MCS4663 Operating Systems - Summer 2022 – Project_Vending_Machine

Devson Butani

06/18/2022

Introduction:

This is a report for the soda(pop) vending machine project. This project gives insight into what it takes to design an operating system command line interpreter by creating a command line operated vending machine program. This includes switching between user and service modes, handling data structures and developing algorithms using object-oriented programming.

Design diagram:

The main driver behind the design of the algorithm was switching between modes and having multiple states that interact with the modes. The two modes, Service mode and User mode, interact with 10 states having limited access to the private inventory data structure inside the vending machine class. Figure 1 below shows how the states can be changed using command line input from user or service person.

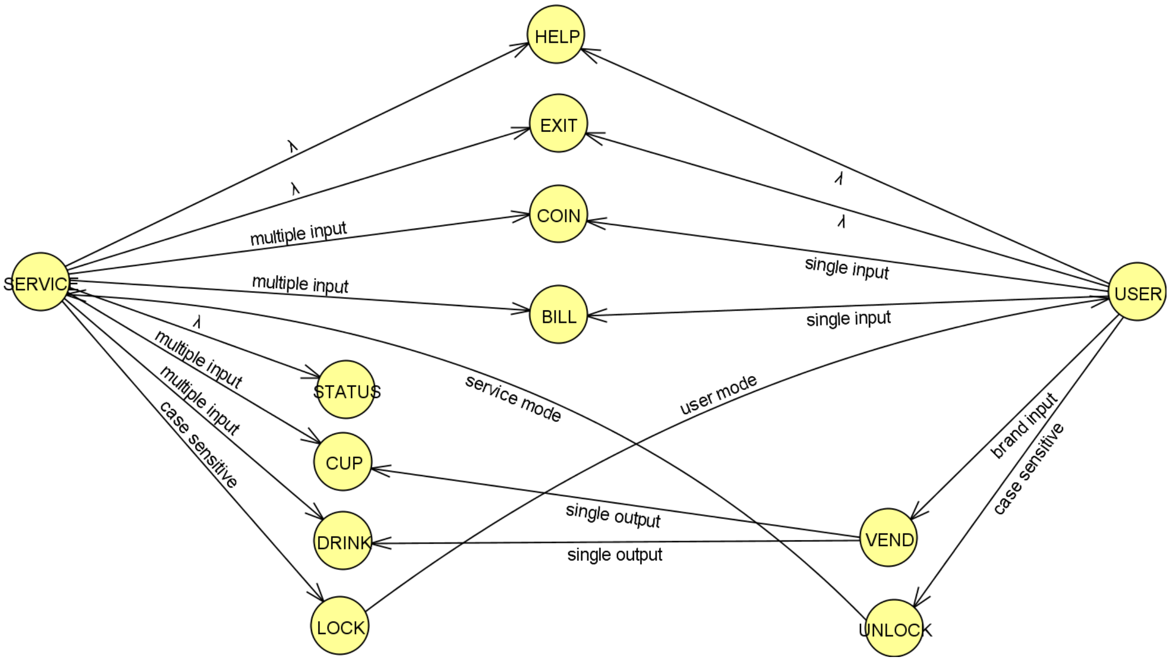


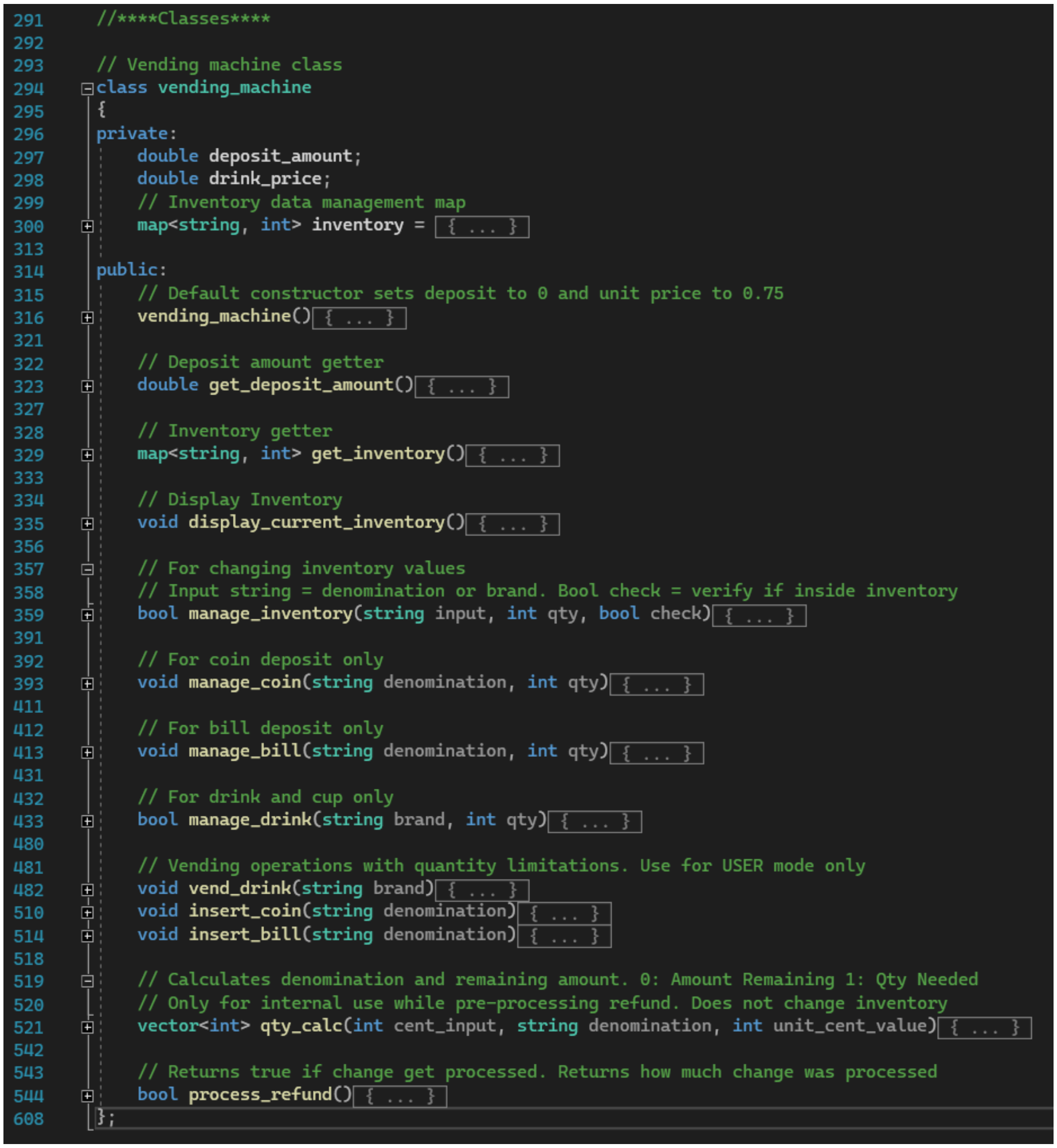
Figure 1: Simplified State Diagram

For example, the DRINK state can be accessed by both user and service modes however, user mode is limited to a single output by going through the VEND state. This allows additional functionality like checking inventory of cups before dispensing the drink. The design was primarily inspired to mimic how a real vending machine might work. Similarly, COIN and BILL states limit the user to a single input whereas the service person can freely alter the entire inventory.

Description of major data structure/algorithm/class:

The major class structure (Figure 2) is representative of an actual vending machine where it manages inventory on its own, disburses change, and handles errors. The data is kept private inside the object and can only accessed from certain states within the command manager switch (Figure 3, next page). This type of object-oriented programming allows running of multiple vending machines with their own inventories, with individual brands and item prices, all at the same time.

