

Riley Wilkins

ICSI 409

Dr. Carol Cusano

Final Project

Vending Machine Simulator

Introduction:

For this final project, I designed a simulator for a vending machine. I chose to simulate a vending machine because it can be classified as a finite state machine, an important topic in the theory of computation. A finite state machine (FSM) or finite state automaton (FSA) is defined as “a mathematical model of computation based on the ideas of a system changing state due to inputs supplied to it, with some of these states being acceptor or final states” (Zeil 1). The formal definition of a finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite called the alphabet, δ is a transition function $(Q \times \Sigma \rightarrow Q)$, q_0 is the start state, and F is the set of accept states (Sipser 35).

Finite state machines fall into two broad categories: machines with output and those without output. Within the “FSMs with output” category are even more specific types of machines, such as Moore and Mealy machines. FSMs without output fall into the categories of Deterministic Finite Automata and Non-deterministic Finite Automata. Vending machines produce an output, so they are best modelled as either a Moore or Mealy machine. A Moore machine’s distinguishing characteristic is that the outputs depend only on the current state, whereas with a Mealy machine, outputs depend on both the current state and the input (Sequence Logic 1-12).

Although, finite state machines may seem quite abstract and academic, there are many real-world applications and most of us interact with them on a daily basis. A few of the many applications of finite state machines include: traffic lights, text parsing, controllers for CPUs, and natural language analyzers (Gribkoff 3). What all these systems have in common is a set of states, inputs that can be applied, transitions to other states, and final or accept states. Even though these applications seem to be unrelated, they all share these fundamental traits and behavior.

With regard to a vending machine, the alphabet, or inputs can be thought of as the set of payments that one can insert into a machine. Traditionally, a vending machine would accept nickels, dimes, quarters, one dollar, and five dollar bills. Although newer models accept credit cards and even mobile payments, I've ignored these and kept to the original nature of a vending machine. Furthermore, another part of the alphabet is the "item code". Typically each item in a vending machine has a label or code consisting of numbers, letters, or both, which uniquely identify each drink, snack or other product. After payment is inserted, the user inputs this code to execute the transaction. Now that the concept of a finite state machine, its importance and relevance to real-world applications has been established, it's time to address the goals of this project.

My goal for this project was to create an interactive vending machine simulator. It was initially designed and written in Java and there was no user interface; only a controller, a couple classes representing "snack", "vending machine", and "customer" with appropriate methods and a driver class to test the functionality. However, the end result seemed a bit dull, so it was redesigned as a full-stack application and written in JavaScript. The end result was an interface

that resembles a real-world vending machine, with functionality that mimics the behavior of an actual vending machine. There are twelve snacks available for purchase, each has a price and an item code to uniquely identify that snack. Users are able to insert payment by clicking on any combination of the nickel, dime, quarter, one dollar or five dollar bills and then make a selection by typing in the code of the snack they wish to purchase. In the following sections, I'll provide more clarity and detail in regard to the design and implementation.

Methodology/Implementation:

The methodology followed when designing and implementing this project was to start at the most fundamental level of what a vending machine is by creating a state diagram (appendix 10) and a flow diagram (appendix 11). Creating these diagrams first gave me a much better understanding of how to approach the problem of modelling the vending machine in code. The flow diagram in particular was very helpful in understanding what functions/methods needed to be written and the various test cases to create to make sure the machine was behaving as expected.

As previously stated in the introduction, the vending machine was first designed, created and tested using Java as the programming language and Eclipse for Java Developers as the integrated development environment. The classes were 'Vending Machine', 'Snack', 'Customer', and a controller/driver class to test the functionality. Some fields for the 'Snack' class were the name of the snack and its price, while the 'Vending Machine' class had fields such as location and an arraylist of 'Snack' objects. Inside the driver class, I simulated a customer ordering their desired snack from the machine, having them input cash and the correct code, and

then the snack would “dispense”. Although it worked properly, the end result was a bit bland, so it was redesigned using a different set of technologies to make the simulation more interactive.

