# Design Document - VM001 Simulator


This is the Design Document for the Water World Vending Machine simulator - model VM001.

In this document you will find the reasoning and justification for the majority of the decisions that were made in regards to how the project was designed.

**The Water World Vending Machine company employees:**

Sam Rice — Ivan Martinez
Justin Kephart — Kelton Malhotra
Jake Lyon — Kiefer Solberg
Riley Wallace

*The Water World Vending Machine company is a subsidiary of the Moorpark College's Computer Science course, CS M20, Object Oriented Data Structure and Algorithm Detection, and is a completely fictional business.*

# Table of Contents

# Input Files

We decided to use two input files: One for Simulated User Input, and the other for the vending machine's inventory. We decided the inventory file would be a .csv so we could create the inventory in a spreadsheet, and use the commas like delimiters, which made a visual representation of the slots easy for the user to edit the input file, while keeping the raw text data of the file easy to input into a C++ program. Raw data is extracted and a Slot number is generated from some for loops. This information is added to the vending machine via the addSlot() function. The parameters for this function are also designed so the client doesn't have to worry about the internal detail of the vending machine such as Slots or maps. The function itself takes the raw data, creates a Slot, which is coupled to the Slot number and inserted into the map so it can be easily found with simply its slot number. The other input file was for simulated user input, and was typed exactly the way we'd expect a user to enter commands into the machine - each line would represent the user being prompted for input, then hitting enter. We decided to use slightly more complicated user input commands, such as "Swipe 111123" instead of "S", since we were anticipating this particular use of the program to be reading from an input file. In the example below, the left side of the table is the more *simple* input, and the right is the slightly more complicated - you'll notice the slightly more complicated side is MUCH easier to read and understand what's happening, which is why we went with that method.

| | |
|---|---|
| S<br>1234<br>C<br>I<br>1<br>D4<br>D5<br>I<br>.25<br>R | SwipeCard 1234<br>PressButton Cancel<br>InsertCash 1<br>PressButton D4<br>PressButton D5<br>InsertCash .25<br>PressButton CoinReturn |

---

# Output Files

For the output files, we decided to create two separate files, similar to the way the input files work. We took the stock of the machine (all current drink names and amounts) and formatted a .csv file to look exactly like the input file so that you could actually use the output file as input later on, and so it's easier to view. This file is created in a similar fashion to the way the input file is initially read. In order to get the raw data out of the machine, instead of simply returning slot, which is part of the server vending machine file, the retrieval function, getSlot() uses reference parameters to return all the pieces of raw data itself. We also included a .txt output file which is the simulated console output and the simulated console input together. We included the input file contents mixed in with the output so you could actually read the file by itself and understand what was happening.

---

# Credit Card Class

We needed a way to consistently know if a credit card would return true or false for if it was a valid card or not, so we decided the simplest way (albeit not realistic) would be to check if the card number was odd or even. To simplify the process, we call the function isValid(), which just returns mod 2 of the card number, which is another way of saying it returns 1 (true) for odd numbers and 0 (false) for even numbers. It also checks if the account has enough to charge the max price, but to simulate this it just returns true. The card will only be charged the amount of the product however.

---

# Graph Class

The graph class works with the combination of the edge class to implement the FSA's states and transitions in the vending machine class. The graph class stores its states and transitions in a vector. Each state is assigned a value in the vector with a map that contains the transitions that are allowed to made. Each key in that map is the transition with a value to what state that transition leads. The graph class is generic so that it can be used with any data type. We chose the string data type to be used as both the states and transitions. One major benefit of using the graph class as an implementation for the automata is the ability to add or remove states without majorly disrupting its structure. This will allow for a more flexible system and in turn, a more permanent solution.

---

# List of Assumptions

This assumes that the Card reader approves or declines a card once swiped, and bills the card after the user makes a valid selection. Also, this assumes if there are cash AND credit, we use the CREDIT and return the cash. Assumes if the user selects a position that is empty, we do NOT simply dispense from a different position of the same item. Does not allow for the user to swipe a valid card more than once. We assume that any credit card being used has enough money on their account. We can assume coin return will always have enough coins. The machine does not need to keep track of change, and does not need to specify what type of change is being returned - example: returned $0.58, not returned two quarters and a nickel and 3 pennies. If the machine already has enough, we stop accepting cash and stop accepting credit. ***If a credit card is declined for any reason, including but not limited to an invalid card or a card has already been swiped; the machine will not display any message or do anything at all, which is surprisingly how the actual vending machines at the college work.