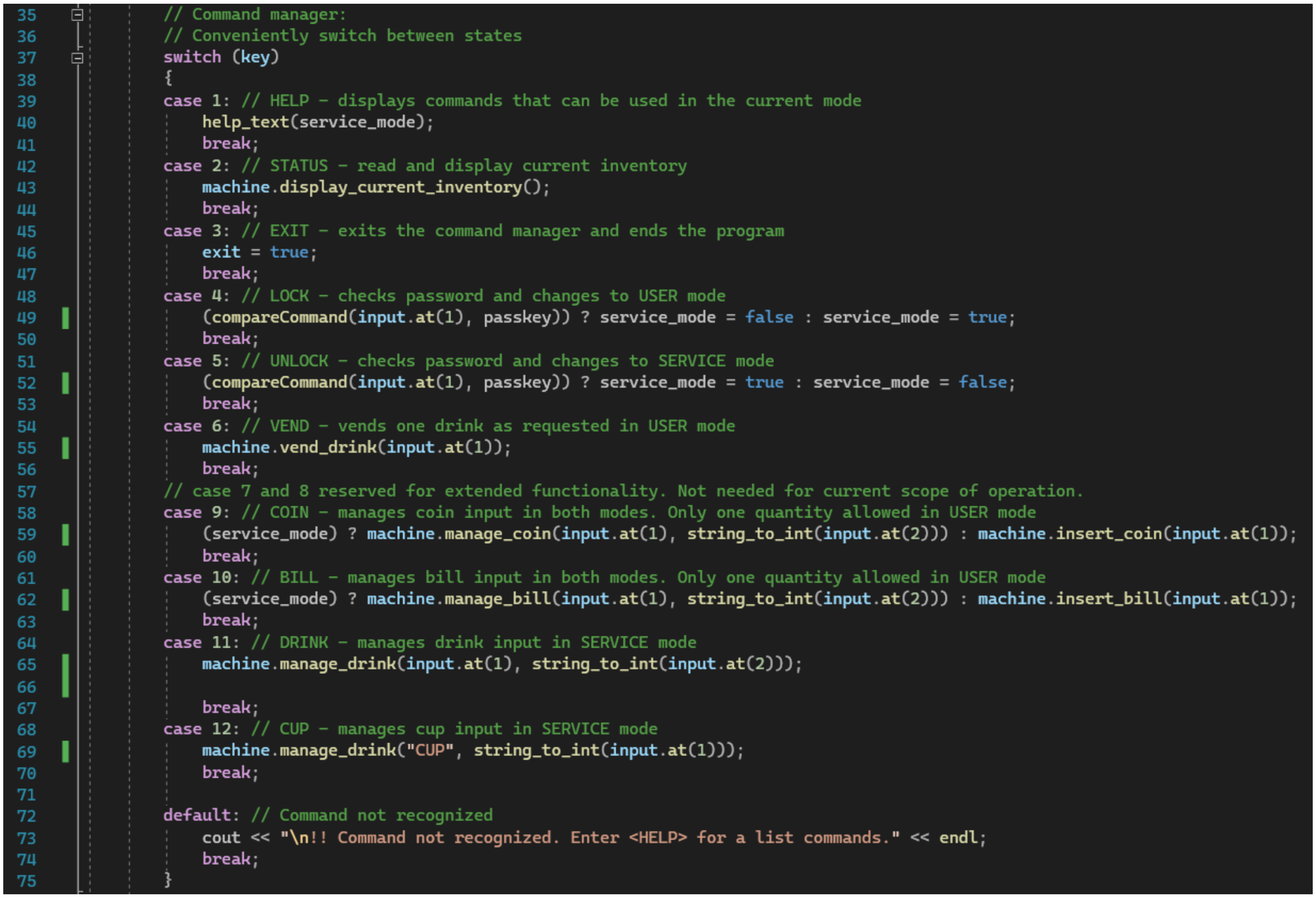
Moreover, user and service modes are separated by allocating them individual function calls to prevent writing errors, overrides, and state mismatch. For example, the single input accepted into “insert_coin” for user mode whereas “manage_coin” can accept multiple or even negative inputs (inventory has zero as lower bound). This allows the program to catch errors with quantities or correct commands in incorrect states.



```
//****Classes****

// Vending machine class
class vending_machine
{
private:
    double deposit_amount;
    double drink_price;
    // Inventory data management map
    map<string, int> inventory = { ... }

public:
    // Default constructor sets deposit to 0 and unit price to 0.75
    vending_machine(){ ... }

    // Deposit amount getter
    double get_deposit_amount(){ ... }

    // Inventory getter
    map<string, int> get_inventory(){ ... }

    // Display Inventory
    void display_current_inventory(){ ... }

    // For changing inventory values
    // Input string = denomination or brand. Bool check = verify if inside inventory
    bool manage_inventory(string input, int qty, bool check){ ... }

    // For coin deposit only
    void manage_coin(string denomination, int qty){ ... }

    // For bill deposit only
    void manage_bill(string denomination, int qty){ ... }

    // For drink and cup only
    bool manage_drink(string brand, int qty){ ... }

    // Vending operations with quantity limitations. Use for USER mode only
    void vend_drink(string brand){ ... }
    void insert_coin(string denomination){ ... }
    void insert_bill(string denomination){ ... }

    // Calculates denomination and remaining amount. 0: Amount Remaining 1: Qty Needed
    // Only for internal use while pre-processing refund. Does not change inventory
    vector<int> qty_calc(int cent_input, string denomination, int unit_cent_value){ ... }

    // Returns true if change get processed. Returns how much change was processed
    bool process_refund(){ ... }
};
```