The final design of the vending machine was a full-stack application using a number of different tools and technologies. The four main technologies involved are Node.js, Express.js, Firebase, and embedded JavaScript. The integrated development environment also changed from Eclipse to Visual Studio Code. Node is JavaScript runtime environment, which allows JavaScript to be executed outside of a web browser. This is useful because it allows developers to write the back-end of an application in JavaScript, instead of the language being limited to being executed in the browser. Express is a small web-framework with helpful built-in features, which can make routes easy to configure. Firebase, a database supported by Google, was kept to limited use (only one table ‘Snack’) to keep consistent with the fact that finite state machines are the most basic computers with limited memory. Using these technologies, a basic application with user interface was created to simulate the vending machine.

There were a number of design considerations that had to be made. One of these was deciding whether to dispense a user’s change automatically upon purchasing a snack or whether to keep the remaining balance and give them the choice as to whether to purchase another snack or dispense their change. I ultimately chose to go with the second option, keeping the remaining balance and enabling users to choose whether to make another purchase, insert more funds to add to their remaining balance, or dispense the balance. Both types of vending machines exist, and it made the design more interactive, which is why this decision was made. Another design consideration was whether users should insert payment first and then make a snack selection or the reverse. The final design choice was that the order of payment or snack choice is irrelevant;

users can choose to insert payment or item code in whichever order they prefer. A third consideration was how to design the insertion of payment; one option was to create a basic form where a user enters an amount that conforms to an acceptable payment amount (i.e. \$1.75), but it seemed more interactive to create buttons out of images of a nickel, dime, quarter, one dollar, and five dollar bills. When the user clicks a coin or dollar image, a field updates with the appropriate amount. They can click the coins and dollars in any combination to input their payment, and the payment field updates with the appropriate total.

Testing/Samples:

The application was tested hundreds of times throughout development to fix bugs and ensure the user interface and functionality worked as expected. One bug that was particularly difficult to solve occurred when inputting the payment. The issue could be seen when clicking the dime and then the nickel might produce a result such as “\$0.1499999998” in the payment field instead of the desired “\$0.15”. While researching the issue, I came across a possible solution using a built-in JavaScript method “.toFixed(2)”, which worked perfectly. A second bug occurred when calculating a user’s change. The issue was not understanding how the division and modulus operators work in JavaScript; I was using them as one would when coding in Java. Ultimately, these were just bugs that caused some temporary frustration, and once solved it was possible to test the four main cases of the application:

Test Case 1 (appendix 6): (Amount Entered \geq Item Price AND Item Quantity > 0)

This can be thought of as the “Accept purchase, and proceed with transaction” case. When a user enters a payment greater than or equal to the price of the snack, and the snack’s quantity is greater than 0, then accept the purchase, dispense the snack, and calculate the remaining balance.

Test Case 2 (appendix 7): (Amount Entered < Price AND Item Quantity > 0)

Also known as the “Please enter more money” case, the user has entered insufficient funds. They must either enter more cash, make a different selection, or dispense their balance.

Test Case 3 (appendix 8): (Item Quantity == 0)

This is the “Item Out Of Stock” case. In this situation, the item needs to be restocked, so the user must either choose a different snack or dispense any funds they’ve entered.

Test Case 4 (appendix 9): (Incorrect Item Code entered)

The last test case occurs when the user enters an incorrect item code. There are twelve unique codes that identify the twelve snacks in the machine, and if the user enters a code that doesn’t match one, they are given an error message. They can then re-enter a correct code or dispense any cash they’ve entered.

Result Analysis:

After much iteration and testing, all four test cases are handled without issue. All of these scenarios are checked when a user clicks to purchase an item, and the appropriate output is displayed. Ensuring that the vending machine could handle all the test cases was fairly straightforward to implement. The “makePurchase()” function, which executes when the user clicks to make a purchase, checks these conditions or calls other methods, which check the cases previously described. In summary, the project’s goals are met, and all cases are accounted for. Later, in the conclusion section of this report, possible improvements and additional test cases are discussed.

