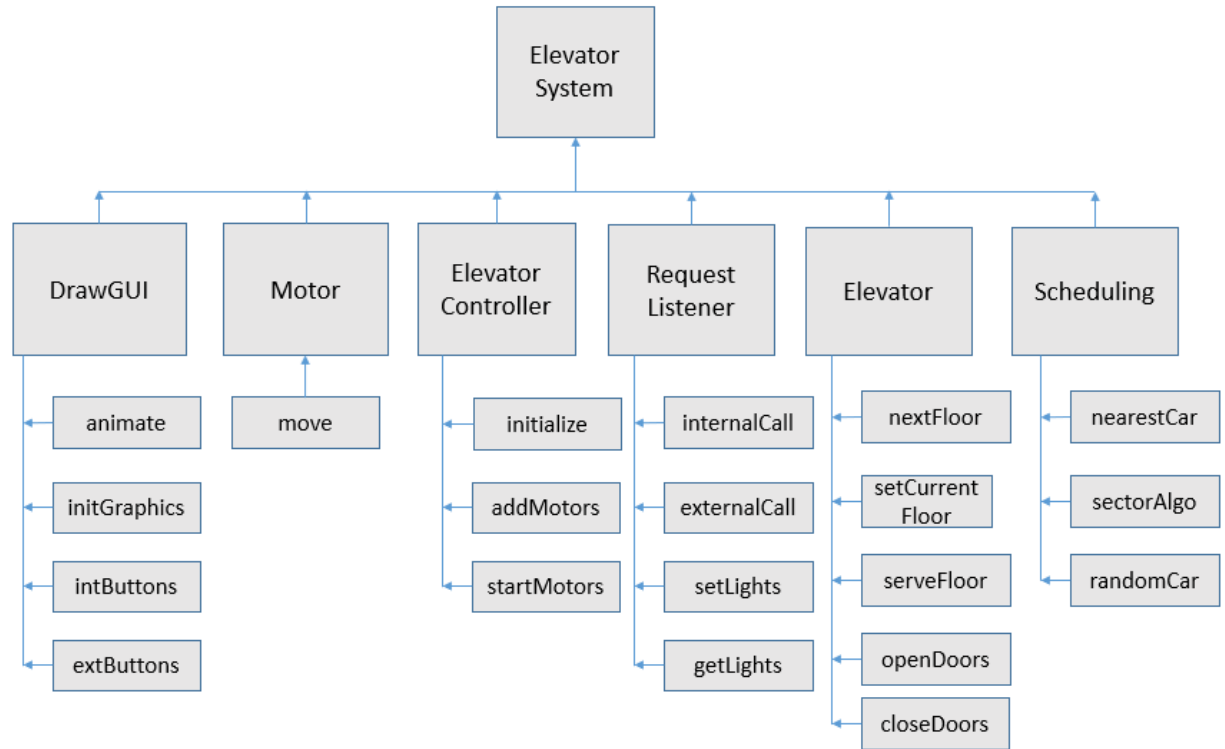


Fig: IS_COMPONENT_OF Relations

4. Method Signatures

Classes	Methods	Descriptions
DrawGUI	animate()	continuously updates GUI
	initGraphics()	creates RequestListener, JFrame, JPanels
	int_Buttons()	creates internal buttons for all elevators and binds them to RequestListener
	ext_Buttons()	creates external buttons for all floors and binds them to RequestListener
Motor	move()	simulates elevator movement
RequestListener	internalCall()	illuminates lights and passes call to serveFloor()
	externalCall()	illuminates lights, calls scheduling method to choose elevator, and passes call to serveFloor()
	setLights()	sets status of specified lights

	getLights()	gets status of specified lights
ElevatorController	initialize()	processes user-supplied number of floors and elevators
	addMotors()	creates elevators, motor threads and binds them
	startMotors()	runs all motor threads
Elevator	nextFloor()	synchronized method that returns the next floor that the elevator will visit
	setCurrentFloor()	changes floor that the elevator is on
	serveFloor()	adds floor to request set of elevator
	openDoors()	simulates opening of elevator doors
	closeDoors()	simulates closing of elevator doors
Scheduling	nearestCar()	chooses elevator to service external call using NearestCar algorithm
	sectorAlgo()	chooses elevator to service external call using fixed sectoring algorithm
	randomCar()	chooses elevator to service external call randomly