Figure 2: Vending Machine Class Structure

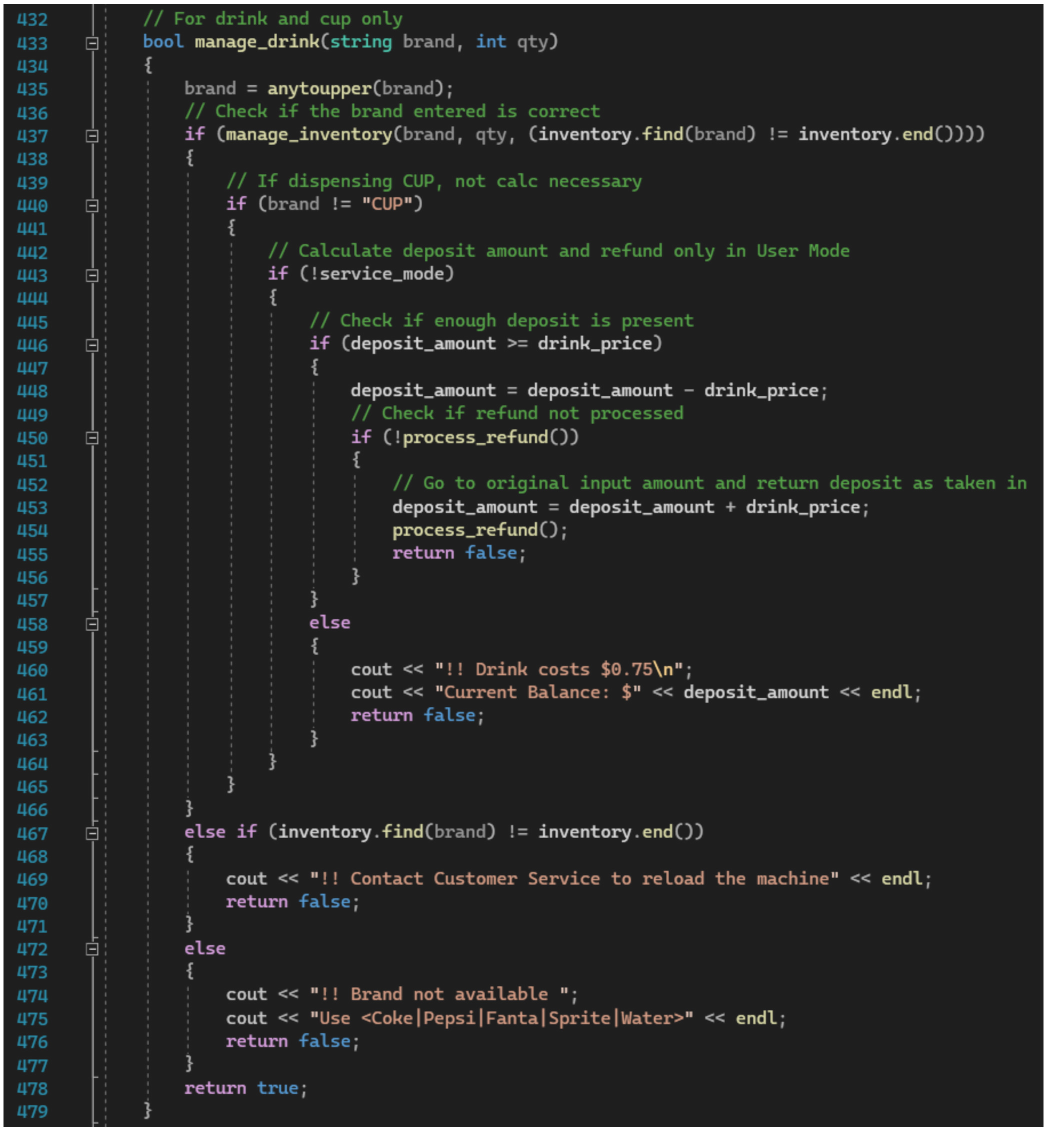
The looping switch structure of the command manager allows dedicated input error identification and state management using space delimited command line input strings stored into vectors. These strings can later be processed into uppercase to identify commands, brands, denominations, or converted into integers to be used as quantities. This method allows for easy expansion of commands in the future. For example, the input of the form [Command] <string 1> <int 2> … <map n>, as well as additional states if necessary.



```
// Command manager:
// Conveniently switch between states
switch (key)
{
case 1: // HELP - displays commands that can be used in the current mode
    help_text(service_mode);
    break;
case 2: // STATUS - read and display current inventory
    machine.display_current_inventory();
    break;
case 3: // EXIT - exits the command manager and ends the program
    exit = true;
    break;
case 4: // LOCK - checks password and changes to USER mode
    (compareCommand(input.at(1), passkey)) ? service_mode = false : service_mode = true;
    break;
case 5: // UNLOCK - checks password and changes to SERVICE mode
    (compareCommand(input.at(1), passkey)) ? service_mode = true : service_mode = false;
    break;
case 6: // VEND - vends one drink as requested in USER mode
    machine.vend_drink(input.at(1));
    break;
// case 7 and 8 reserved for extended functionality. Not needed for current scope of operation.
case 9: // COIN - manages coin input in both modes. Only one quantity allowed in USER mode
    (service_mode) ? machine.manage_coin(input.at(1), string_to_int(input.at(2))) : machine.insert_coin(input.at(1));
    break;
case 10: // BILL - manages bill input in both modes. Only one quantity allowed in USER mode
    (service_mode) ? machine.manage_bill(input.at(1), string_to_int(input.at(2))) : machine.insert_bill(input.at(1));
    break;
case 11: // DRINK - manages drink input in SERVICE mode
    machine.manage_drink(input.at(1), string_to_int(input.at(2)));

    break;
case 12: // CUP - manages cup input in SERVICE mode
    machine.manage_drink("CUP", string_to_int(input.at(1)));
    break;

default: // Command not recognized
    cout << "\n!! Command not recognized. Enter <HELP> for a list commands." << endl;
    break;
}
```

Figure 3: Command Manager

Condensing functions by using their required operational data and conditions allows for multi-use function calls. Example, "manage_drink" function (Figure 4, next page) inside the vending machine class can be used to alter the drink quantity in both user and service modes. However, user mode triggers a sub condition where it calculates deposit and the change to return. This also allows for conditional error messages such as "machine is out of change" or "not enough money inserted" all within the same function. Moreover, this method allows for expansion to other items like protein bars or candy at-will due to the input flexibly checking for the item in the entire inventory.



```cpp
// For drink and cup only
bool manage_drink(string brand, int qty)
{
    brand = anytoupper(brand);
    // Check if the brand entered is correct
    if (manage_inventory(brand, qty, (inventory.find(brand) != inventory.end())))
    {
        // If dispensing CUP, not calc necessary
        if (brand != "CUP")
        {
            // Calculate deposit amount and refund only in User Mode
            if (!service_mode)
            {
                // Check if enough deposit is present
                if (deposit_amount >= drink_price)
                {
                    deposit_amount = deposit_amount - drink_price;
                    // Check if refund not processed
                    if (!process_refund())
                    {
                        // Go to original input amount and return deposit as taken in
                        deposit_amount = deposit_amount + drink_price;
                        process_refund();
                        return false;
                    }
                }
                else
                {
                    cout << "!! Drink costs $0.75\n";
                    cout << "Current Balance: $" << deposit_amount << endl;
                    return false;
                }
            }
        }
    }
    else if (inventory.find(brand) != inventory.end())
    {
        cout << "!! Contact Customer Service to reload the machine" << endl;
        return false;
    }
    else
    {
        cout << "!! Brand not available ";
        cout << "Use <Coke|Pepsi|Fanta|Sprite|Water>" << endl;
        return false;
    }
    return true;
}
```

Figure 4: Condensed Multi-use functions

Insights and conclusion:

Using object-oriented programming, privately stored data containers in the class, and open-ended input and state switching allows for versatile use cases. For example, with minimal editing and more states, this program structure can be turned into a warehouse management software or a simple in-game item manager for a role-playing game. Moreover, the linear control modes draw similarity between how a program or user could interact with an operating system using the shell while the system generates error codes or processes data.