User Guide:

To use this application, a user should download the Vending Machine (see appendix) repo on my GitHub to their computer, unzip the folder, open the folder in Visual Studio Code (preferably). Next, create a new Terminal (access command line) inside Visual Studio Code. Then the user must install Node.js, Express.js, and npm (see references). Next, change directory into the functions folder using 'cd functions'. Then use npm to install the remaining dependencies by typing 'npm install' (this step might take a few minutes or so). Now, typing the command 'node index.js' and hitting enter will start the server. If the server is working properly, you will see the message "index.js is listening on port: 3000" appear after a few seconds. The last step is to open Google Chrome web browser (preferred) and type in 'localhost:3000'. Hit enter, and the vending machine application will appear and be ready to use.

Once the application has loaded, click on any combination of coins and bills to enter your payment, and type in the item code associated with the snack you wish to purchase. Click either the 'Purchase' or 'Dispense Change' button to either purchase the snack or receive your payment back. If you purchase a snack, your remaining balance will appear in the form for you to either make another purchase, input more cash, or dispense your change. Then you can click on the quantity buttons associated with each snack to find out how many of each snack is left in the machine. If the quantity is zero, you must choose a different snack because the item needs to be restocked.

Conclusion:

The project's goal was to simulate a vending machine, one of the many real-world applications of finite state machines, which are the simplest of computers. The initial design and implementation in Java was a success, but it seemed more appropriate when simulating a

vending machine to try and make it as interactive as possible. In the final design, this goal was achieved with the development of a full-stack application, designed from a user's perspective, that simulates the appearance and behavior of an actual vending machine. The test cases, developed during the creation of the state diagram and flowchart, were all tested and succeeded with their intended behavior or function.

To recap, these four test cases were: When a user enters sufficient funds and the quantity of a snack is greater than zero, to accept the transaction and dispense the snack. The second test case was when a user has entered insufficient funds for their chosen snack with quantity greater than zero; this is when a message displays asking the user to insert more funds to complete the purchase. The third case occurs when the quantity of a snack is equal to zero; the item is out of stock and unable to be purchased, and the user must choose a different snack or dispense their change. Finally, the fourth case occurs when the input string for the item code is incorrect; the user must enter a correct code or dispense their change.

Although the project overall was a success, there is room for improvement. In the future, I would like to implement the functionality for an additional test case, which is when a vending machine has run out of change to dispense. This situation occurs when a customer attempts to make a purchase, but the machine dispenses either no change or too little change. In this situation it would be ideal for the machine to reject all input, but at the very least, the machine must dispense the entire payment the user has input, both coins and bills.

References:

EJS. (2019). Retrieved from <https://ejs.co/>

Firebase Google. (2019). Retrieved from <https://firebase.google.com/>

Foundation, Node. (2019). Retrieved from <https://nodejs.org/en/>

Gribkoff, E. (2013, May 29). Finite State Machine Applications. Retrieved from <https://web.cs.ucdavis.edu/~rogaway/classes/120/spring13/eric-applications>

Node.js web application framework. (2019). Retrieved from <https://expressjs.com/>

Npm. (2019). Retrieved from <https://www.npmjs.com/>

Otto, M., & Thornton, J. (2019). Bootstrap. Retrieved from <https://getbootstrap.com/>

Sequence Logic. (2005, August). Retrieved from <http://www-inst.eecs.berkeley.edu/~cs150/fa05/Lectures/07-SeqLogicIIIx2.pdf>

Sipser, M. (2012). *Introduction to the theory of computation*(3rd ed.). Cengage Learning.

Stock Images, Royalty-Free Illustrations, Vectors, & Stock Video Clips - iStock. (2019). Retrieved from <https://www.istockphoto.com/>

Visual Studio IDE, Code Editor, Azure DevOps, & App Center. (2019). Retrieved from <https://visualstudio.microsoft.com/>

Zeil, S. (2019, February). Finite State Automata. Retrieved from
<https://www.cs.odu.edu/~zeil/cs390/latest/Public/fsa/index.html>

Appendix:

1) Code: (too long to include) see GitHub repository.