---

# Vending Machine Class

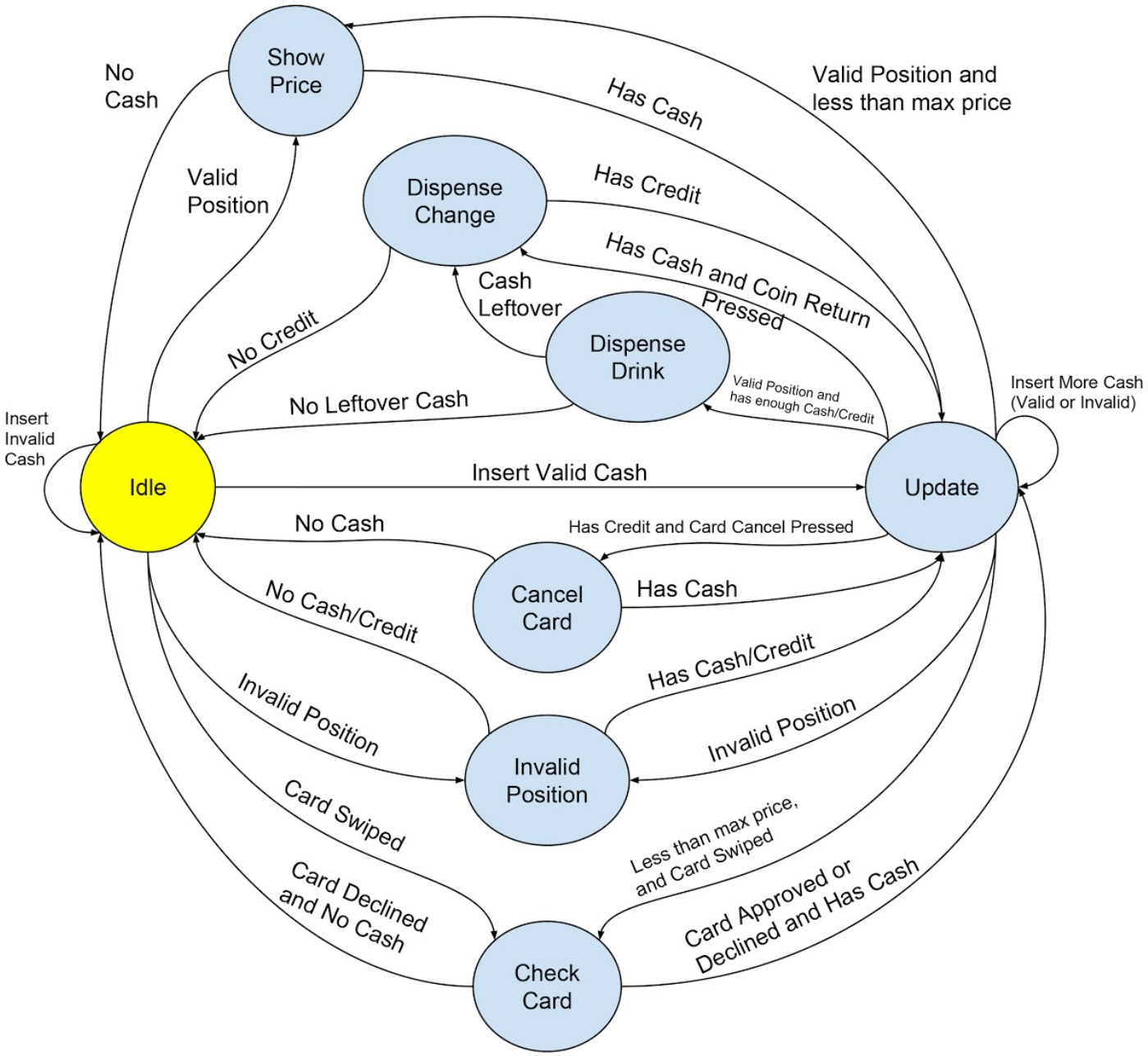
We choose to use the graph class to hold all of our states that the vending machine could use and the edge class to hold all the actions or the requirements that have to be met in order to move to the next state. The reason we choose to use the graph class is because we were told to use it by the client. We also have a struct called Atable which is an array where each item in the array contains a string to hold the current state that we are in and the callback function to the state we are in. That way instead of having a bunch of if else statements to figure out which state we go to next we can have only a few if else statements and use the function goToNextState() which just uses a for loop to to go through the Atable to figure out what state we should be in by comparing the string that holds the current state and then calling the callback function that is associated with the current state. By doing this, our code looks clean and is easy to maintain if we need to go back and add new states we only have to update a few things compared to just having a bunch of if else statements where the code is very hard to read and understand.

---

# Slot Class

In order to store the slot information, we use a Slot class, which is included in the same file as VendingMachine since the two classes are so tightly coupled. This is a very simple data structure that holds the item name, price, and stock of the item slot in the vending machine. Objects of this class are created, coupled, and added to the map via the addSlot() function in VendingMachine. The map allows easy lookup of the proper slot using the paired slot number.

# Finite State Automata (FSA)




**Valid Position:** A position that exists and has an item in it.
**Invalid Position:** Either incorrect input, a position that does not exists, or a position that does not have an item in it.

**Cancel Card**: Card Transaction Canceled, displays message, then goes to next state. Does not return any cash, but removes the max price credit from the total.
**Check Card**: Check Card if valid, displays "Checking Card…" then goes to next state.
**Dispense Drink**: Dispense drink. Shows "Dispensing…" then goes to next state.
**Dispense Change**: Dispense Change, returns the amount of cash in the machine (NOT credit!)
**Invalid Position**: Error Invalid Position, displays warning message, then goes to next state.
**Idle**: Starting Point, machine waits for input and displays Generic Message
**Show Price**: Shows the Price of the selected item. Waits here for set time, then goes to next state.
**Update**: Update Counter, machine displays the current amount of cash/credit available.

FSA
Version: X