Design a Vending machine

Requirements:

1. Allow users to select products.
2. Accepts cash as payment.
3. Return the selected product and remaining change if any.
4. Allow reset operation for vending machine supplier.
5. Allow inventory manager to add/remove products.

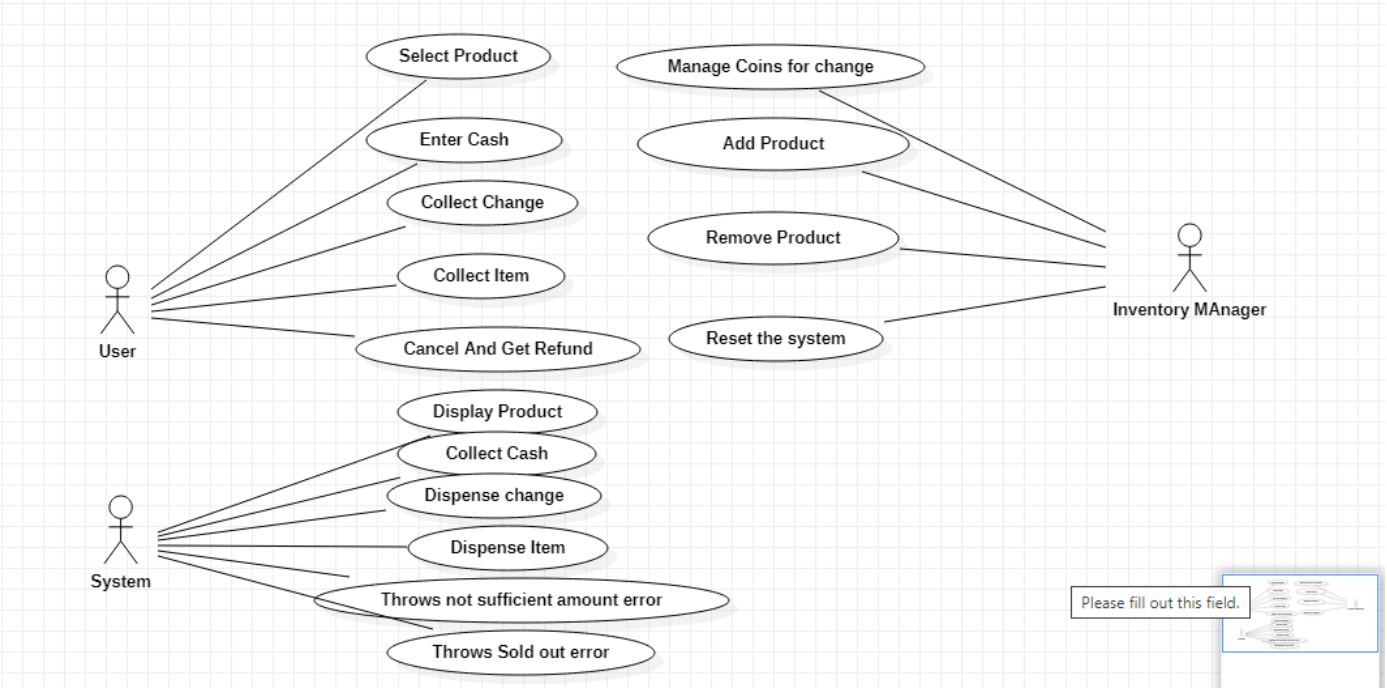
Use Case Diagram:

There are 3 actors involved in the system:

- User

- Inventory Manager
- Vending machine or system

Here are the top use cases for Vending Machine:



Class Diagram:

VendingMachine:

It defines the public API of a vending machine, usually, all high-level functionality should go in this class

Inventory:

Class to represent an Inventory, used for creating the cash and item inventory inside Vending Machine.

Item:

Class to represent Item served by Vending Machine.

Coin:

Another Enum to represent Coins supported by Vending Machine.

NotFullPaidException:

An Exception is thrown by Vending Machine when a user tries to collect an item, without

paying the full amount.

NotSufficientChangeException:
Vending Machine throws this exception to indicate that it doesn't have sufficient change to complete this request.

SoldOutExcepiton:
Vending Machine throws this exception if the user requests a product that is sold out.

StateMachine:
Class to handle all Vending Machine states.

inventory manager:
Class to adds and runs the vending machine

Implementation:

We can think of each use case: Display product, collect cash, etc as a series of commands to the vending machine system. And start thinking about implementing vending machines using the C**ommand design pattern**. But as we have already studied command pattern is suitable when there are multi-level rollbacks involved. And in Vending machines there are rollbacks involved only in canceling transactions otherwise no need to roll back any transaction.

It's one of the best examples of a ***State design pattern (*****trivial-use-of-the-state-pattern*****)***. In-state design pattern the vending machine behaves differently in different states like returning a product if the machine is not empty, otherwise, it just returns coins, so it ideally fits in the state design pattern.

We can define the following states for the Vending Machine:-

- Ready — Machine ready to display the items
- Item Selected: Item has been selected, It should be able to accept cash.
- Cash Collected — The machine has collected cash & the user can now select the product or cancel the transaction
- Cash Refund Processing — Give back the change to the user

- Item Dispensing — Dispense the item upon successful validation of entered cash & the price of the selected item in inventory
- Transaction Cancelled — If the user cancels the transaction, return the cash given by the user

```
from enum import Enum,auto
from enum import IntEnum
from abc import ABC, abstractmethod
from collections import defaultdict

# *********** Custom Exception handler ***************
class CustomError(Exception):
    def __init__(self,*args, **kwargs): # Pass any no of arguments
        self.message = kwargs.get('message')
        super().__init__(*args)

class NotFullPaidException(CustomError):
    pass

class NotSufficientChangeException(CustomError):
    pass

class SoldOutExcepiton(CustomError)
    pass

#********* Inventory class to hold items ******************

class Inventory:
    def __init__(self):
        self.storage = defaultdict() # item : number of count
        self.item_map = defaultdict()

    def add(self, item):
        self.storage[item]+=1
        self.item_map[item.code] = item

    def deduct(self, item):
        self.storage[item]-=1

    def has_item(self, code):
        return code in self.item_map

    def clear(self):
        self.storage = defaultdict()
        self.item_map = defaultdict()
```

```
        def put(self, item, quantity):
            print("Inventory manager called for item", item.code)
            self.storage[item] = quantity
            self.item_map[item.code] = item

        def display_item(self):
            print(len(self.item_map))
            for code, item in self.item_map.items():
                print(int(code), item.name)
                print()

        def get_item(self, code):
            return self.item_map[code]

    class ITEMS(IntEnum):
        PEPSI=1
        COKE=2
        KITKAT=3

    class CASH(Enum):
        FIVE=1
        TEN=2
        TWENTY=2

    #******** Item interface class to hold both product and cash ***
    class Item(ABC):
        def __init__(self, name, price, code):
            self.name = name
            self.price = price
            self.code = code

    def get_price():
            return self.price

    def get_name():
            return self.name

    #************ State machine Implementation ************************

    class STATES(Enum):
        READY=auto()
        ITEMSELECTED=auto()
        CASHCOLLECTED=auto()
        CHANGEPROCESSED=auto()
        ITEMDISPENSED=auto()
        ERROR=auto()

    class StateMachine:
        def __init__(self):
            self._handlers = {}
            self.start_state = STATES.READY
```

```
            self.end_states = []

        def _add_state(self, name, handler, end_state=0):
            self._handlers[name] = handler
            if end_state:
                self.end_states.append(name)

        def run(self):
            try:
                handler = self._handlers[self.start_state]
            except:
                raise InitializationError("must call .set_start() before
    .run()")
            if not self.end_states:
                raise  InitializationError("at least one state must be an
    end_state")

            while True:

                (newState) = handler()
                print("Arrived at a new state", newState)
                if newState in self.end_states:
                    print("reached ", newState)
                    break
                else:
                    handler = self._handlers[newState]

    #************ Actual Vending Machine Class ********

    class VendingMachine:
        def __init__(self):
            self.last_selected_product = None
            self.product_inventory = Inventory()
            self.cash_inventory = Inventory()
            self.total_cash_avl = 0
            self._init_state_machine()

        def display_item(self):
            self.product_inventory.display_item()
            while 1:
                code = int(input("select one item from above or type 0"))
                if code == 0:
                    break
                if not self.product_inventory.has_item(code):
                    continue

                item = self.product_inventory.get_item(code)
                self.last_selected_product = item
                break

            self.machine_state = STATES.ITEMSELECTED
```

```
            return self.machine_state


        def collect_cash(self):
            print("Collect cash called")
            return STATES.CASHCOLLECTED

        def process_cash_change(self):
            print("Process_cash_change called")
            return STATES.CHANGEPROCESSED

        def process_item_dispense(self):
            print("process_item_dispense called")
            return STATES.ITEMDISPENSED

        def _init_state_machine(self):
            self.state_machine = StateMachine()
            self.state_machine._add_state(STATES.READY, self.display_item)
            self.state_machine._add_state(STATES.ITEMSELECTED,
    self.collect_cash)
            self.state_machine._add_state(STATES.CASHCOLLECTED,
    self.process_cash_change)
            self.state_machine._add_state(STATES.CHANGEPROCESSED,
    self.process_item_dispense)
            self.state_machine._add_state(STATES.ITEMDISPENSED, None, 1)

    # Inventory Manager which will add products and Manage the cash
    # ************************************************************

    class InventoryManager:
        def __init__(self):
            self.machine = VendingMachine()
            self._initialize_inventories()

        def _initialize_inventories(self):
            # Add items to the vending machine
            # Add coins to inventory
            items = [Item("pepsi", 25, ITEMS.PEPSI), Item("Coke", 20,
    ITEMS.COKE), Item("kitkat", 15, ITEMS.KITKAT)]
            for item in items:
                self.machine.product_inventory.put(item, 5)

    def driver():
        manager = InventoryManager()
        manager._initialize_inventories()
        manager.machine.state_machine.run()

    driver()
```

Questions? Suggestions? Comments?