2) Carolcusano's invite link to GitHub repository:

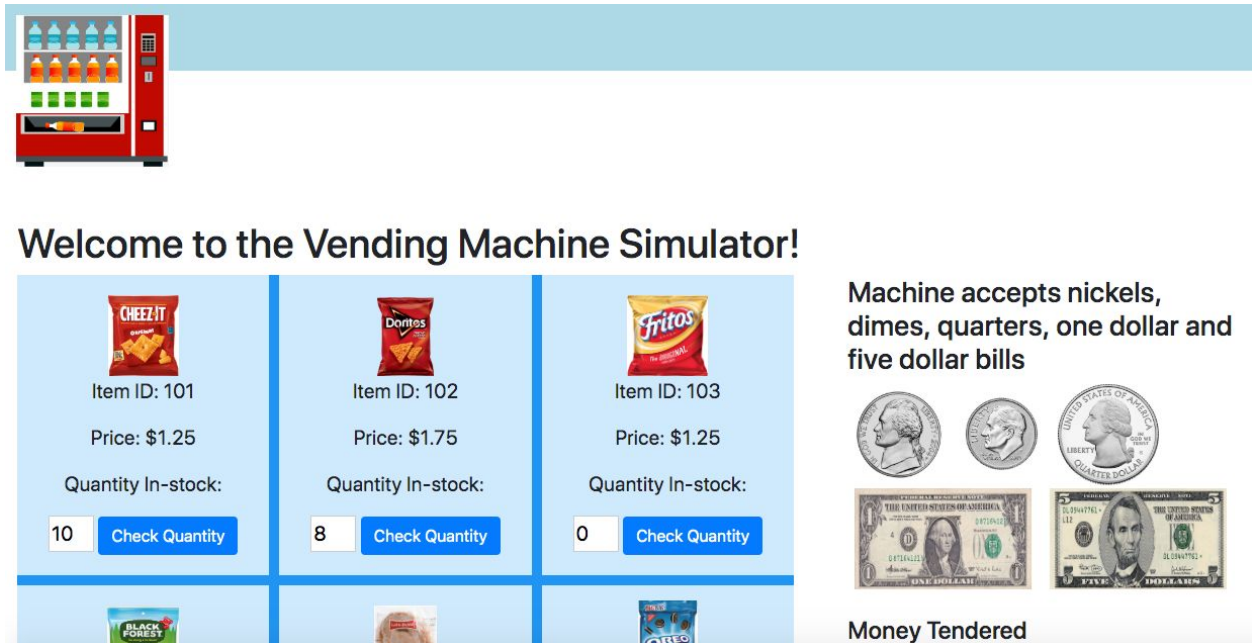
<https://github.com/rwilk19/VendingMachine/invitations>

3) Screencast (video of the application and test cases):







<https://www.screencast.com/t/Uo2AE5eL>



4) Screenshots of the User Interface:

4a)



4b)

<input type="text"/> Check Quantity	<input type="text"/> Check Quantity	<input type="text"/> Check Quantity
 Item ID: 104 Price: \$2.75 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 105 Price: \$2.25 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 106 Price: \$1.75 Quantity In-stock: <input type="text"/> Check Quantity
 Item ID: 107 Price: \$2.00 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 108 Price: \$1.25 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 109 Price: \$0.75 Quantity In-stock: <input type="text"/> Check Quantity



Money Tendered

Enter item code

Purchase







Or, dispense your change

Dispense Change

Service Machine

Re-stock Snack:

4c)

 Item ID: 107 Price: \$2.00 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 108 Price: \$1.25 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 109 Price: \$0.75 Quantity In-stock: <input type="text"/> Check Quantity
 Item ID: 110 Price: \$1.25 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 111 Price: \$1.75 Quantity In-stock: <input type="text"/> Check Quantity	 Item ID: 112 Price: \$0.75 Quantity In-stock: <input type="text"/> Check Quantity

Dispense Change

Service Machine

Re-stock Snack:

Enter Service-Key:

Restock Snack





5) How the Vending Machine application's server is started:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: node
Riley's MacBook Pro:VendingMachine rileyhugheswilkins$ cd functions
Riley's MacBook Pro:functions rileyhugheswilkins$ node index.js