5. System Flowchart

The flowchart shows the instantiation process for the Elevator System. While this representation is somewhat casual, there is a methodology to its construction. An outgoing edge from a node represents that the results of the process at that node is a prerequisite for the destination node. In the case that a process may be preceded by fewer than all of its incoming arrow, the paths have been color-coded. A self-loop indicates that the process continues indefinitely after its first execution.

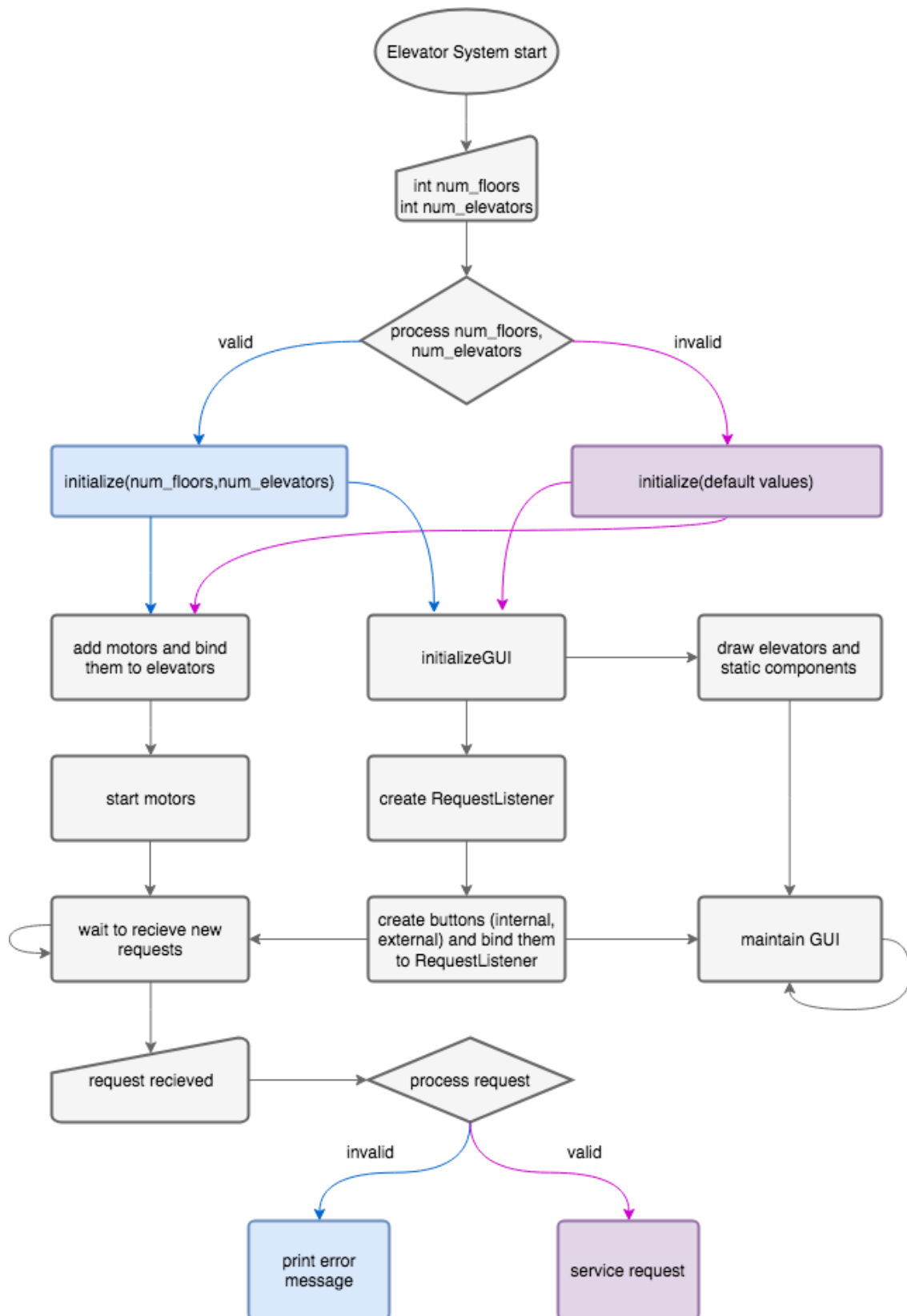


Fig: System Flowchart

6. Traceability

Specification	Element of Design that meets the specification
S1	all buttons bound to RequestListener
S2	1 RequestListener is created
S3	field requestSet of Elevator
S4	requestSet is a TreeSet
S5	field motorThread of Elevator
S6	method move() of Motor
S7	1 call to Scheduling class in RequestListener
S8	initialization of system with command line arguments
S9	setLights() called upon request processing
S10	openDoors(), closeDoors() of Elevator called upon request completion

Project Management

Building the desired system is basically completing a set of tasks that we need to achieve within certain constraints such as the requirements and the submission deadline. So, project management is an unavoidable aspect which involves every phase of the project. Hence, we began by specifying the team, the roles and responsibilities, and identifying the tools and methodologies we will use to track and manage task assignments to the team members.

1. Project Team

We have 4 team members working on this project, with professor and the TA as supervisors.

Resource Person	Role
Stephen Siegel	Supervisor (Professor)
Yihao Yan	Supervisor (TA)
Dixit Bhatta	Team Leader/Developer
Emily Evans	Developer
Hui Ren	Developer
Vinit Veerendraveer Singh	Developer

Table: Project Team

2. Roles and Responsibilities

For improving efficiency, we identified specific responsibilities for each role. Especially the team member roles being “Developer” of all, the responsibilities help assign specific tasks to each member. Following are the responsibilities for individuals

Role	Responsibilities
Supervisor	Providing Suggestions and Feedback Confirming requirements Mentoring/Guiding
Team Leader	Creating and Assigning Tasks Project Progress Tracking
Developer	Collecting requirements Designing Coding/Testing Verification Documentation

Table: Roles and Responsibilities

3. Selected Methodology

We used a combination of Scrum and Extreme programming to complete our project. Extreme programming in a sense that when one person coded, the other person would simultaneously review the code. The reviewer would later add validation checks for the code as applicable. We also used Scrum to organize the project progress in terms of specific “sprints” with specific goals associated with each of them. Although we achieved those goals most of the time, the sprints were not of the same length. So, it was not truly a “time-boxed” iteration. In overall, we believe the method suited the team and we were able to complete the project within the desired time.

4. Tools Used

We considered a variety of tools to manage our project. We first tried to use JIRA with GitHub as they are easily integrated. However, free GitHub repositories are public and have limitations on amount of space we could use. So, we then decided to switch to BitBucket Cloud, which can also be easily integrated with JIRA, and moreover the repositories can be made private for free if a team is limited to 5 members. However, we quickly realized that including the professor and the TA, the team will have 6 members. So, finally we decided to go with SVN as everyone had it already

configured in their systems, and even integrated SVN plugin with Eclipse IDE. A tradeoff of the decision to use SVN was that we now had to manually keep track of backlogs in JIRA.

Verification and Testing

Since the simulation of our elevator system is concurrent thus there is a need for verification using both black box and white box testing. Unit testing is done on various methods of each Java class of our elevator system simulator. To check if our program works correctly for the desired concurrent behavior we use model checking using the Spin verification tool.

1. Unit Testing

Black box testing:

Unit testing is done to check if methods inside of a class are functioning as they should be irrespective to what is occurring at other modules and other methods.

Unit Testing is performed on each of the Java classes using both JUnit and EvoSuite tool.

White box testing:

Our elevator simulation system that is programmed in Java programming language. We translate this simulation system into a model that can be used by Spin verification tool. Spin can be used to check certain correctness properties of the concurrently executing processes such as freedom of deadlock and mutually exclusion. And it can also verify if all those property hold on elevator system for an indefinite amount of time using linear temporal logic.

2. Verification using Model Checking

We have tested the following properties of our elevator system using the Spin verification tool:

- Requests get served in correct logical order as specified by our elevator system.
- All requests are eventually served
- All doors are eventually closes
- All lights turned on due to request are eventually turned off

Result Analysis

A project prototype was completed within desired period for initial demonstration, and the completed system was ready within the deadline for final in-class presentation. Nonetheless, there were some important feedback from the instructor during the presentation, which we considered and fixed the system

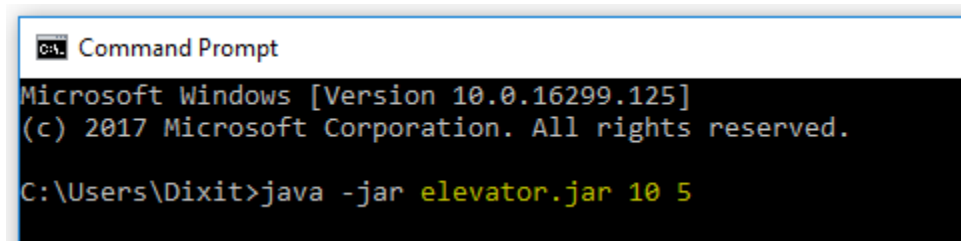
to get the current i.e. “final” system for submission. Following feedback was provided and we fixed the related issues accordingly:

- a) Requirements elicitation should be clear and precise
- b) Refactor code to remove cycle from the design
- c) Uses relation should have all the components, including “utilities”
- d) Methods should be named consistently in the comprises/is component of relation
- e) Properly cite the sources
- f) Combine separate lights icons and buttons into single GUI entity i.e. buttons themselves should light on and off
- g) Perform unit testing and try to achieve a high code coverage
- h) Complete verification of a couple of properties of the system using model checking i.e. Spin.

Hence, our final system has the following features:

1. Results

The final system can be run on an IDE or exported as a JAR file. As per the specifications, we need to run it with two command-line arguments: first represents the number of floors and second the number of elevators in the system. In our system, both arguments should be positive integers greater than zero.



```
Microsoft Windows [Version 10.0.16299.125]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Dixit>java -jar elevator.jar 10 5
```

Fig: Initializing system with 10 floors and 5 elevators from command-line

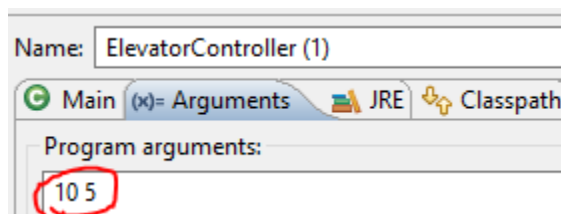


Fig: Setting up command-line arguments in IDE (Eclipse)

Once the system is initialized with command line arguments, we can see the GUI showing the elevators, floors and button panels as below:



Fig: System Initialized with all elevators at bottom floor

Once we make requests using buttons, they light-up and the elevators start moving towards the requested floors using applied algorithm. The internal buttons, in that regard, can be enabled after first request to an elevator is completed, and they propel specific elevator to the requested floors. We can see an instance of such in the figure below.

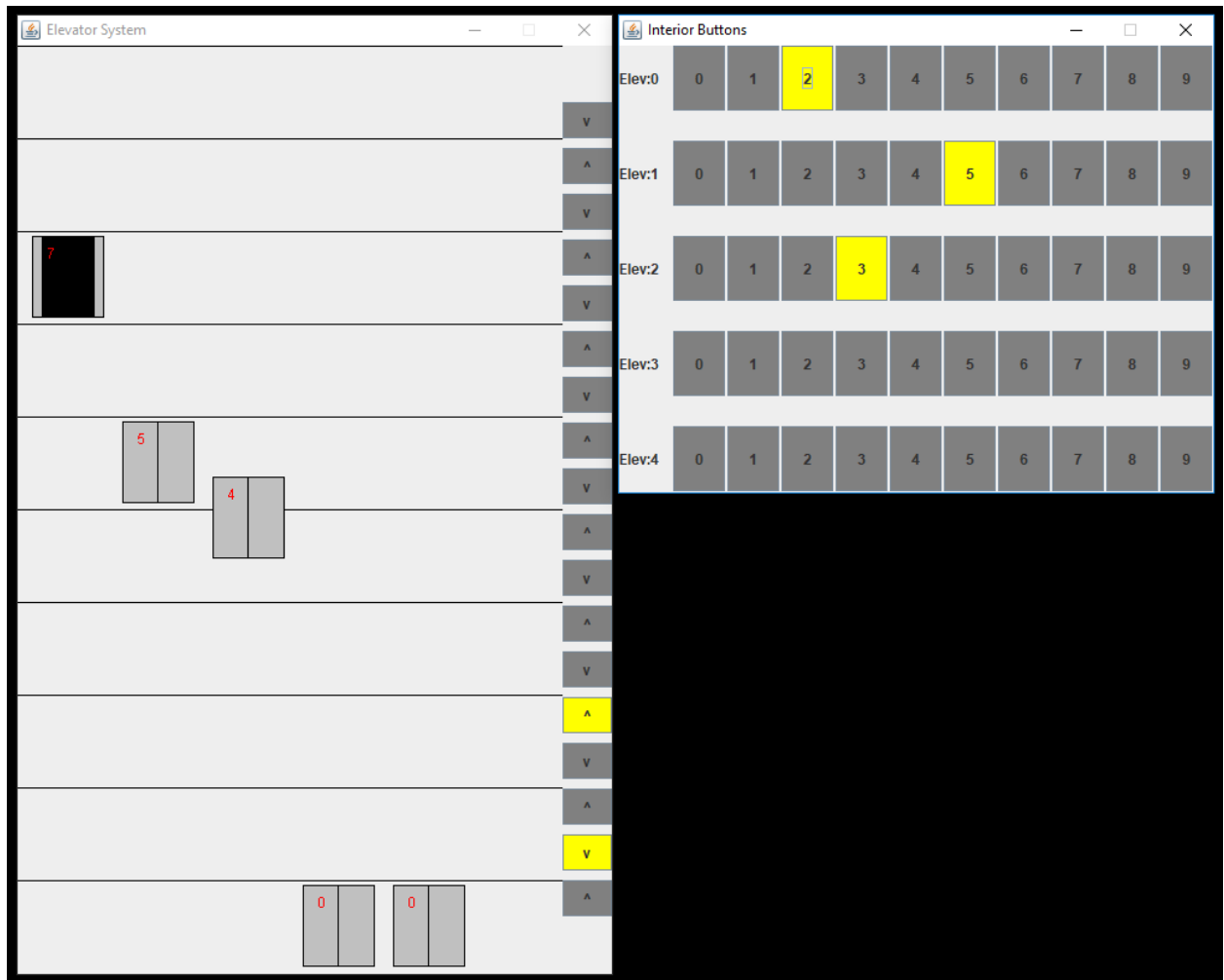


Fig: Elevator System serving some requests

2. Limitations and future enhancements

At the end of the project, we have met all our requirements. Nonetheless, there are some limitations of this system, which can improve upon in future.

- a. The doors can't be interrupted in the middle of closing, in case of any obstruction.
- b. There are no emergency functions.
- c. There are no administrative features to enable/disable elevators or get into maintenance mode.
- d. Passengers and weight capacity considerations are not present.

Conclusion

The project correctly simulates the behavior of a real-world elevator system based on the provided number of floors and elevators. It is very interactive, and users can simultaneously send requests, both internal and external, to observe the movement of elevators. The movements are different based on the chosen scheduling algorithm out of the three we have implemented, switching between which is one of the primary requirements of the project. The code is well tested with unit tests covering significant of core code. We also verified some of the properties if the elevator using model checking (spin), and have not encountered any major bug thus far. So, we can say that the project has been concluded successfully.