

Lab

by

Jason McCauley and Aidan Nestor

Stevens.edu

November 22, 2024

© Jason McCauley and Aidan Nestor
Stevens.edu
ALL RIGHTS RESERVED

Lab

Jason McCauley and Aidan Nestor
Stevens.edu

Table 1: Document Update History

Date	Updates
10/2/2024	<p>JM:</p> <ul style="list-style-type: none">• Under section 1.4, added the EPS figure of the Mealy state machine diagram for the vending machine. Embedded the corresponding Python code file, along with an EPS figure showing the output of the code for each of the three scenarios.• Under section 1.5, added EPS figure of the CRC cards extracted from the user story. Additionally, added EPS figures for the corresponding use case diagram, object diagram, and class diagram. Wrote a detailed paragraph for each of the three diagrams, explaining the specific components of each, how they are connected, along with my thought process behind the designs.
10/3/2024	<p>AN:</p> <ul style="list-style-type: none">• Under section 1.1, added the EPS figure of the revised interactive editor. Responded to the questions regarding the diagram.• Under section 1.2, added the EPS figure of the graphical document editor with support for grouping.• Under section 1.3, added EPS figure of the class diagram for rotating electrical motors. Responded as to why we did not use multiple inheritance.• Under section 1.6, added three EPS figures. One was a use case diagram for the e-Bookstore, the next was a class diagram for the e-Bookstore, and the final was a sequence diagram for the e-Bookstore.

Table 1: Document Update History

Date	Updates
10/16/2024	<p>JM:</p> <ul style="list-style-type: none"> • Under section 2.2, manually calculated the worst, best, and average execution times of the branches in the given binary search tree. • Under section 2.2, added EPS file for the master Simulink model simulating the entire binary search tree, and EPS files for each of the models used. Also added EPS files of the output scope for the original and modified simulation. • Under section 2.2, explained the functionality of each model in the Simulink binary search tree and how they work together to output execution times. Also explained the output scope of the original simulated binary search tree, the issue we encountered. Explained the output scope of the modified simulated binary search tree and the necessary changes to the master Simulink model to achieve those results.
10/17/2024	<p>AN:</p> <ul style="list-style-type: none"> • Under section 2.1, added all EPS files for each component of the Simulink Model. • Under section 2.1, added EPS file for the output of the Simulink Model.
10/23/2024	<p>JM:</p> <ul style="list-style-type: none"> • Under section 3.1, added EPS files of the given software execution model for the server checkout use case, the corresponding Simulink block diagram, and the output scope and signal statistics.
10/24/2024	<p>AN:</p> <ul style="list-style-type: none"> • Under section 3.1, added the explanation for the Simulink diagram shown in Figure 3.2.
10/31/2024	<p>JM:</p> <ul style="list-style-type: none"> • Created chapter for System Execution Modeling, formatted and organized section for Exercise 8.1, each subsection for the corresponding questions.
10/31/2024	<p>AN:</p> <ul style="list-style-type: none"> • Completed questions with help from Jason for Exercise 8.1.

Table 1: Document Update History

Date	Updates
11/6/2024	<p>JM:</p> <ul style="list-style-type: none"> • Under section 5.1, added EPS figures of screenshots from the Excel sheet for the 3CoinExercise, the StoreExercise, and the performance metrics of the StoreExercise • Added explanations under each of the EPS figures to explain my thought process in performing each of the calculations for the 3CoinExercise and StoreExercise.
11/7/2024	<p>AN:</p> <ul style="list-style-type: none"> • Added EPS screenshots under section 5.2 for the code and scope outputs. • Explained why the scopes did not match up with the Excel simulation.
11/20/2024	<p>JM:</p> <ul style="list-style-type: none"> • Created chapter for Abstract Factory, which includes EPS files for the screenshots of the output results when ordering a cheese pizza from the NY store and a veggie pizza from the Chicago store. • Added Java code for concrete classes ChicagoPizzaStore, PlumTomatoSauce, MarinaraSauce, MozzarellaCheese, ReggianoCheese, Garlic, Onion, Mushroom, RedPepper, SlicedPepperoni, FrozenClams, FreshClams, ChicagoPizzaIngredientFactory, PepperoniPizza, ClamPizza, VeggiePizza, ThinCrustDough • Added Java code for abstract classes Sauce, Cheese, Veggies, Pepperoni, Clam, and the driver code MyPizzaApp
11/21/2024	<p>AN:</p> <ul style="list-style-type: none"> • Created Class Diagram for the Abstract Factory. • Created Sequence Diagram for Veggie Pizza from NYPizzaStore.

Table of Contents

1	Advanced Modeling	
	– <i>Jason McCauley and Aidan Nestor</i>	1
1.1	Exercise 3.1	1
1.1.1	Question 1	1
1.1.2	Question 2	1
1.2	Exercise 3.2	2
1.3	Exercise 3.3	2
1.4	Exercise 3.4	3
1.5	Exercise 3.5	6
1.6	Exercise 3.6	9
1.6.1	Use Case Diagram	9
1.6.2	Class Diagram	9
1.6.3	Sequence Diagram	10
2	Simulations with Simulink	
	– <i>Jason McCauley and Aidan Nestor</i>	11
2.1	Exercise 7.1	11
2.2	Exercise 7.2	13
2.2.1	Question 1	13
2.2.2	Question 2	15
3	CasePardoSplit Simulations	
	– <i>Jason McCauley and Aidan Nestor</i>	19
3.1	Exercise 7.4	19
4	System Execution Modeling	
	– <i>Jason McCauley and Aidan Nestor</i>	22
4.1	Exercise 8.1	23
4.1.1	Question 1, 2, and 3	23
4.1.2	Question 4	24
5	System Execution Excel and Simulink Simulations	
	– <i>Jason McCauley and Aidan Nestor</i>	25

5.1	Exercise 8.2	25
5.2	Exercise 8.3	27
6	Abstract Factory	
	– <i>Jason McCauley and Aidan Nestor</i>	30
6.1	Exercise 10.1	30

Chapter 1

Advanced Modeling

– Jason McCauley and Aidan Nestor

1.1 Exercise 3.1

1.1.1 Question 1

- I see that a sheet is made up of lines and boxes.
- Each line is made of line segments that each have two points.
- You can select either a box or a line.
- A point can have one or two line segments.
- You can select a box or a line.
- A buffer is a selection of boxes and lines.

1.1.2 Question 2

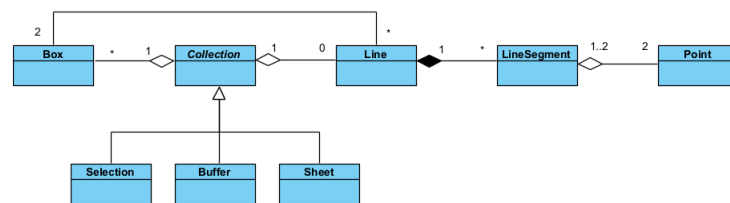


Figure 1.1: Revised class diagram of the partial diagram of an interactive editor.

In this revision, we added a superclass called *Collection* that has a generalization of the three subclasses *Selection*, *Buffer*, and *Sheet*. This allows us to apply the constraint that a line or a box belongs to exactly one buffer, one selection, or one sheet. The *Box* and *Line* classes have aggregation with *Collection* with proper multiplicity to satisfy the constraint.

1.2 Exercise 3.2

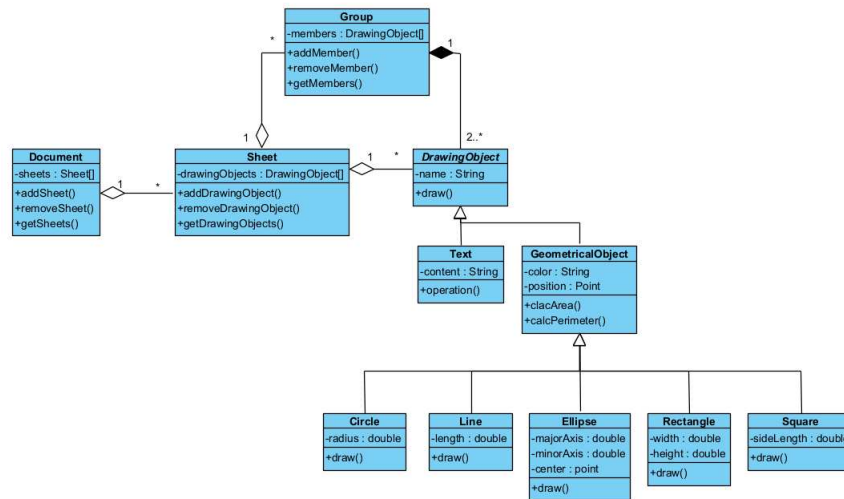


Figure 1.2: Class diagram for a graphical document editor that supports grouping.

1.3 Exercise 3.3

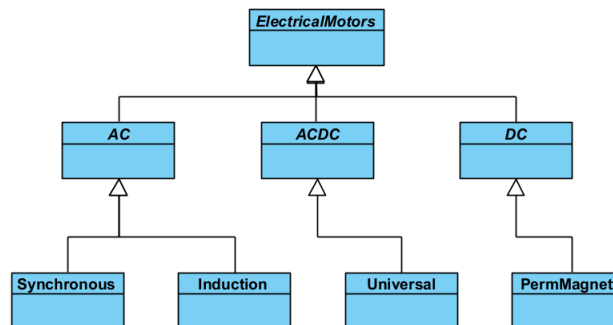


Figure 1.3: Class diagram that represents rotating electrical motors.

We decided not to use multiple inheritance in the initial diagram because we could just make another base class for that specific engine type.

1.4 Exercise 3.4

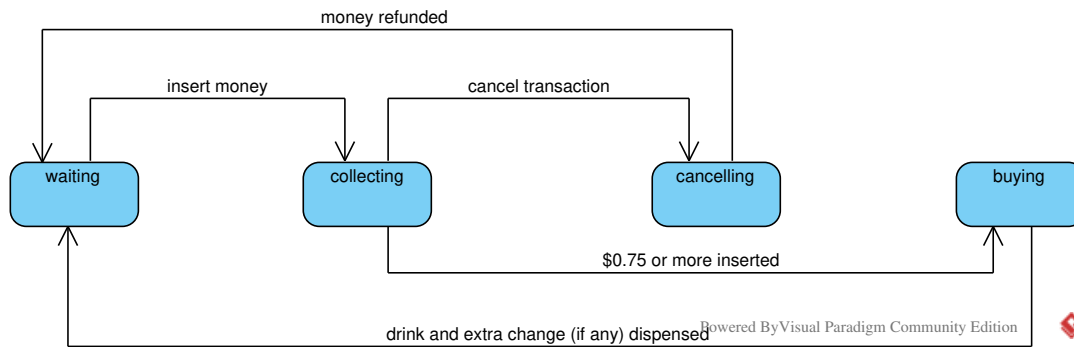


Figure 1.4: Mealy state machine diagram for the vending machine, depicting the four possible states, along with the inputs that connect each state.

```

1 class SM():
2     def __init__(self):
3         self.state = "waiting"
4         print(self.state)
5         self.money_entered = 0
6
7     def insert_dollar(self):
8         self.money_entered += 1.00
9         self.state = "collecting"
10        print(self.state)
11        print(f"Dollar accepted, you have entered ${round(self.money_entered,
12        2)}")
13        if (self.money_entered >= 0.75):
14            self.buy_drink()
15
16    def insert_quarter(self):
17        self.money_entered += 0.25
18        self.state = "collecting"
19        print(self.state)
20        print(f"Quarter accepted, you have entered ${round(self.money_entered,
21        2)}")
22        if (self.money_entered >= 0.75):
23            self.buy_drink()
24
25    def insert_dime(self):
26        self.money_entered += 0.10
27        self.state = "collecting"
28        print(self.state)
29        print(f"Dime accepted, you have entered ${round(self.money_entered, 2
30        )}")
31        if (self.money_entered >= 0.75):
32            self.buy_drink()
33
34    def insert_nickel(self):

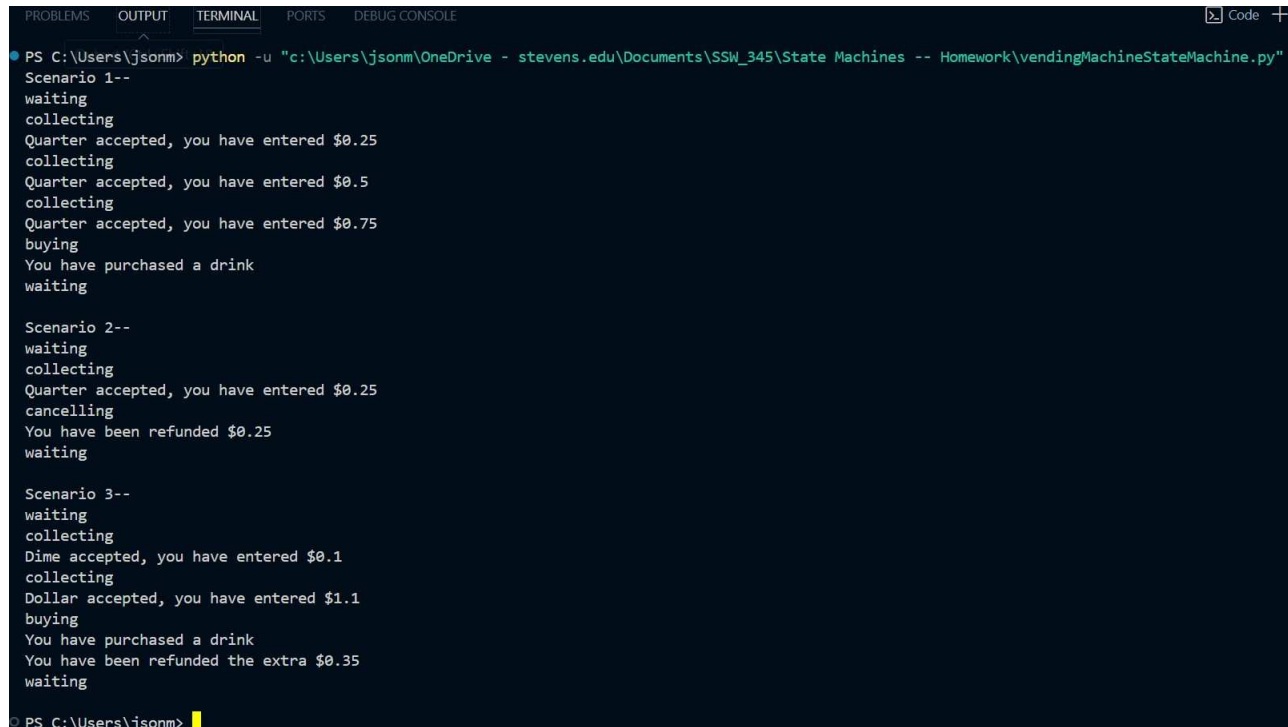
```

```

32     self.money_entered += 0.05
33     self.state = "collecting"
34     print(self.state)
35     print(f"Nickel accepted, you have entered ${round(self.money_entered,
2)})")
36     if (self.money_entered >= 0.75):
37         self.buy_drink()
38
39     def cancel_sale(self):
40         self.state = "cancelling"
41         print(self.state)
42         print(f"You have been refunded ${round(self.money_entered, 2)}")
43         self.money_entered -= self.money_entered
44         self.state = "waiting"
45         print(self.state)
46
47     def buy_drink(self):
48         self.state = "buying"
49         print(self.state)
50         print(f"You have purchased a drink")
51         self.money_entered -= 0.75
52         if (self.money_entered > 0):
53             print(f"You have been refunded the extra ${round(self.
money_entered, 2)}")
54             self.money_entered -= self.money_entered
55             self.state = "waiting"
56             print(self.state)
57
58     print("Scenario 1--")
59     vending_machine1 = SM()
60     vending_machine1.insert_quarter()
61     vending_machine1.insert_quarter()
62     vending_machine1.insert_quarter()
63     print("")
64
65     print("Scenario 2--")
66     vending_machine2 = SM()
67     vending_machine2.insert_quarter()
68     vending_machine2.cancel_sale()
69     print("")
70
71     print("Scenario 3--")
72     vending_machine3 = SM()
73     vending_machine3.insert_dime()
74     vending_machine3.insert_dollar()
75     print("")

```

Listing 1.1: Code from vendingMachineStateMachine.py



```
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
PS C:\Users\jasonm> python -u "c:\Users\jasonm\OneDrive - stevens.edu\Documents\SSW_345\State Machines -- Homework\vendingMachineStateMachine.py"
Scenario 1--
waiting
collecting
Quarter accepted, you have entered $0.25
collecting
Quarter accepted, you have entered $0.5
collecting
Quarter accepted, you have entered $0.75
buying
You have purchased a drink
waiting

Scenario 2--
waiting
collecting
Quarter accepted, you have entered $0.25
cancelling
You have been refunded $0.25
waiting

Scenario 3--
waiting
collecting
Dime accepted, you have entered $0.1
collecting
Dollar accepted, you have entered $1.1
buying
You have purchased a drink
You have been refunded the extra $0.35
waiting

PS C:\Users\jasonm>
```

Figure 1.5: Screenshot showing the output of the code. As seen, for each of the three scenarios, the appropriate states are reached depending on the input, whether it be inserting money, buying a drink, or cancelling the transaction.

1.5 Exercise 3.5

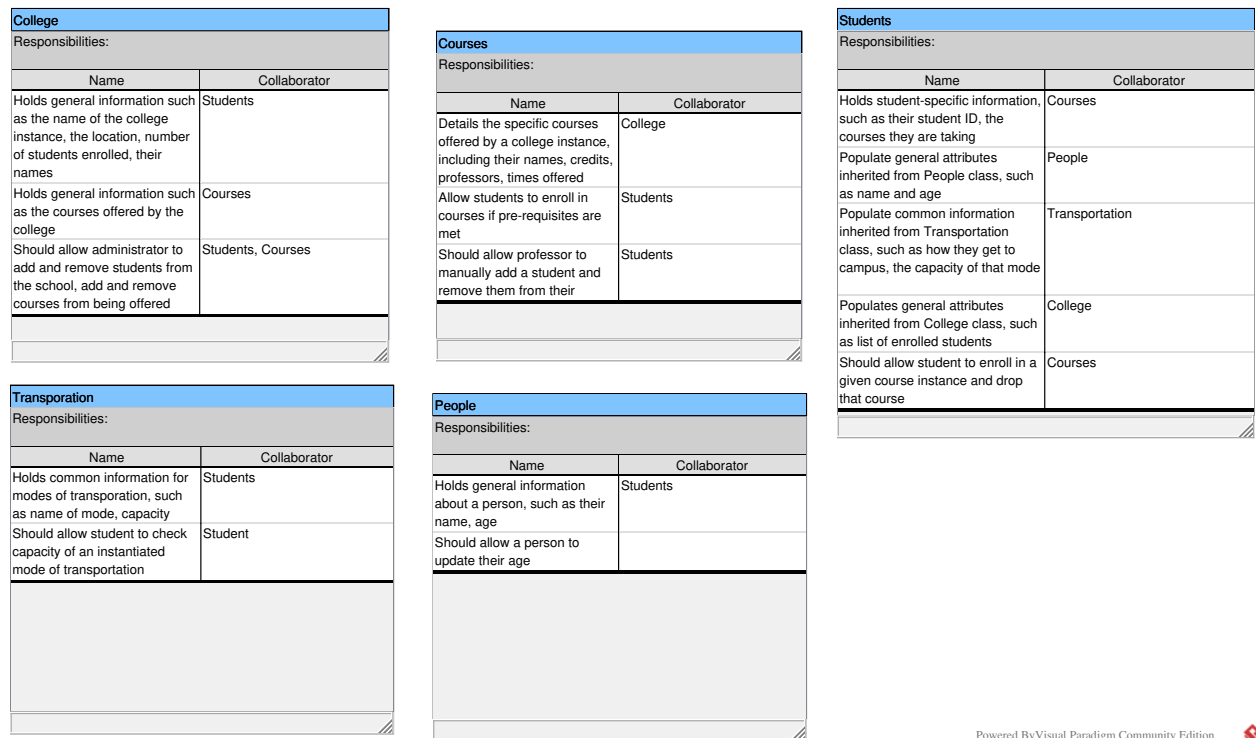


Figure 1.6: CRC Cards depicting the responsibilities and collaborators for each of the five classes extracted from the given user story.

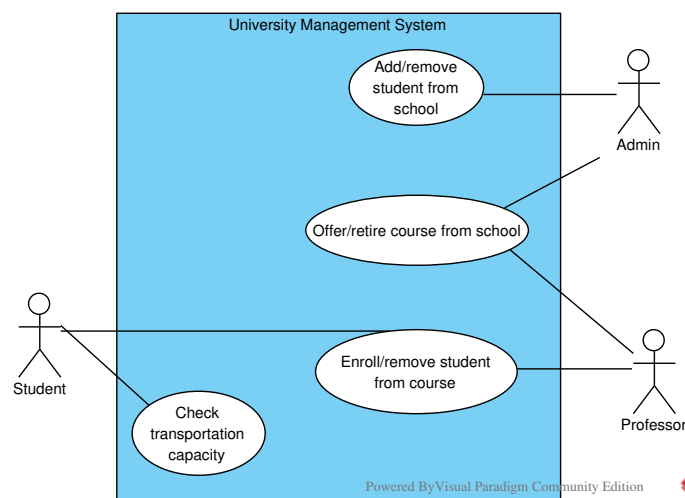


Figure 1.7: Use case diagram for the given user story, depicting the actors along with the use cases they are each associated with.

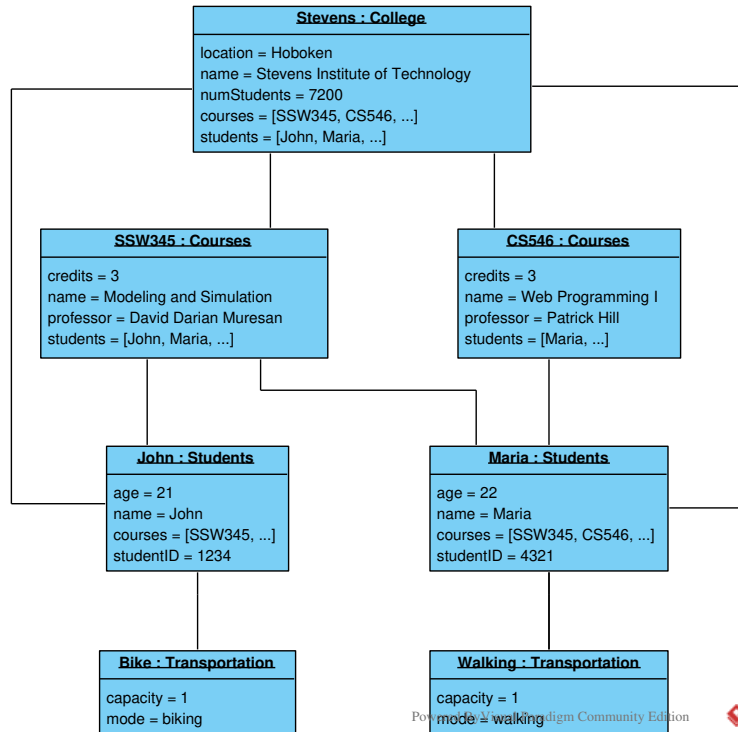


Figure 1.8: Object diagram for the given user story, depicting specific instances for each of the five classes, as well as how they are linked.

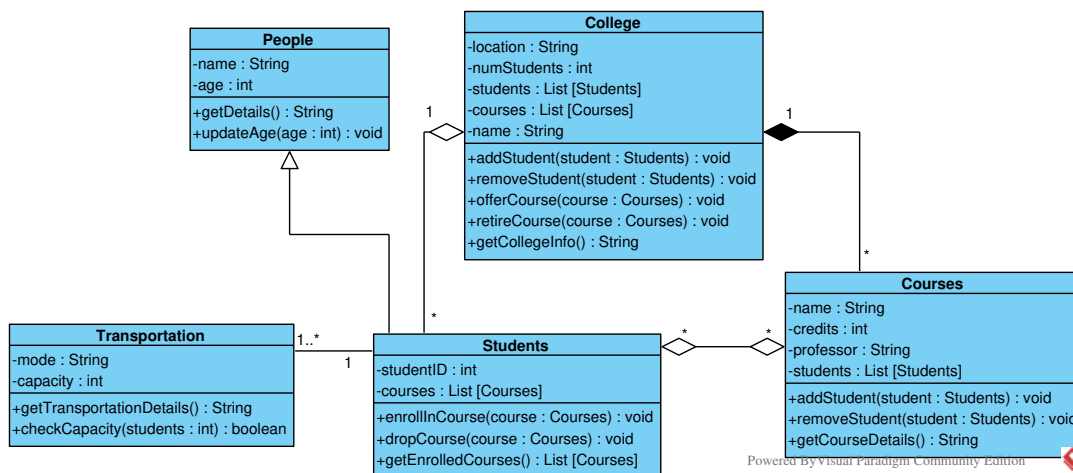


Figure 1.9: Class diagram for the given user story, detailing the attributes and methods for each class, along with the multiplicities and different types of relationships that exist between them.

After defining my classes in the CRC cards, along with each of their responsibilities and collaborators, the next step was to put together a use case diagram to depict the user story. I knew that one actor would be the Student, given that the responsibilities of the Students class include the ability to check the capacity of their transportation method, as well as enroll in and remove them-

self from a course. This would then connect with the Professor actor, which shares the ability to add and remove a student from their course, fulfilling the responsibilities of the Courses class. Additionally, the Professor actor has the ability to offer a course at the school, as well as retire a course, aligning with the responsibilities of the College class. This would then connect with the Admin actor, which shares the Professor’s use case of offering and retiring a course at the school. Furthermore, the Admin actor has the exclusive ability of adding and removing students from the school, fulfilling the responsibilities of the College class.

For my object diagram, I began by creating the instance of College, which in this case, is Stevens Institute of Technology. This instance contains information such as the name of the school, its location, a list of the students attending, and the courses offered. Furthermore, this links to the instances of Courses offered at Stevens, in this example, SSW345 and CS546. The Courses instances contain information such as the credits of the course, the name, the professor, and the list of students taking the course. Consequently, the instances of Courses link to the instances of Students, which according to the given user story, are John and Maria. The instances of Students contain information such as the name of the student, their age, their student ID, and the courses they are taking. Because each of these Students are accounted for in the College class through the list of students, the Stevens instance is linked to the Students instances as well. Finally, as entailed in the user story, John and Maria have different modes of transportation for getting to campus. Therefore, I created different instances of the Transportation class, linking the Bike instance to John and the Walking instance to Maria. Each of the Transportation instance contain information such as the mode of transportation and their capacity.

For my class diagram, I began by creating the People superclass, which contains general attributes, such as the name of a person and their age. This class would then be inherited by the Students class, adding student-specific attributes such as a student ID and the list of Courses that they are taking. Methods for the Students class, as they correspond to the responsibilities outlined in the CRC cards, include enrolling in courses and dropping courses. The Students class then sees one or more instances of the Transportation class, which contains attributes such as the mode of transportation, its capacity, and a method to check the capacity. Each instance of the Transportation class only sees one Students instance. Next, the Students class is an aggregate of the College class, with the many instances of Students seeing just one instance of the College class. The Students class is an aggregate, since each student can exist independent of the College class. Each instance of the College class contains attributes such as the location, the name, the list of students attending, and the list of courses offered. Again, corresponding to the responsibilities on the CRC card, the College class has functionality for adding and removing students from the school, and offering and retiring courses from the school. Next, each instance of College sees many instances of Courses. Courses are a composition of the College class, as if you remove the College class, the Courses instances cannot exist independently. Each instance of the Courses class contains information such as the name of the course, the number of credits, the professor teaching that course, and the list of Students in that course. Courses instances also have the functionality of adding and removing Students from that course. Interestingly, the Courses and Students classes are aggregates of each other, with many instances of Students seeing many instances of Courses. This is because each Students instance contains an attribute for the list of Courses they are taking, and each Courses instance contains an attribute for the list of Students taking that course. However, they are aggregates of each other, since Courses can exist without Students taking them, and Students can exist

without taking Courses, for instance, if they are partaking in a Co-Op semester.

1.6 Exercise 3.6

1.6.1 Use Case Diagram

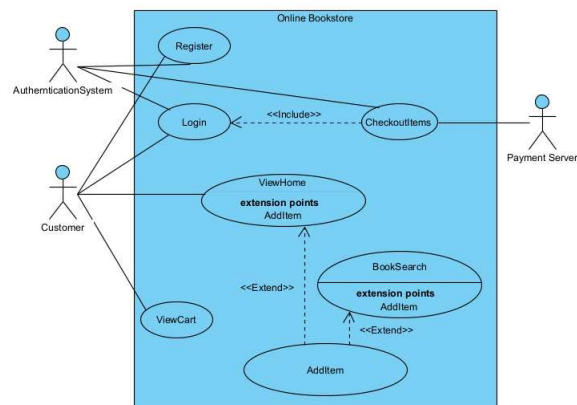


Figure 1.10: Use Case Diagram for the e-Bookstore system.

1.6.2 Class Diagram

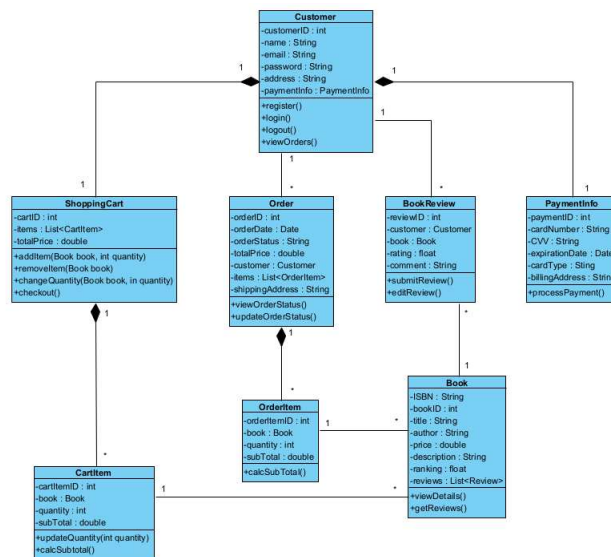


Figure 1.11: Class Diagram for the e-Bookstore system.

1.6.3 Sequence Diagram

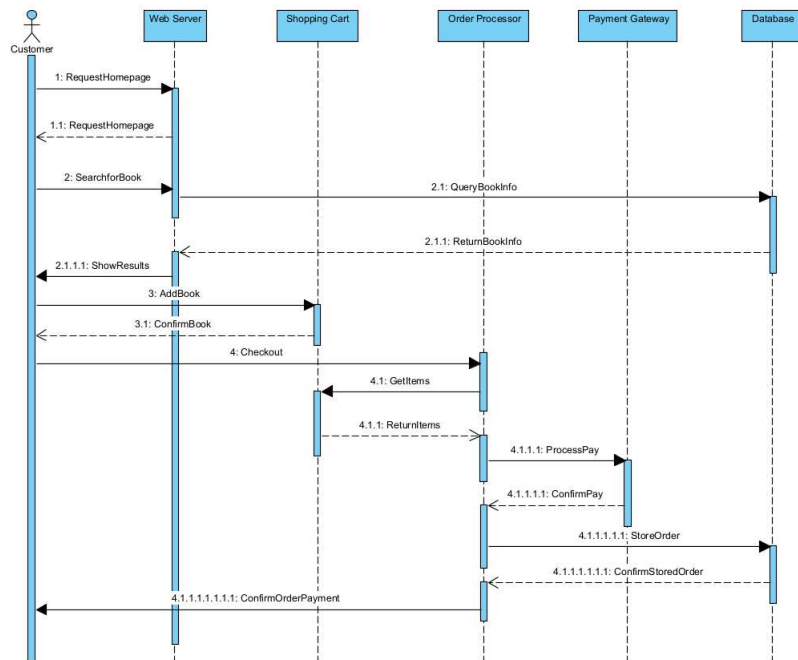


Figure 1.12: Sequence Diagram for the e-Bookstore system.

Chapter 2

Simulations with Simulink

– Jason McCauley and Aidan Nestor

2.1 Exercise 7.1



Figure 2.1: Delay Simulink Model.



Figure 2.2: Loop Simulink Model.



Figure 2.3: CaseMuxOfFive Simulink Model.

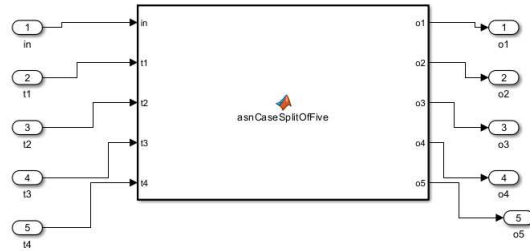


Figure 2.4: CaseSplitOfFive Simulink Model.

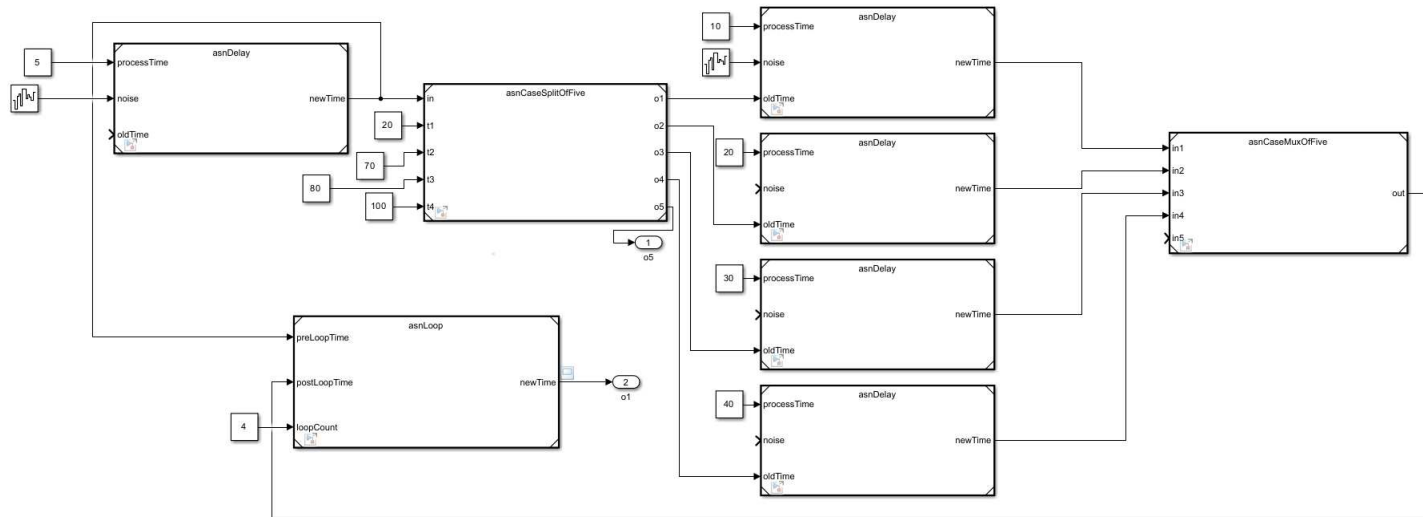


Figure 2.5: Entire Simulink Model for 7.1.

Max = 167.4s Min = 34.39s Avg = 99.79s

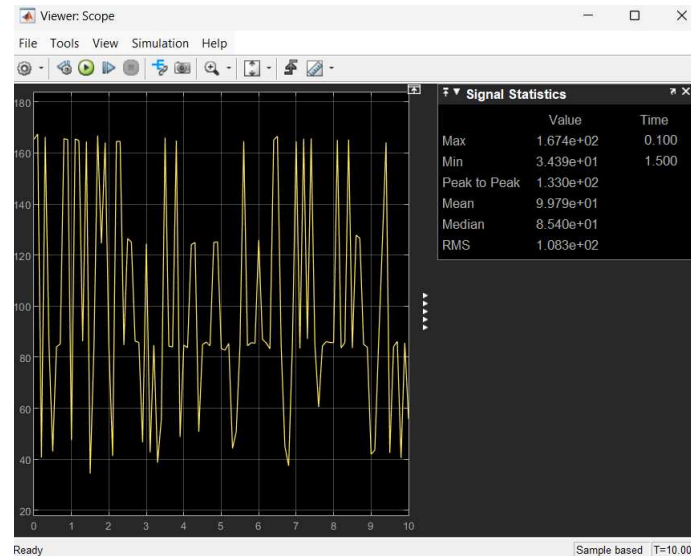


Figure 2.6: Output for above Simulink Model.

2.2 Exercise 7.2

2.2.1 Question 1

To manually calculate the fastest, slowest, and average search times for the given binary search tree, we can begin by calculating the execution time of each branch in a depth-first-search approach. In this approach, we would sum the execution time of each node in a branch, moving from root to leaf node.

$$timeBranch1 = 1 + 2 + 1 + 1 + 1 = 6us$$

$$timeBranch2 = 1 + 2 + 1 + 1 + 2 = 7us$$

$$timeBranch3 = 1 + 2 + 1 + 2 + 1 = 7us$$

$$timeBranch4 = 1 + 2 + 1 + 2 + 2 = 8us$$

$$timeBranch5 = 1 + 2 + 2 + 2 + 1 = 8us$$

$$timeBranch6 = 1 + 2 + 2 + 2 + 2 = 9us$$

$$timeBranch7 = 1 + 2 + 2 + 2 + 1 = 8us$$

$$timeBranch8 = 1 + 2 + 2 + 2 + 2 = 9us$$

$$timeBranch9 = 1 + 3 + 1 + 1 + 1 = 7us$$

$$timeBranch10 = 1 + 3 + 1 + 1 + 2 = 8us$$

$$timeBranch11 = 1 + 3 + 1 + 2 + 1 = 8us$$

$$timeBranch12 = 1 + 3 + 1 + 2 + 2 = 9us$$

$$timeBranch13 = 1 + 3 + 2 + 2 + 1 = 9us$$

$$timeBranch14 = 1 + 3 + 2 + 2 + 2 = 10us$$

$$timeBranch15 = 1 + 3 + 2 + 1 + 1 = 8us$$

$$timeBranch16 = 1 + 3 + 2 + 1 + 2 = 9us$$

Next, we can find the cumulative probability of each branch by multiplying the probability of each node from root to leaf.

$$probBranch1 = 0.4 * 0.3 * 0.7 * 0.3 = 0.0252$$

$$probBranch2 = 0.4 * 0.3 * 0.7 * 0.7 = 0.0588$$

$$probBranch3 = 0.4 * 0.3 * 0.3 * 0.4 = 0.0144$$

$$probBranch4 = 0.4 * 0.3 * 0.3 * 0.6 = 0.0216$$

$$probBranch5 = 0.4 * 0.7 * 0.3 * 0.5 = 0.042$$

$$probBranch6 = 0.4 * 0.7 * 0.3 * 0.5 = 0.042$$

$$probBranch7 = 0.4 * 0.7 * 0.7 * 0.3 = 0.0588$$

$$probBranch8 = 0.4 * 0.7 * 0.7 * 0.7 = 0.1372$$

$$probBranch9 = 0.6 * 0.5 * 0.8 * 0.5 = 0.12$$

$$probBranch10 = 0.6 * 0.5 * 0.8 * 0.5 = 0.12$$

$$probBranch11 = 0.6 * 0.5 * 0.2 * 0.6 = 0.036$$

$$probBranch12 = 0.6 * 0.5 * 0.2 * 0.4 = 0.024$$

$$probBranch13 = 0.6 * 0.5 * 0.5 * 0.3 = 0.045$$

$$probBranch14 = 0.6 * 0.5 * 0.5 * 0.7 = 0.105$$

$$probBranch15 = 0.6 * 0.5 * 0.5 * 0.5 = 0.075$$

$$probBranch16 = 0.6 * 0.5 * 0.5 * 0.5 = 0.075$$

To find the best and worst execution times, we simply look at which branch had the lowest and highest cumulative execution times respectively, as they are independent of probability. With that being said, the best execution time was Branch1, with a time of 6 us. Furthermore, the worst execution time was Branch14, with a time of 10 us. Now, to find the average execution time of the entire binary search tree, we have to take path probability into account. To do this, we will multiply each branch's cumulative execution time by their corresponding cumulative probability. This will give us the weighted execution time of each branch, and from there, we can sum them to get the total weighted execution time of the binary search tree.

$$avgExecutionTime = timeBranch1 * probBranch1 + ... + timeBranch16 * probBranch16 = 8.29us$$

2.2.2 Question 2

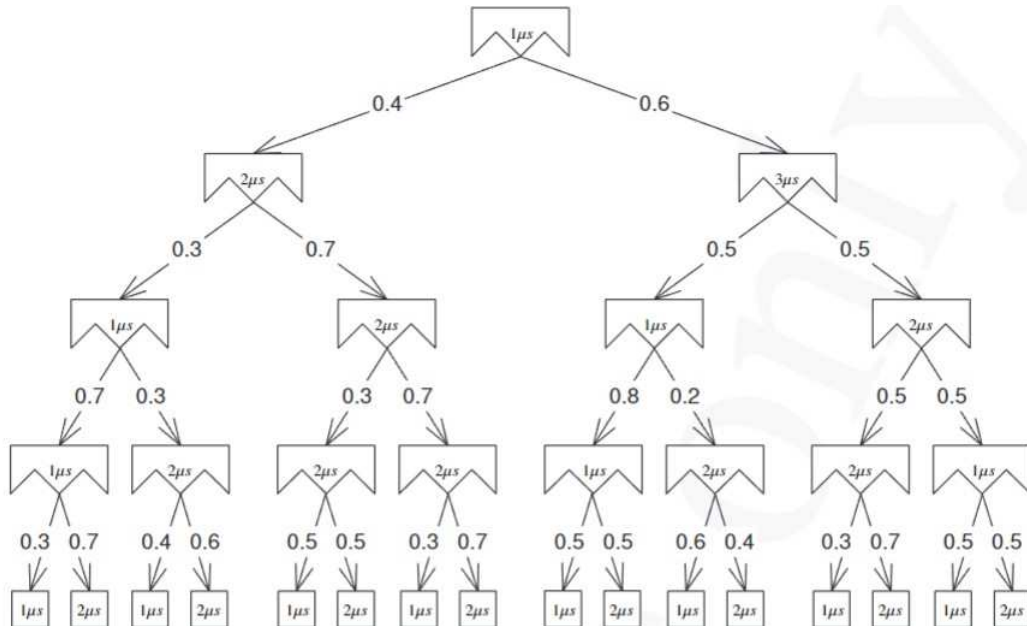


Figure 2.7: Binary search tree, provided in the spec sheet as Figure 7.11.

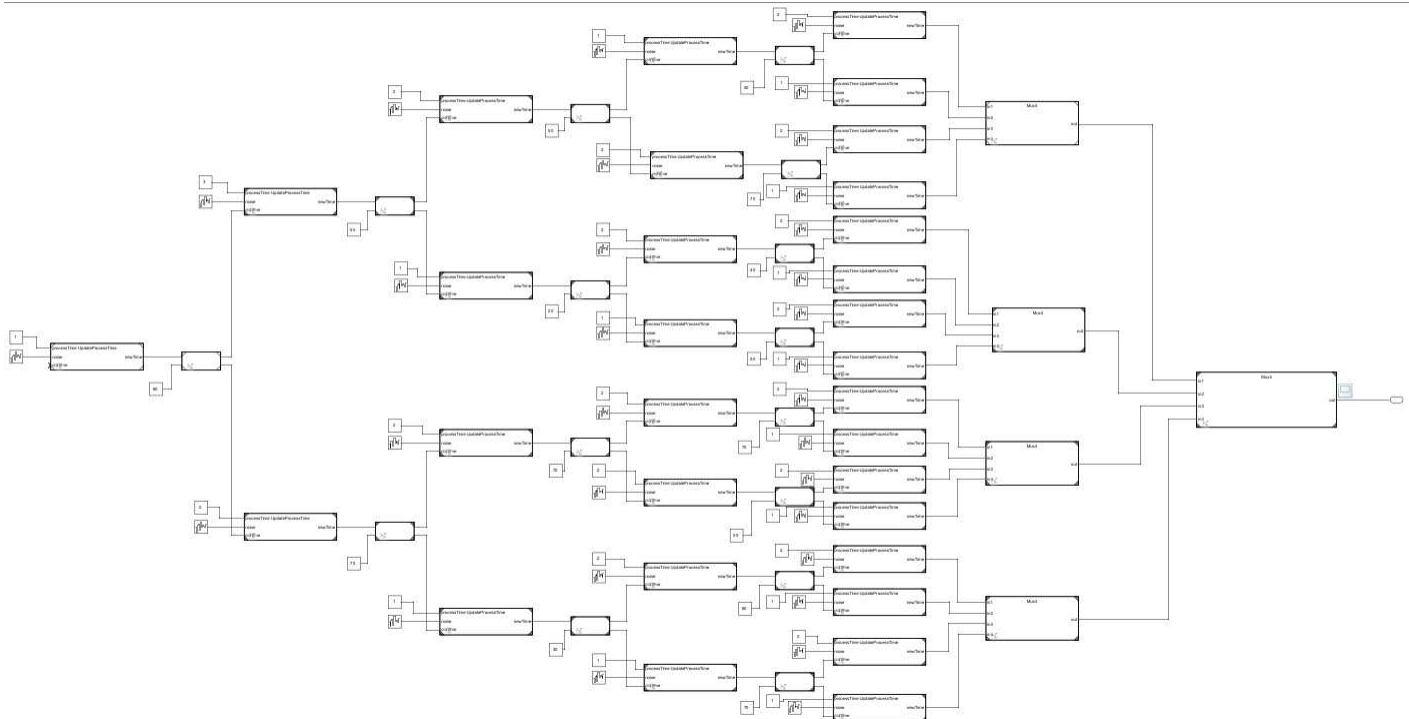


Figure 2.8: Simulink model for the above binary search tree.

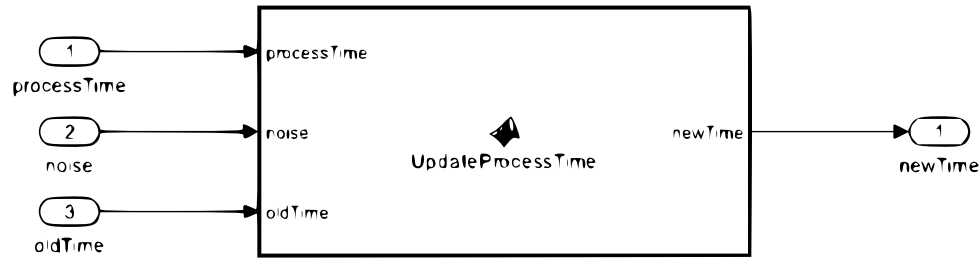


Figure 2.9: UpdateProcessTime Simulink model.

Each node in the binary search tree is represented in Simulink as an UpdateProcessTime model linked to a CaseNode2 model. The UpdateProcessTime model was provided to us in the spec sheet, and takes in the process time of the current node, a noise factor, and the time from the previous node. The model functions by checking the number of arguments entered – if there are less than 3, the oldTime input will automatically be assigned to 0, which occurs at the start of the binary search tree (since there is no previous node). The new time is then calculated by summing the time from the previous node with the inputted process time and the noise for that node.

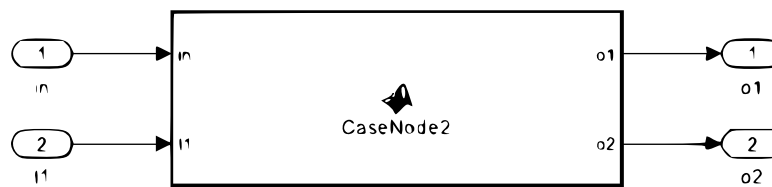


Figure 2.10: CaseNode2 Simulink model.

That new time is then fed into the CaseNode2 model, which was also provided to us in the spec sheet. In the CaseNode2 we also provide a threshold input t1. Let's say the inputted threshold was 40 – a random number (between 0 and 1, multiplied by 100) will be generated inside of the CaseNode2 model. If that randomly generated number falls between 0 and the inputted threshold, 40, it will set the output o1 equal to the inputted new time, and it will be passed into the next UpdateProcessTime model. In that example, the output o2 would be set equal to -1 and passed to the sibling node, effectively meaning that the branch was not activated. If that randomly generated number was to instead fall outside of that range of 0 and the inputted threshold, the output o2 would then be assigned to the inputted new time, and then passed to the next UpdateProcessTime model. Conversely, the output o1 would then be assigned to -1, and passed to the sibling node.

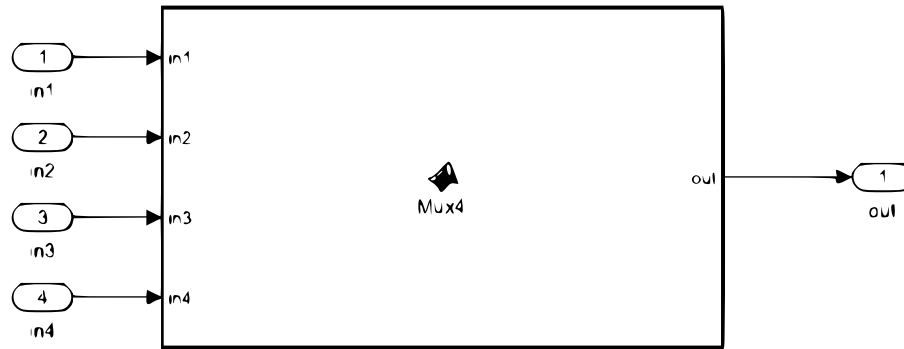


Figure 2.11: Mux4 Simulink model.

After all of the branch execution times have been calculated, we feed the outputted times into a multiplexer. This multiplexer, called Mux4, was also provided in the spec sheet, and takes in 4 inputs. It will then check for the first of the 4 inputs that is a positive value (meaning that the branch was activated). If none of the 4 inputs are positive, a -1 will be outputted. Since we have 16 branches in total, we will use 4 Mux4 models at the first output layer, and then feed those outputs into a second layer Mux4 model, which can be seen in the Simulink model provided. Essentially, this is filtering out all of the branches that were not activated and have an outputted time of -1 by assigning the output of each multiplexer to the first positive input. The output of that final multiplexer will then be viewed with a scope.

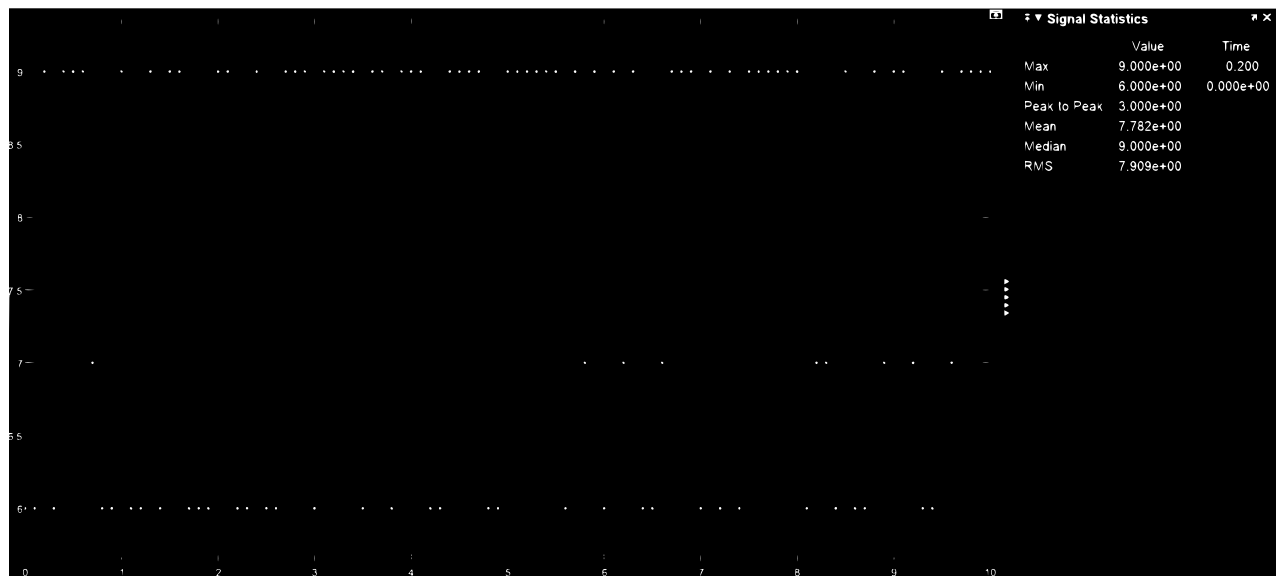


Figure 2.12: Output of the simulated binary search tree.

As seen in the scope above, the minimum execution time was 6 us as expected, however, the maximum execution time was 9 us when it should have found 10 us. After lots of experimentation, we found that this issue arises with the seeding of the rand function in MATLAB. When it generates

a random number in the first CaseNode2 model, it uses that same random number for the entire model. This becomes problematic, as for some reason, the randomly generated number never activates the branch with the highest execution time. This branch (starting from the root) would be right-right-left-right. Once the program reaches right-right, the randomly generated number never falls outside of the inputted threshold, which would be $0 \leq x \leq 50$. Because of this, the provided binary search tree, when simulated, never reaches the path of worst execution time, and appears to be 9 us. Furthermore, this causes the mean running time to decrease – according to the scope, it is only 7.78 us, where our calculations showed 8.28 us.

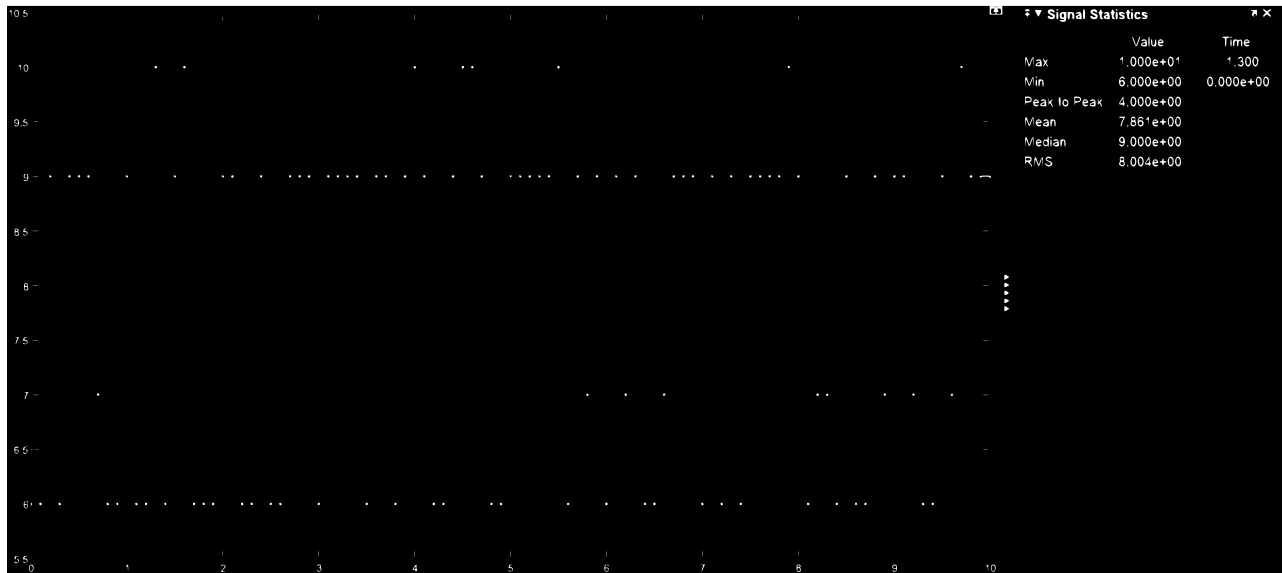


Figure 2.13: Output of the simulated binary search tree with modified case node probability.

One way that we were able to fix this issue of the worst execution path not being reached was by modifying the threshold at the right-right node. For the scope above, I inputted a threshold of 40, meaning that if the randomly generated number fell in the range of $0 \leq x \leq 40$, the right child node would activate. Moreover, this increases the chance of the randomly generated number falling outside of the threshold from 50% to 60%, which would activate the left child node that we are looking for. Now, when this simulation is run, changing the inputted threshold at the right-right node to 40, keeping everything else the same, we see a few instances of that path being activated. The maximum execution time is now the 10 us that we expected, and the minimum execution time is still 6 us. The average execution time only increased slightly, however, up to 7.86 us, which makes sense, since only a few instances reached the expected maximum execution time.

Chapter 3

CasePardoSplit Simulations

– Jason McCauley and Aidan Nestor

3.1 Exercise 7.4

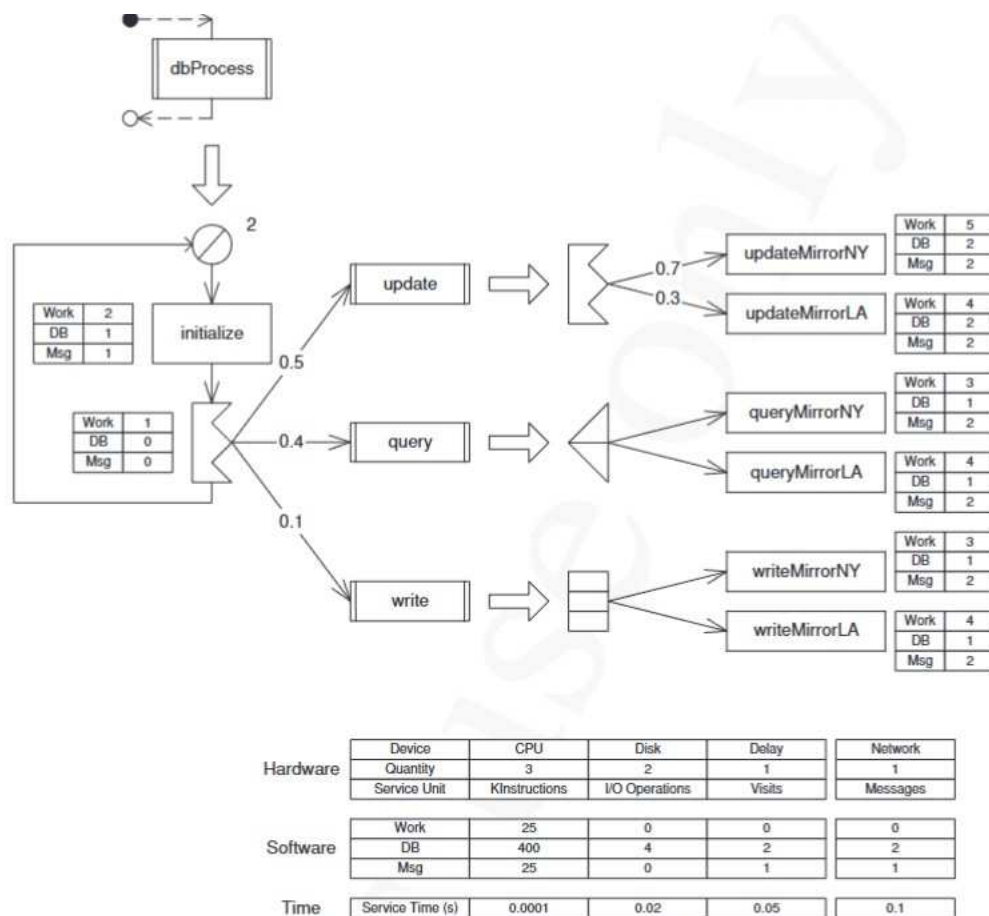


Figure 3.1: Given software execution model for server checkout use case.

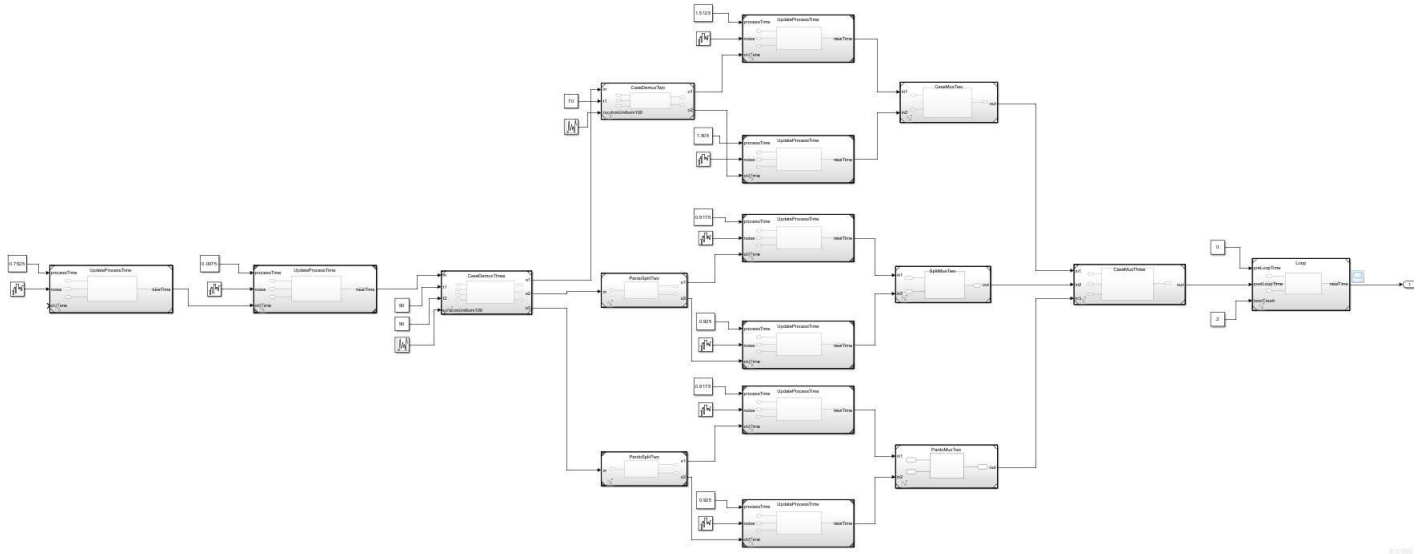


Figure 3.2: Simulink block diagram for the given software execution model.

The Simulink diagram begins with the initialize node, inputting its execution time and Gaussian noise. The model then proceeds to the first case node, where the execution time is updated to incorporate the Work call. Thereafter, the case node demuxes the execution time into three possible paths. Additionally, we input a random number between 0 and 100 that is generated by seed, so that the random number at each case node demux is not the same for the entire model. From there, the first potential path is another case node, which has a 50% chance to be reached, as indicated by the inputted threshold. The next potential path leads to a split node, which has a 40% chance to be reached – we represent this by inputting the threshold 90, as there is a 40% chance a randomly generated number falls between 50 and 90. Lastly, if the randomly generated number does not fall between either threshold, it will reach a pardo node, which is a 10% chance. At the case node, we repeat the process of generating a random number according to seed, and provide a threshold of 70, since there is a 70% chance a randomly generated number will be between 0 and 70 and follow the upper branch. Conversely, there is a 30% chance the randomly generated number will fall outside of this threshold, and follow the lower branch. For each branch of this case node, we continue to update the execution time according the provided model, incorporating noise. We then mux the two branches of that case node by outputting the first branch that is not negative. Similarly, for the split node, we continue to update the execution time for each branch according to the model, however, when we mux them, we only output the shortest of the two execution times. Lastly, for the pardo node, we continue to update the execution time for each branch according to the model, but when muxed together, output the longest of the two execution times. After muxing each the case node, split node, and pardo node to their respective outputs, we can now mux the three of them together so that the output for the entire model is the first non-negative output. Lastly, we have to incorporate the loop – since there is no execution before the loop, we can simply multiply the output of the model by the number of iterations, which in this case, is 2, to give us the entire execution time for the server checkout.

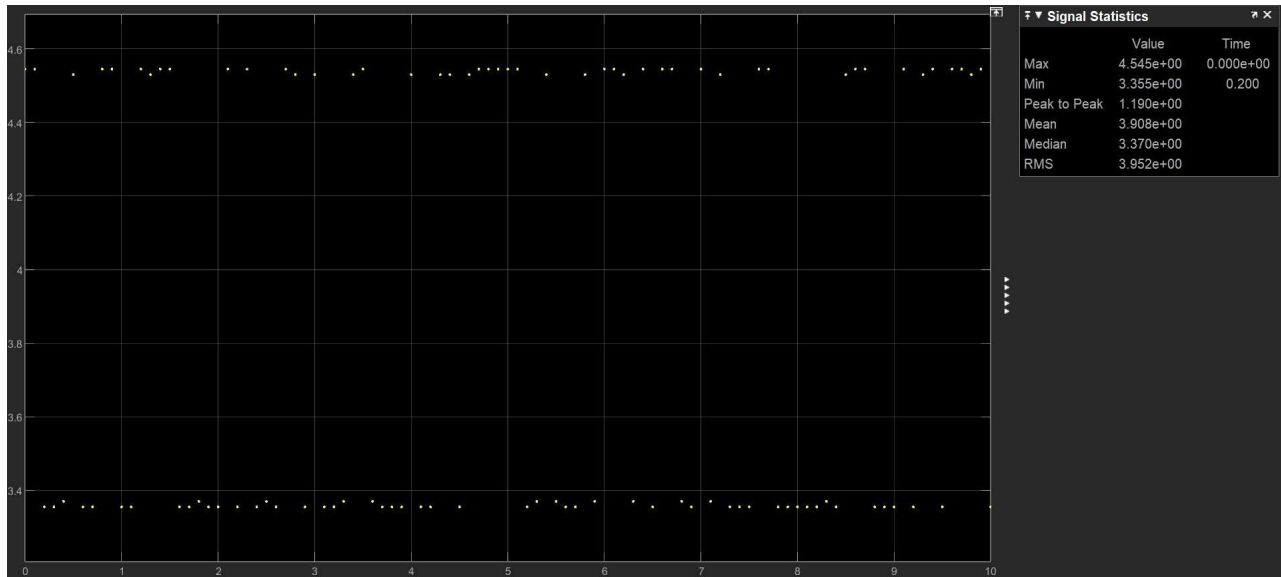


Figure 3.3: Output scope and signal statistics for the above Simulink block diagram.

Chapter 4

System Execution Modeling

– Jason McCauley and Aidan Nestor

4.1 Exercise 8.1

4.1.1 Question 1, 2, and 3

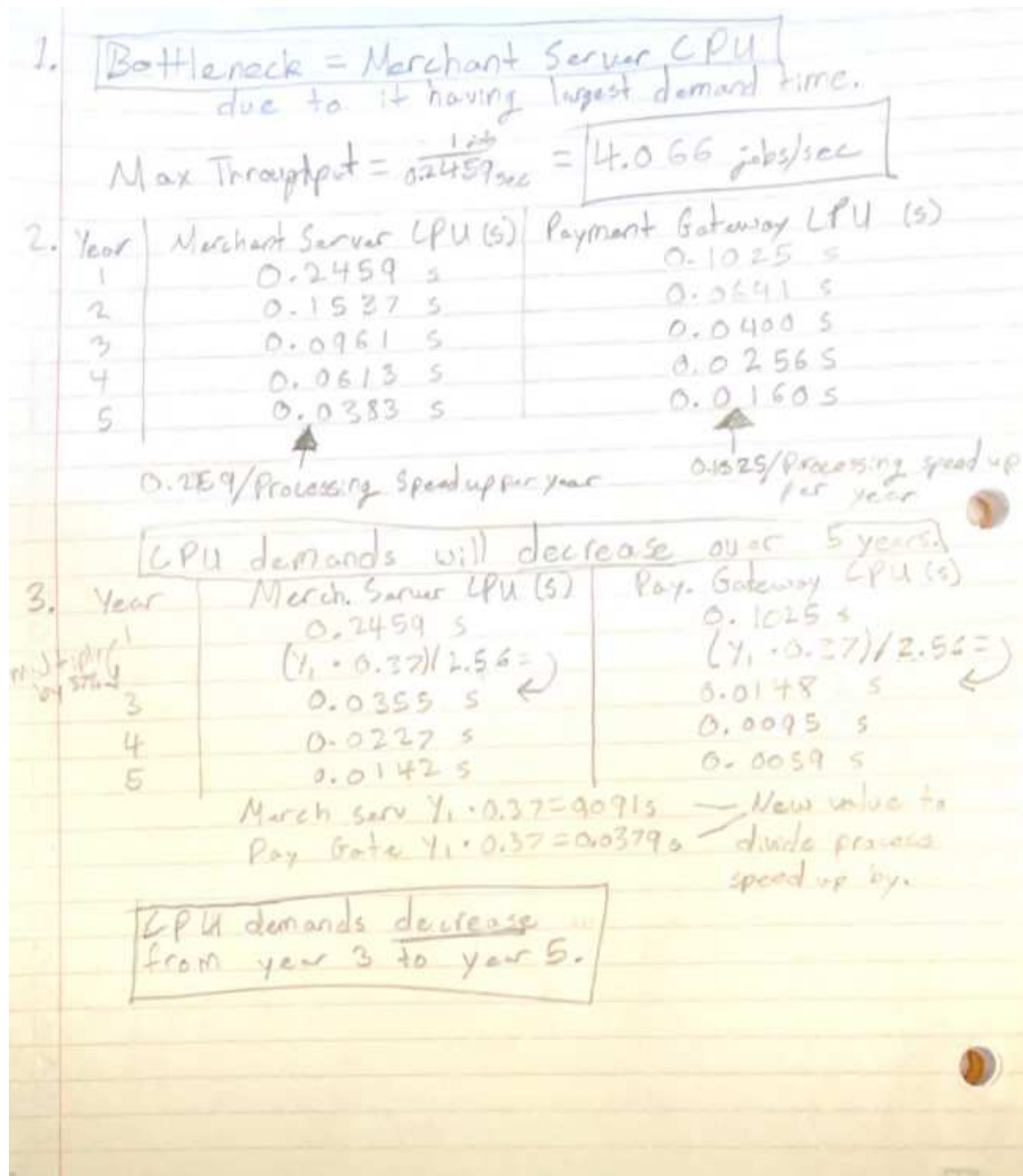


Figure 4.1: Questions 1, 2, and 3

4.1.2 Question 4

4. CPU util for non-SFT = 60% @ merchant server

$$Y_1 \#CPU = \frac{0.2459 + 0.1025}{0.6 \cdot 0.3484} = 1.667$$

$$Y_2 \#CPU = \frac{0.1537 + 0.0691}{0.6 \cdot 0.3484} = 1.04$$

$$Y_3 \#CPU = \frac{0.0961 + 0.04}{0.6 \cdot 0.3484} = 0.45$$

#CPU's	Year
2	1
2	2
1	3
1	4
1	5

$$Y_4 = \frac{0.0513 + 0.0254}{0.6 \cdot 0.29} = 0.41$$

$$Y_5 = \frac{0.0313 + 0.016}{0.6 \cdot 0.06} = 0.26$$

We pledge our honor we have abided by the Stevens Honor System.

Figure 4.2: Question 4

Chapter 5

System Execution Excel and Simulink Simulations

– Jason McCauley and Aidan Nestor

5.1 Exercise 8.2

Rand 1	Rand 2	Rand 3	Coin 1	Coin 2	Coin 3	Result	Outcome	Frequency
2	1	2 T	H	T	THT		HHH	124
2	1	1 T	H	H	THH		HHT	133
2	2	2 T	T	T	TTT		HTT	107
2	2	2 T	T	T	TTT		HTH	129
1	1	2 H	H	T	HHT		TTT	118
2	2	1 T	T	H	TTH		TTH	142
1	1	2 H	H	T	HHT		THH	125
2	2	2 T	T	T	TTT		THT	122
1	2	1 H	T	H	HTH			
1	2	1 H	T	H	HTH			
1	1	1 H	H	H	HHH			
2	2	2 T	T	T	TTT			
2	2	1 T	T	H	TTH			
1	1	2 H	H	T	HHT			

Figure 5.1: Screenshot from the 3CoinExercise Excel sheet.

Completing the 3CoinExercise sheet was pretty straightforward, given the 2CoinExample that was provided. I created a third random number column that would generate either 1 or 2, and then the third coin be would assigned either H or T depending on the number generated. Then, I appended the H or T from the three coins to get the result for that trial. To determine the frequency of each outcome, it was just a matter of iterating through the Result column and checking to see how many of the results matched each possible outcome.

Cust. #	Interval	(Service)		Arrive Clock	Begin Checkout Clock	Wait in Queue Time	End Shopping Clock	Time in Store	Time Owner Idle
	Arrival Time	Shopping Time	Checkout Time						
1	8	7	4	8	15	0	19	11	0
2	9	4	4	17	21	0	25	8	2
3	8	6	2	25	31	0	33	8	6
4	3	1	1	28	29	4	34	6	0
5	2	6	4	30	36	0	40	10	2
6	6	5	4	36	41	0	45	9	1
7	9	7	2	45	52	0	54	9	7
8	9	4	1	54	58	0	59	5	4
9	9	6	4	63	69	0	73	10	10
10	7	5	3	70	75	0	78	8	2
11	10	4	2	80	84	0	86	6	6
12	3	3	3	83	86	0	89	6	0
13	1	6	1	84	90	0	91	7	1
14	5	2	4	89	91	0	95	6	0

Figure 5.2: Screenshot from the StoreExercise Excel sheet.

Completing the columns for the StoreExercise was also a relatively straightforward process. For columns such as Interval Arrival Time, Shopping Time, and Service Time, we were told to implement the `=RANDBETWEEN()` function as well as the numbers to use. Many of the other columns, such as Begin Checkout Clock, End Shopping Clock, and Time in Store were straightforward to implement or derive based on the provided variable descriptions. For Arrive Clock, I initialized the first row by referencing the first Interval Arrival Time. For subsequent rows, I added the previous Arrive Clock to the current Interval Arrival Time. To calculate Wait in Queue Time, I initialized the first row with 0, and for subsequent rows checked if the current customer's Begin Checkout Clock was greater than the previous customer's End Shopping Clock – this would mean that there is no line. However, if the current customer's Begin Checkout Clock was less than the previous customer's End Shopping Clock, we could take the difference to determine Wait in Queue Time. For Time Owner Idle, I initialized the first row with 0, and for subsequent rows checked if the current customer's Begin Checkout Clock was less than the previous customer's End Shopping Clock – if so, this would indicate that the owner would not have time to idle. However, if the current customer's Begin Checkout Clock was greater than the previous customer's End Shopping Clock, the difference would indicate the time the owner had to idle.

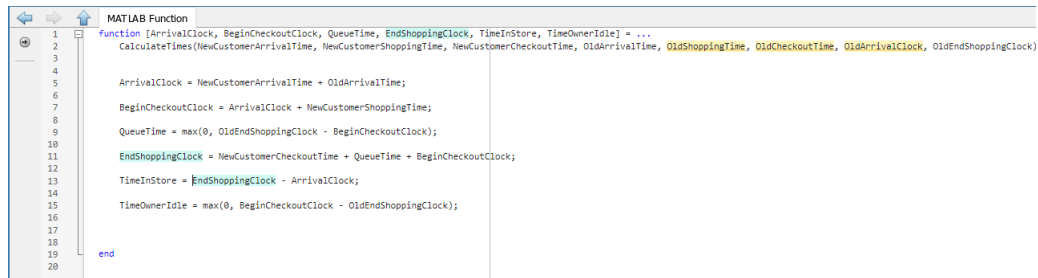
Performance Metrics based on simulation data:	
Throughput =	0.208333333
Utilization =	52.38095238
Response Time =	5.266666667

Figure 5.3: Screenshot of the StoreExercise performance metrics calculated in the Excel sheet.

After understanding each of the variables in the StoreExercise, calculating the performance metrics became straightforward as well. To calculate throughput as customers per minute, I divided 105 by the observed time, which I calculated as the difference between the last customer's End

Shopping Clock and the first customer's Arrive Clock. Next, to calculate utilization, I first had to calculate OwnerBusyTime – I did this by subtracting the sum of the Time Owner Idle column from the observed time. Now that I had OwnerBusyTime, I could divide by the observed time and multiply by 100 to get the percentage of time the owner was busy. Lastly, to calculate response time, or the average time in the system per customer, I summed the Time in Store column and subtracted it by the sum of the Time Owner Idle column to get the StoreBusyTime. Finally, I divided this by 105, or the number of customers.

5.2 Exercise 8.3



```

MATLAB Function
1 function [ArrivalClock, BeginCheckoutClock, QueueTime, EndShoppingClock, TimeInStore, TimeOwnerIdle] = ...
2   CalculateTimes(NewCustomerArrivalTime, NewCustomerShoppingTime, NewCustomerCheckoutTime, OldArrivalTime, OldShoppingTime, OldCheckoutTime, OldArrivalClock, OldEndShoppingClock)
3
4
5   ArrivalClock = NewCustomerArrivalTime + OldArrivalTime;
6
7   BeginCheckoutClock = ArrivalClock + NewCustomerShoppingTime;
8
9   QueueTime = max(0, OldEndShoppingClock - BeginCheckoutClock);
10
11   EndShoppingClock = NewCustomerCheckoutTime + QueueTime + BeginCheckoutClock;
12
13   TimeInStore = EndShoppingClock - ArrivalClock;
14
15   TimeOwnerIdle = max(0, BeginCheckoutClock - OldEndShoppingClock);
16
17
18
19
20 end

```

Figure 5.4: Screenshot of code for the calcTime code block.



```

MATLAB Function
1 function [Throughput, Utilization, ResponseTime, TotalTimeInStore, TotalTimeIdle, TotalEndShoppingClock] = PerformanceMetrics(CustomerNumber, EndShoppingClock, TimeInStore, TimeOwnerIdle, PrevTotalTimeInStore, PrevTotalTimeIdle, PrevTotalEndShoppingClock)
2
3   TotalTimeInStore = PrevTotalTimeInStore + TimeInStore;
4
5   TotalTimeIdle = PrevTotalTimeIdle + TimeOwnerIdle;
6
7   TotalEndShoppingClock = PrevTotalEndShoppingClock + EndShoppingClock;
8
9   FirstArrivalTime = EndShoppingClock - TimeInStore;
10
11   Throughput = CustomerNumber / (TotalEndShoppingClock - FirstArrivalTime);
12
13   TotalBusyTime = TotalTimeInStore;
14   Utilization = TotalBusyTime / (TotalBusyTime + TotalTimeIdle);
15
16   ResponseTime = TimeInStore / CustomerNumber;
17
18 end

```

Figure 5.5: Screenshot of code for the performance metrics code block.

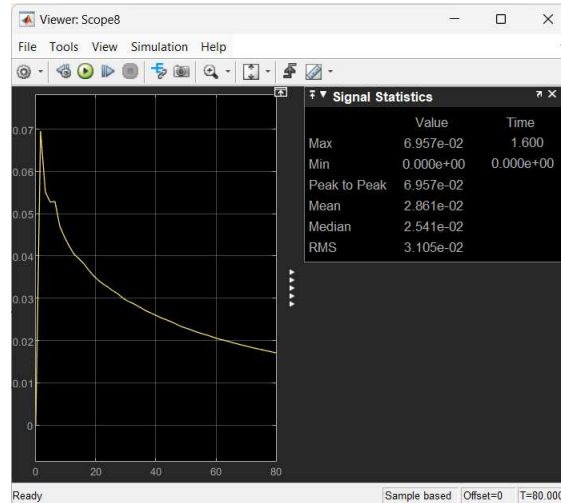


Figure 5.6: Screenshot of scope for throughput.

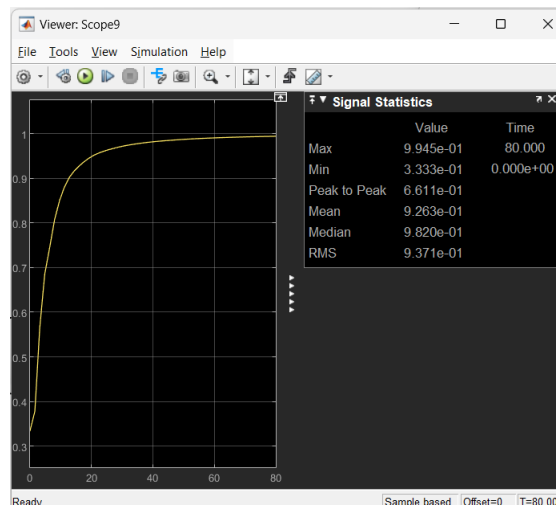


Figure 5.7: Screenshot of scope for utilization.

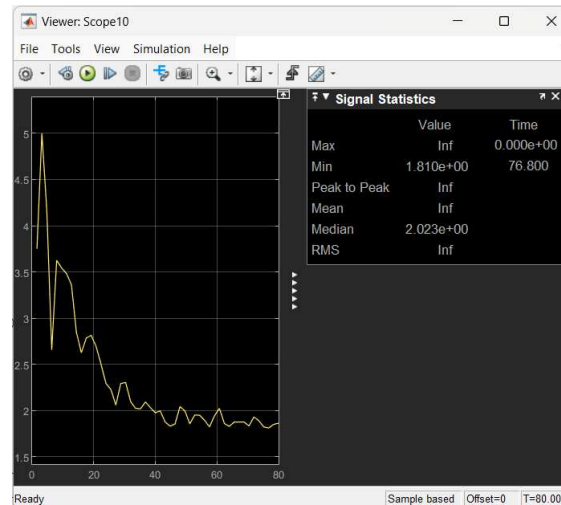


Figure 5.8: Screenshot of scope for response time.

The Simulink Model does not match the Excel very well. This is because I could not figure out how to calculate the observed time without the arrival time of the first customer. I don't know how to perform that task in Simulink and it greatly affected my results as that value should stay the same as we step through the simulation. However, ours changes with each step so there is inaccurate data for all the outputs.

Chapter 6

Abstract Factory

– Jason McCauley and Aidan Nestor

6.1 Exercise 10.1

```
1 public class MyPizzaApp {
2     public static void main(String[] args){
3         if(args.length != 2){
4             System.out.println("Please enter the following: MyPizzaApp <
franchise> <pizzaKind>");
5             System.exit(1);
6         }
7
8         String franchise = args[0];
9         String pizzaKind = args[1];
10        PizzaStore store = null;
11        if(franchise.equals("NYPizzaStore")){
12            store = new NYPizzaStore();
13        }
14        else if (franchise.equals("ChicagoPizzaStore")){
15            store = new ChicagoPizzaStore();
16        }
17        else{
18            System.out.println("Unknown franchise: " + franchise);
19            System.exit(1);
20        }
21
22        Pizza pizza = store.orderPizza(pizzaKind);
23        System.out.println("\nHere's your pizza:\n" + pizza.toString());
24    }
25 }
```

Listing 6.1: Driver code MyPizzaApp

```
1 public class ChicagoPizzaStore extends PizzaStore {
2     protected Pizza createPizza(String item){
3         Pizza pizza = null;
4         PizzaIngredientFactory ingredientFactory = new
ChicagoPizzaIngredientFactory();
```

```
5
6     if(item.equals("cheese")) {
7         pizza = new CheesePizza(ingredientFactory);
8         pizza.setName("Chicago Style Cheese Pizza");
9     }
10    else if(item.equals("veggie")){
11        pizza = new VeggiePizza(ingredientFactory);
12        pizza.setName("Chicago Style Veggie Pizza");
13    }
14    else if(item.equals("clam")){
15        pizza = new ClamPizza(ingredientFactory);
16        pizza.setName("Chicago Style Clam Pizza");
17    }
18    else if(item.equals("pepperoni")){
19        pizza = new PepperoniPizza(ingredientFactory);
20        pizza.setName("Chicago Style Pepperoni Pizza");
21    }
22
23    return pizza;
24 }
25 }
```

Listing 6.2: Concrete class ChicagoPizzaStore

```
1 public interface Sauce {
2     public String toString();
3 }
```

Listing 6.3: Abstract class Sauce

```
1 public interface Cheese {
2     public String toString();
3 }
```

Listing 6.4: Abstract class Cheese

```
1 public interface Veggies {
2     public String toString();
3 }
```

Listing 6.5: Abstract class Veggies

```
1 public interface Pepperoni {
2     public String toString();
3 }
```

Listing 6.6: Abstract class Pepperoni

```
1 public interface Clams {
2     public String toString();
3 }
```

Listing 6.7: Abstract class Clams

```
1 public class PlumTomatoeSauce implements Sauce {  
2     public String toString() {  
3         return "Plum Tomato Sauce";  
4     }  
5 }
```

Listing 6.8: Concrete NY sauce class PlumTomatoeSauce

```
1 public class MarinaraSauce implements Sauce {  
2     public String toString() {  
3         return "Marinara Sauce";  
4     }  
5 }
```

Listing 6.9: Concrete Chicago sauce class MarinaraSauce

```
1 public class MozzarellaCheese implements Cheese {  
2     public String toString() {  
3         return "Mozzarella Cheese";  
4     }  
5 }
```

Listing 6.10: Concrete NY cheese class MozzarellaCheese

```
1 public class ReggianoCheese implements Cheese {  
2     public String toString() {  
3         return "Reggiano Cheese";  
4     }  
5 }
```

Listing 6.11: Concrete Chicago cheese class ReggianoCheese

```
1 public class Garlic implements Veggies {  
2     public String toString() {  
3         return "Garlic";  
4     }  
5 }
```

Listing 6.12: Concrete veggies class Garlic

```
1 public class Onion implements Veggies {  
2     public String toString() {  
3         return "Onion";  
4     }  
5 }
```

Listing 6.13: Concrete veggies class Onion

```
1 public class Mushroom implements Veggies {  
2     public String toString() {  
3         return "Mushrooms";  
4     }  
5 }
```

```
5 }
```

Listing 6.14: Concrete veggies class Mushroom

```
1 public class RedPepper implements Veggies {  
2     public String toString() {  
3         return "Red Pepper";  
4     }  
5 }
```

Listing 6.15: Concrete veggies class RedPepper

```
1 public class SlicedPepperoni implements Pepperoni {  
2     public String toString() {  
3         return "Sliced Pepperoni";  
4     }  
5 }
```

Listing 6.16: Concrete NY and Chicago pepperoni class

```
1 public class FrozenClams implements Clams {  
2     public String toString() {  
3         return "Frozen Clams from Chesapeake Bay";  
4     }  
5 }
```

Listing 6.17: Concrete NY clam class

```
1 public class FreshClams implements Clams {  
2     public String toString() {  
3         return "Fresh Clams from Long Island Sound";  
4     }  
5 }
```

Listing 6.18: Concrete Chicago clam class

```
1 public class ChicagoPizzaIngredientFactory implements PizzaIngredientFactory {  
2     public Dough createDough() {  
3         return new ThickCrustDough();  
4     }  
5  
6     public Sauce createSauce() {  
7         return new PlumTomatoeSauce();  
8     }  
9  
10    public Cheese createCheese() {  
11        return new MozzarellaCheese();  
12    }  
13  
14    public Veggies[] createVeggies() {  
15        Veggies veggies[] = {new Garlic(), new Onion(), new Mushroom(), new  
16        RedPepper()};  
17        return veggies;  
18    }
```



```
17     }
18
19     public Pepperoni createPepperoni() {
20         return new SlicedPepperoni();
21     }
22
23     public Clams createClam() {
24         return new FrozenClams();
25     }
26 }
```

Listing 6.19: Concrete ChicagoPizzaIngredientFactory

```
1 public class PepperoniPizza extends Pizza {
2     PizzaIngredientFactory ingredientFactory;
3
4     public PepperoniPizza(PizzaIngredientFactory ingredientFactory) {
5         this.ingredientFactory = ingredientFactory;
6     }
7
8     void prepare() {
9         System.out.println("Preparing " + name);
10        dough = ingredientFactory.createDough();
11        sauce = ingredientFactory.createSauce();
12        cheese = ingredientFactory.createCheese();
13        veggies = ingredientFactory.createVeggies();
14        pepperoni = ingredientFactory.createPepperoni();
15    }
16 }
```

Listing 6.20: Concrete pizza PepperoniPizza

```
1 public class ClamPizza extends Pizza {
2     PizzaIngredientFactory ingredientFactory;
3
4     public ClamPizza(PizzaIngredientFactory ingredientFactory) {
5         this.ingredientFactory = ingredientFactory;
6     }
7
8     void prepare() {
9         System.out.println("Preparing " + name);
10        dough = ingredientFactory.createDough();
11        sauce = ingredientFactory.createSauce();
12        cheese = ingredientFactory.createCheese();
13        clam = ingredientFactory.createClam();
14    }
15 }
```

Listing 6.21: Concrete pizza ClamPizza

```
1 public class VeggiePizza extends Pizza {
2     PizzaIngredientFactory ingredientFactory;
3 }
```

```

4 public VeggiePizza(PizzaIngredientFactory ingredientFactory){
5     this.ingredientFactory = ingredientFactory;
6 }
7
8 void prepare () {
9     System.out.println("Preparing " + name);
10    dough = ingredientFactory.createDough();
11    sauce = ingredientFactory.createSauce();
12    cheese = ingredientFactory.createCheese();
13    veggies = ingredientFactory.createVeggies();
14 }
15 }

```

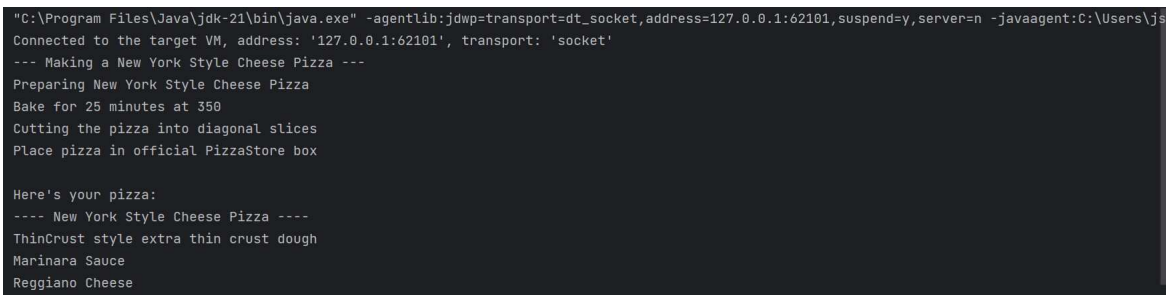
Listing 6.22: Concrete pizza VeggiePizza

```

1 public class ThinCrustDough implements Dough{
2     public String toString(){
3         return "ThinCrust style extra thin crust dough";
4     }
5 }

```

Listing 6.23: Concrete implementation of ThinCrustDough



```

"C:\Program Files\Java\jdk-21\bin\java.exe" -agentlib:jdwp=transport=dt_socket,address=127.0.0.1:62101,suspend=y,server=n -javaagent:C:\Users\js
Connected to the target VM, address: '127.0.0.1:62101', transport: 'socket'
--- Making a New York Style Cheese Pizza ---
Preparing New York Style Cheese Pizza
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box

Here's your pizza:
---- New York Style Cheese Pizza ----
ThinCrust style extra thin crust dough
Marinara Sauce
Reggiano Cheese

```

Figure 6.1: Screen-shot of the output when ordering a cheese pizza from the NY store.



```

"C:\Program Files\Java\jdk-21\bin\java.exe" -agentlib:jdwp=transport=dt_socket,address=127.0.0.1:62145,suspend=y,server=n -javaagent:C:\Users\js
Connected to the target VM, address: '127.0.0.1:62145', transport: 'socket'
--- Making a Chicago Style Veggie Pizza ---
Preparing Chicago Style Veggie Pizza
Bake for 25 minutes at 350
Cutting the pizza into diagonal slices
Place pizza in official PizzaStore box

Here's your pizza:
---- Chicago Style Veggie Pizza ----
ThickCrust style extra thick crust dough
Plum Tomato Sauce
Mozzarella Cheese
Garlic, Onion, Mushrooms, Red Pepper

```

Figure 6.2: Screen-shot of the output when ordering a veggie pizza from the Chicago store.

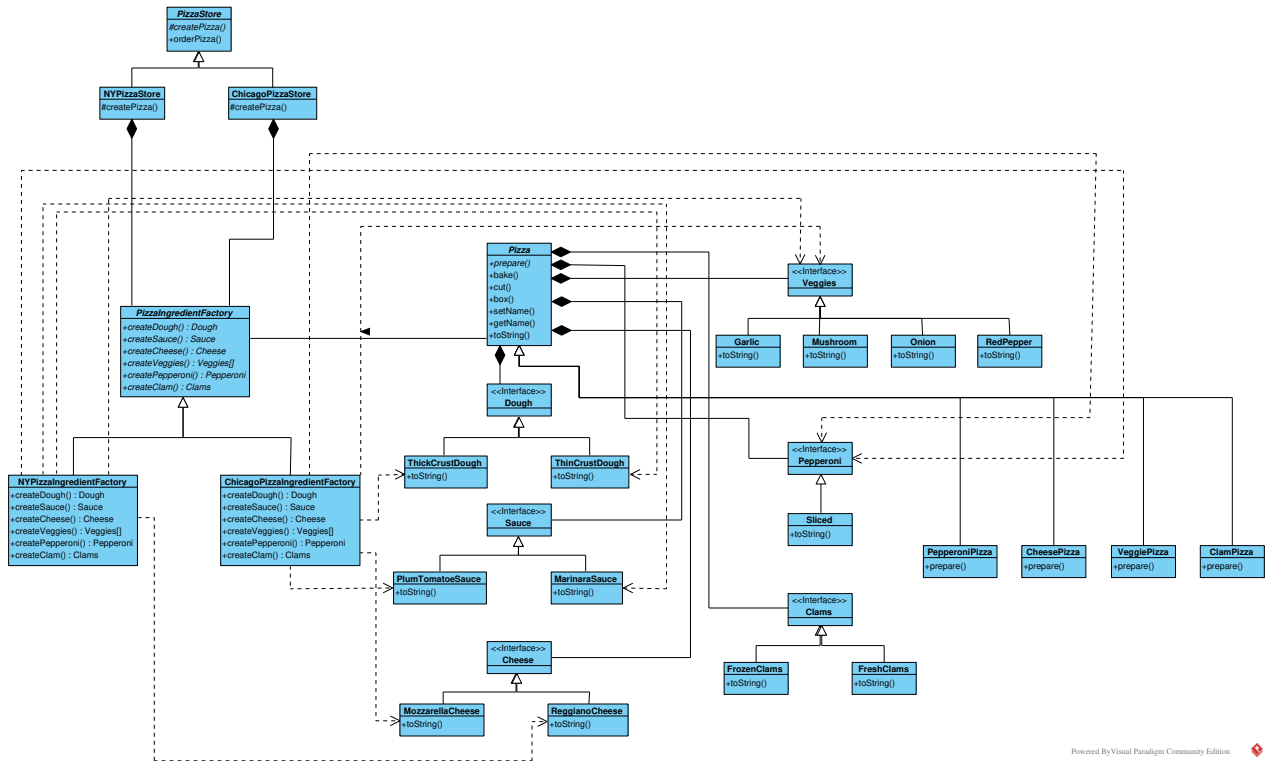


Figure 6.3: Updated UML Class Diagram of the Abstract Factory.

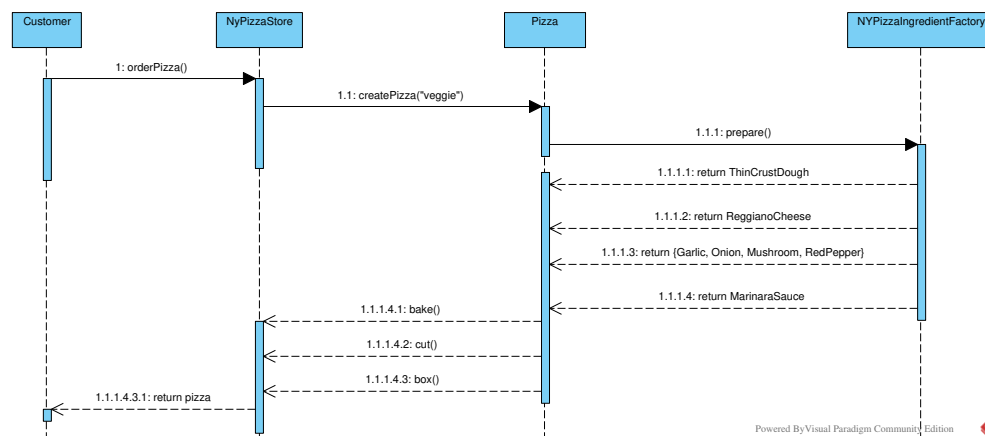


Figure 6.4: NY Veggie Pizza Sequence Diagram.