index.js is listening on port: 3000
```

6) Test Case 1 (“Accept purchase and proceed with transaction”):





The user is looking to purchase a honey bun. They have entered \$2.75, which is greater than the price of \$2.25, and the quantity of honey buns in stock is 8, which is greater than 0.

 Item ID: 105 Price: \$2.25 Quantity In-stock: 8 <input type="button" value="Check Quantity"/>	 Item ID: 106 Price: \$1.75 Quantity In-stock: <input type="text"/> <input type="button" value="Check Quantity"/>	Money Tendered 2.75 Enter item code 105 <input type="button" value="Purchase"/> Or, dispense your change <input type="button" value="Dispense Change"/>
 Item ID: 105 Price: \$2.25 Quantity In-stock: 8 <input type="button" value="Check Quantity"/>	 Item ID: 106 Price: \$1.75 Quantity In-stock: <input type="text"/> <input type="button" value="Check Quantity"/>	Money Tendered 0.50 Enter item code item code <input type="button" value="Purchase"/> Or, dispense your change <input type="button" value="Dispense Change"/> <div>Thank you for your purchase of: Honey Bun</div>

The machine thanks the user for their purchase and displays their remaining change, which they can either use to purchase another item or dispense.

7) Test Case 2 (“insufficient funds”):

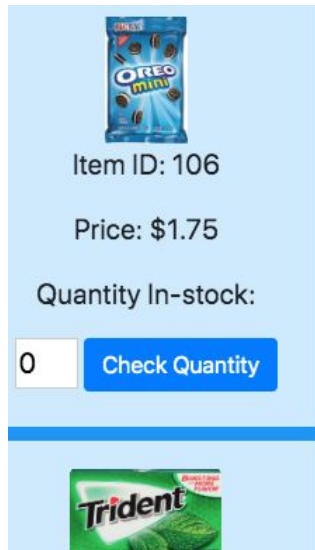
The user is looking to purchase a honey bun. They have entered \$1.75, which is less than the price of \$2.25. The quantity of honey buns in stock is 8, which is greater than 0.

 Item ID: 105 Price: \$2.25 Quantity In-stock: 8 <input type="button" value="Check Quantity"/>	 Item ID: 106 Price: \$1.75 Quantity In-stock: <input type="text"/> <input type="button" value="Check Quantity"/>	Money Tendered 1.75 Enter item code 105 <input type="button" value="Purchase"/> Or, dispense your change <input type="button" value="Dispense Change"/>
 Item ID: 105 Price: \$2.25 Quantity In-stock: 8 <input type="button" value="Check Quantity"/>	 Item ID: 106 Price: \$1.75 Quantity In-stock: <input type="text"/> <input type="button" value="Check Quantity"/>	Money Tendered 1.75 Enter item code item code <input type="button" value="Purchase"/> Or, dispense your change <input type="button" value="Dispense Change"/> <div>Error, please enter more money! \$1.75 < \$2.25</div>

The machine displays an error message to the user, telling them that they haven't entered enough money to purchase that item.

8) Test Case 3 (“Snack Out Of Stock”):

The user is looking to purchase Oreo’s. They have entered \$2.00, but checking the quantity of Oreo’s shows that there are zero left in the machine.



Money Tendered

2.00

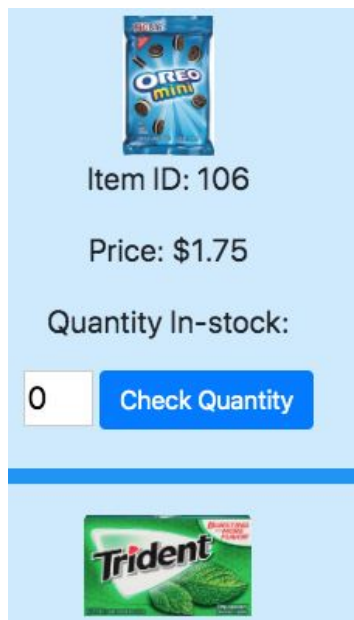
Enter item code

106

Purchase

Or, dispense your change

Dispense Change



Money Tendered

2.00

Enter item code

item code

Purchase

Or, dispense your change

Dispense Change

Item Out Of Stock: Oreo's, Quantity: 0

The machine displays an error message to the user, telling them that Oreo’s are out of stock.

They can now either choose a different item or dispense the cash they entered.

9) Test Case 4 (“Wrong Item Code”):

The user is looking to purchase a snack, but they have entered an invalid item code. They have entered \$2.00, but the item code ‘999’ doesn’t correspond to any snack in the machine.

Money Tendered

2.00

Enter item code

999

Purchase

Or, dispense your change

Dispense Change

Money Tendered

2.00

Enter item code

item code

Purchase

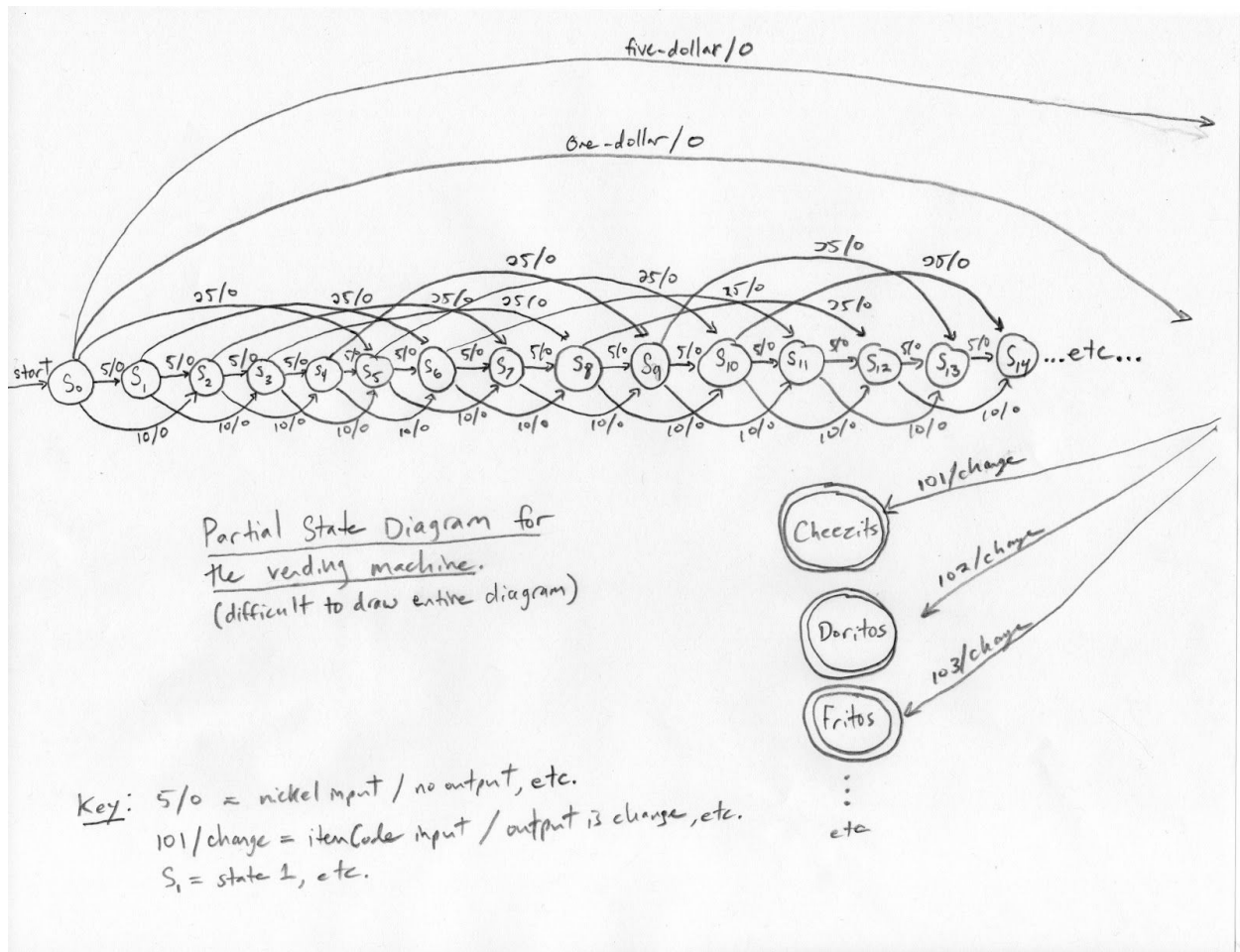
Or, dispense your change

Dispense Change

Error: Invalid Item Code, try again

The machine displays an error to the user telling them that they have entered an invalid item code and to try again.

10) Vending Machine State diagram:



11) Vending Machine Flow diagram:

