Vending Machine design — A State design pattern approach

Animesh Gaitonde Follow
Nov 17, 2019 · 6 min read ★

Designing a Vending Machine software using the State design pattern

Vending Machine

Introduction

A few years ago, I was interviewing with a Tech major & was asked to design a Vending Machine in one of the rounds. Back then, I had less experience with practical software design & didn't find time to cram the '*Gang of Four Design Patterns*' book.

However, I knew the theory well & thanks to the book '*Cracking the Coding Interview*', I had understood the approach to tackle a software design interview question. In this article, I'll walk you through the journey of how I started with a naive solution, improved the design, & finished off with a clean, modular and readable code.

Problem Statement

Vending Machine design was an open-ended and vague problem. At first thought, many different images flashed my mind and the problem looked intimidating. I decided to drill down the requirements by asking the Interviewer as many questions as I could.

After all the clarification, the expectation was to build software the could store items (track the inventory), accept cash, & dispense items. Additionally, the interviewer wanted the design and code to be extensible, reusable & modular.

Vending Machine

Requirements

I listed the following requirements for the above problem statement

- Vending Machine must keep track of the inventory

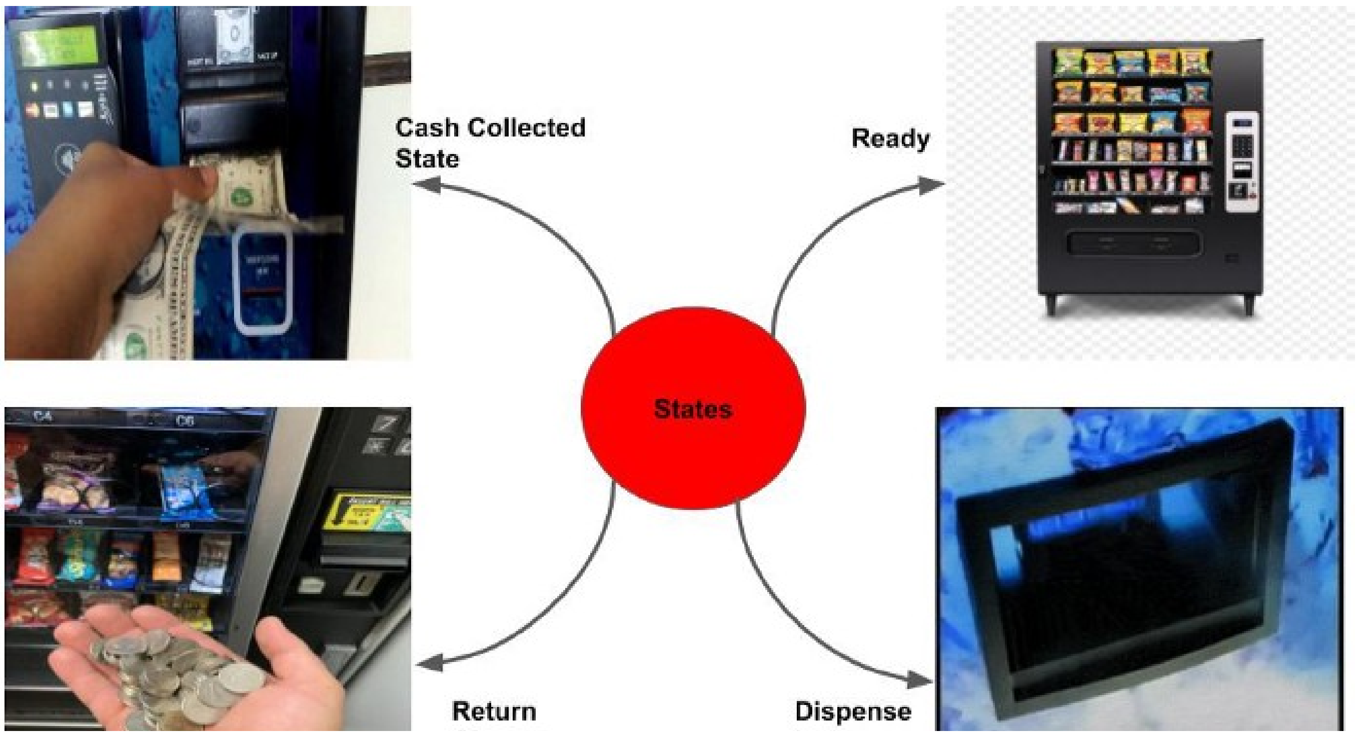
- A person should be able to insert cash into the machine & choose an item
- The Machine should confirm the inserted cash with the price of the selected item
- The machine must display an error in case of insufficient cash or unavailable item
- Finally, if all the above steps succeed then the user gets the selected item

Design

In any Object-oriented design interview, it always helps to identify the actors and the actions performed by them. In the above case, we have two actors Vending Machine and the User interacting. The user issues a command to the Machine and the Machine takes the necessary action.

If you think of buying an item like a transaction, the machine only processes one transaction at a time. For eg: If the machine is in the process of dispensing an item, then the user can't insert cash and try to buy another item. After the machine dispenses the item, the user can buy a new item.

In simple words, a user can buy a new item by either aborting or completing the existing transaction. To solve this we can define different states of the Vending Machine. Depending on the request, either the machine can change it's state or stay in the same state.

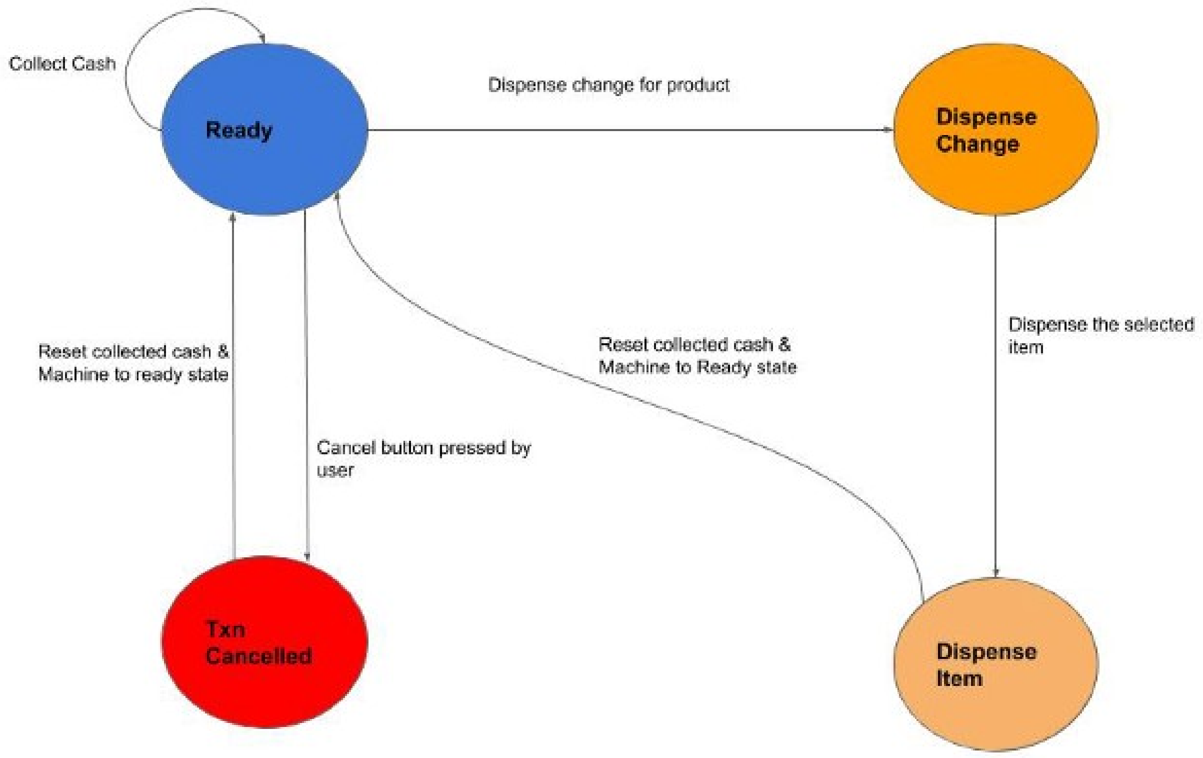


Vending Machine States

States

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

- Ready — Machine ready to accept cash
- CashCollected — Machine has collected cash & user can now select the product or cancel the transaction
- DispenseChange — Give back the change to the user
- DispenseItem — Dispense the item upon successful validation of entered cash & the price of the selected item in inventory
- TransactionCancelled — If the user cancels the transaction, return the cash given by the user



Collect Cash
Ready
Dispense change for product
Dispense Change
Reset collected cash & Machine to ready state
Cancel button pressed by user
Reset collected cash & Machine to Ready state
Dispense the selected item
Txn Cancelled
Dispense Item

Vending Machine State Transitions

Initial Code

Let’s take a look at the code that I wrote initially during the interview. I came up with the below naive code.

```java
package vendingMachine;

import java.util.Map;
import java.util.Set;

public class VendingMachine {
    private int collectedCash;
    private State state;
    private Map<String, Set<String>> productCodeItemMap;
    private Map<String, Integer> productCodePriceMap;

    public VendingMachine() {
        this.collectedCash = 0;
        this.state = State.READY;
    }

    public void collectCash(int cash) {
        switch (state) {
            case READY:
                handleCollectCash(cash);
                break;
            case DISPENSE_CHANGE:
                throw new RuntimeException("Dispensing change. Unable to collect cash");
            case DISPENSE_ITEM:
                throw new RuntimeException("Dispensing item. Unable to collect cash");
            case TRANSACTION_CANCELLED:
                throw new RuntimeException("Transaction cancelled. Unable to collect cas

        }
    }

    public void dispenseItem(String productCode) {
        switch (state) {
            case READY:
                throw new RuntimeException("Unable to dispense Item. Cash not collected"
            case DISPENSE_CHANGE:
                handleDispenseChange(productCode);
                break;
            case DISPENSE_ITEM:
                handleDispenseItem(productCode);
                break;
                break;
            case TRANSACTION_CANCELLED:
                throw new RuntimeException("Transaction cancelled. Unable to dispense It

        }
    }

    private void handleCollectCash(int cash) {
        this.collectedCash += cash;
    }

    private void handleDispenseItem(String productCode) {
        // logic to dispense item
    }

    private void handleDispenseChange(String productCode) {
        // logic to dispense Change
    }
}
```

BuggyVendingMachine.java hosted with ❤ by GitHub view raw

The class ‘***VendingMachine***’ exposed different methods to the user for interaction. Further, it encapsulated all the business logic to process the user commands. The code looked clumsy to me & I was sure the Interviewer would ask me a difficult question.

On taking a look, the Interviewer’s first question was ‘*Does your design confirm to the **S.O.L.I.D principles**?* ’. I quickly glanced at my code & assed the effort needed to introduce a new state. It would require the introduction of a new class with an additional switch-case block.

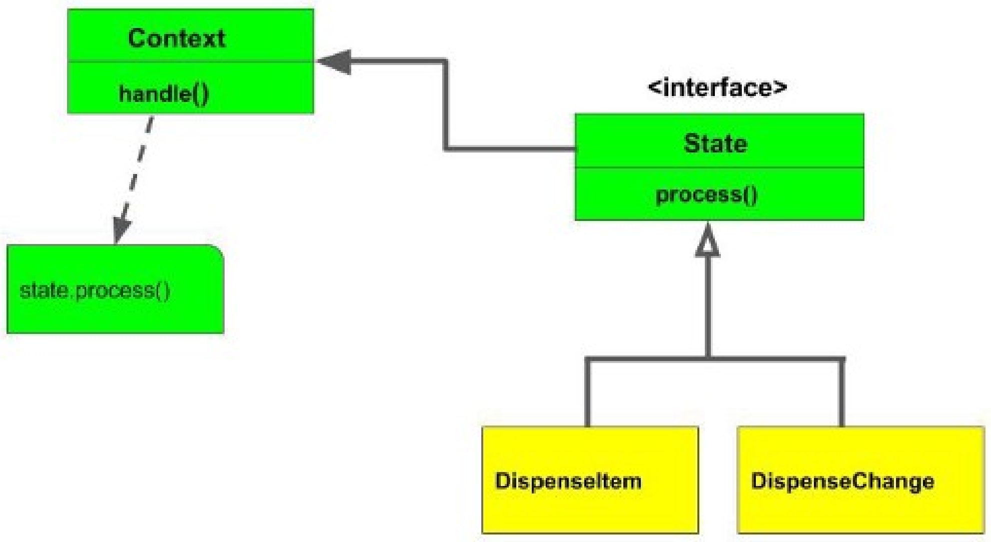
State-related logic is hard-coded in the *VendingMachine* class which means the *Single-Responsibility* principle is violated. Besides, a new feature requires me to modify the same class thus going against the *Open-Closed* principle.

I pondered over it for a while & eventually the thought of using ***State Design Pattern*** crossed my mind.

State Design Pattern

The core principle behind the State design pattern is to abstract out the state-related behaviour in a separate class. A context class stores a reference to the state class. The states can be accessed through a common interface. For every state-transition, the reference to the specific state class is modified.

Following is a UML representation of the State design pattern:-



UML Diagram of State Design Pattern

For the Vending Machine design, we can declare a state interface which exposes the APIs — *collectCash*, *dispenseChange*, *dispenseItem*, *cancelTransaction*

```java
package vendingMachine;

public interface State {
    public void collectCash(int cash);
    public void dispenseChange(String productCode);
    public void dispenseItem(String productCode);
    public void cancelTransaction();
}
```

VendingMachine.java hosted with ❤ by GitHub   view raw

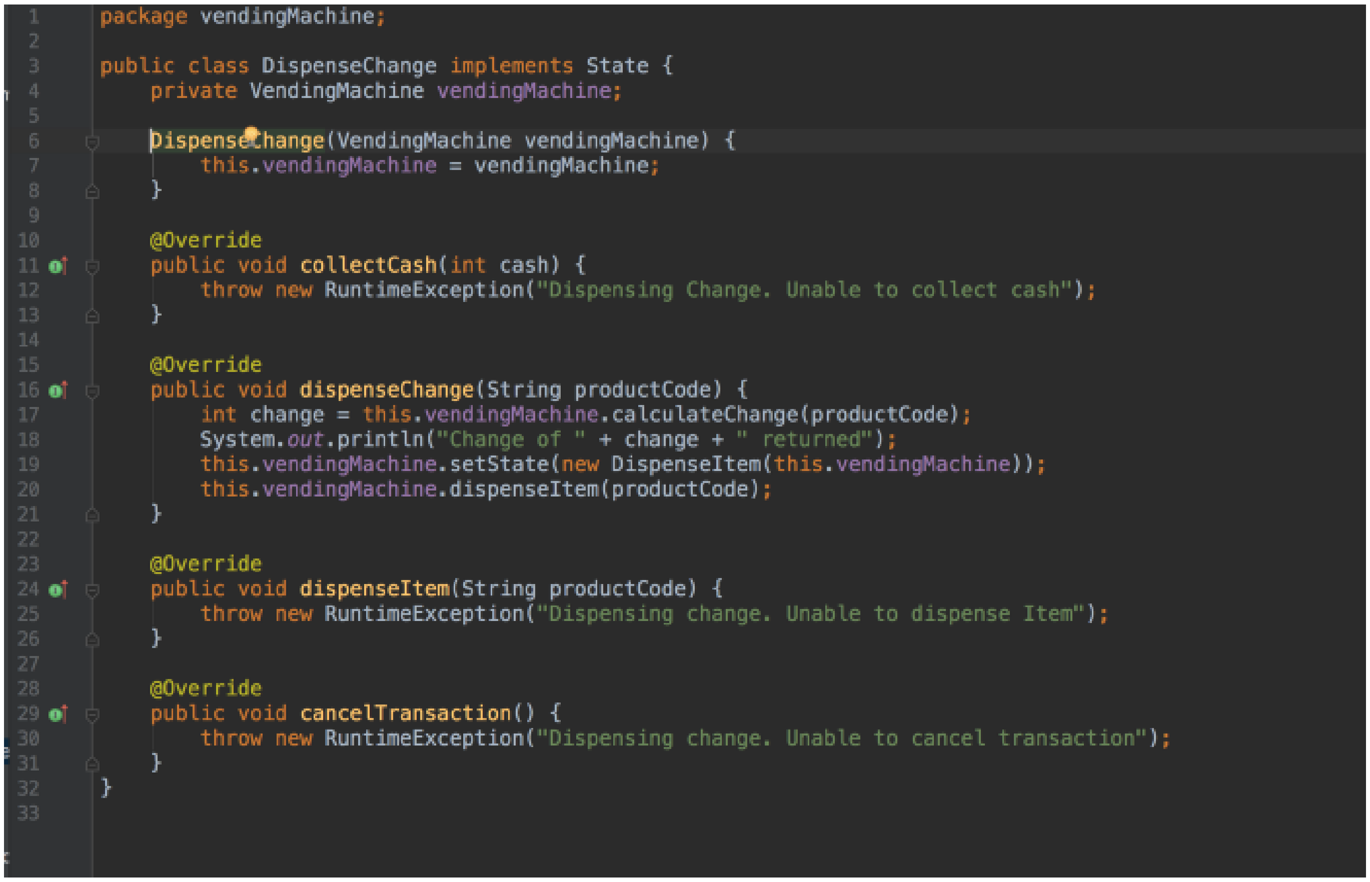
All the states that we identified will implement the state interface. The Vending Machine becomes a context and stores a reference to the state.

```java
public class VendingMachine {
    private int collectedCash;
    private State state;
    private Map<String, Set<String>> productCodeItemMap;
    private Map<String, Integer> productCodePriceMap;

    public void addCollectedCash(int collectedCash) {
        this.collectedCash += collectedCash;
    }

    public VendingMachine setCollectedCash(int collectedCash) {
        this.collectedCash = collectedCash;
        return this;
    }

    public State getState() {
        return state;
    }

    public VendingMachine setState(State state) {
        this.state = state;
        return this;
    }

    public void removeProduct(String productCode) {

    }

    public void dispenseChange(String productCode) {
        this.state.dispenseChange(productCode);
    }

    public void cancelTransaction() {
        this.state.cancelTransaction();
    }

    public int calculateChange(String productCode) {
        return collectedCash - productCodePriceMap.get(productCode);
    }

    public void dispenseItem(String productCode) {
        this.state.dispenseItem(productCode);
    }

    public int getCollectedCash() {
```

```
    public int getCollectedCash() {
        return collectedCash;
    }
}
```

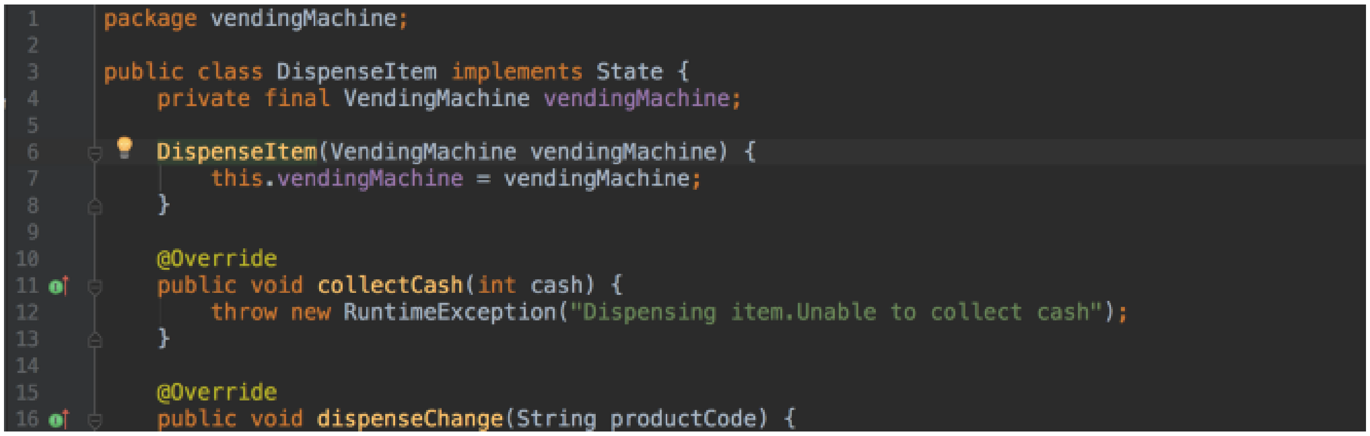
VendingMachineState.java hosted with ❤ by GitHub view raw

Vending Machine class will delegate all the actions that it receives to the specific state classes. The individual states will process the command and perform a state transition by resetting the state in the context. Let’s have look at my modified code.

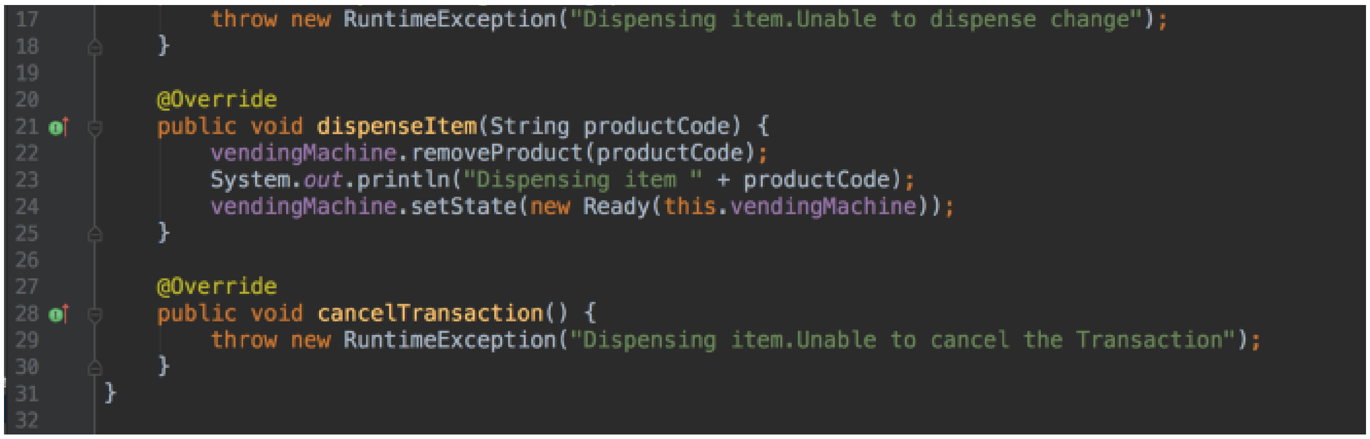


```
package vendingMachine;

public class DispenseChange implements State {
    private VendingMachine vendingMachine;

    DispenseChange(VendingMachine vendingMachine) {
        this.vendingMachine = vendingMachine;
    }

    @Override
    public void collectCash(int cash) {
        throw new RuntimeException("Dispensing Change. Unable to collect cash");
    }

    @Override
    public void dispenseChange(String productCode) {
        int change = this.vendingMachine.calculateChange(productCode);
        System.out.println("Change of " + change + " returned");
        this.vendingMachine.setState(new DispenseItem(this.vendingMachine));
        this.vendingMachine.dispenseItem(productCode);
    }

    @Override
    public void dispenseItem(String productCode) {
        throw new RuntimeException("Dispensing change. Unable to dispense Item");
    }

    @Override
    public void cancelTransaction() {
        throw new RuntimeException("Dispensing change. Unable to cancel transaction");
    }
}
```

DispenseChange

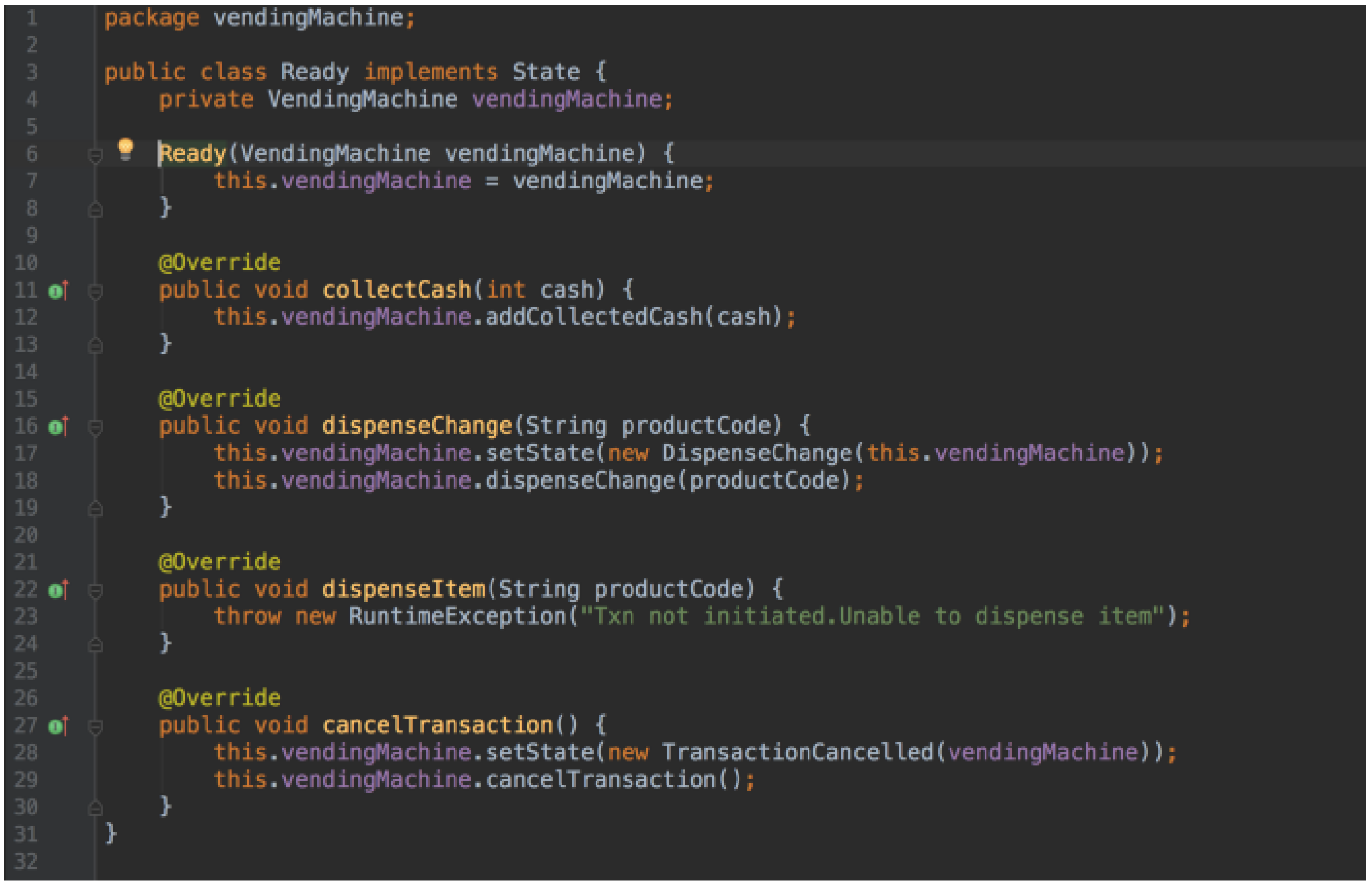


```
package vendingMachine;

public class DispenseItem implements State {
    private final VendingMachine vendingMachine;

    DispenseItem(VendingMachine vendingMachine) {
        this.vendingMachine = vendingMachine;
    }

    @Override
    public void collectCash(int cash) {
        throw new RuntimeException("Dispensing item.Unable to collect cash");
    }

    @Override
    public void dispenseChange(String productCode) {
```



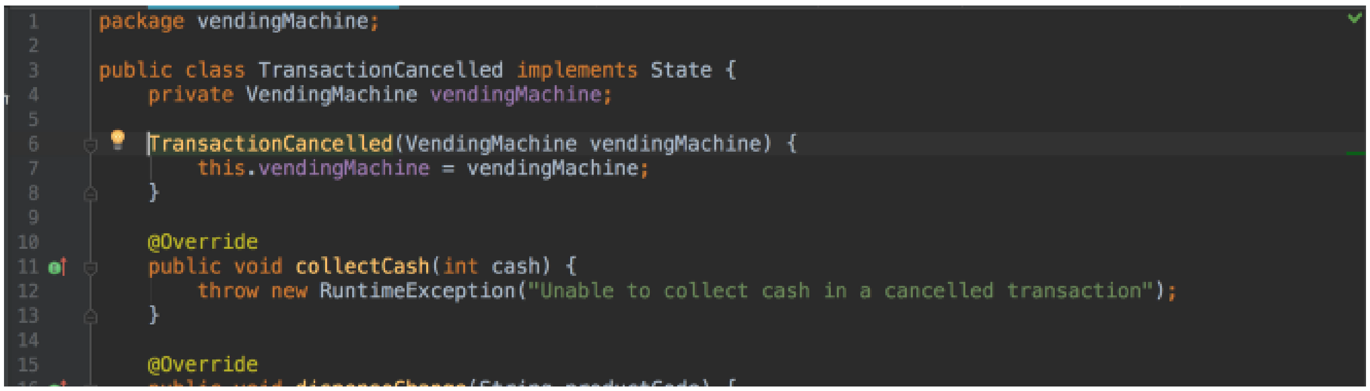
```java
        throw new RuntimeException("Dispensing item.Unable to dispense change");
    }

    @Override
    public void dispenseItem(String productCode) {
        vendingMachine.removeProduct(productCode);
        System.out.println("Dispensing item " + productCode);
        vendingMachine.setState(new Ready(this.vendingMachine));
    }

    @Override
    public void cancelTransaction() {
        throw new RuntimeException("Dispensing item.Unable to cancel the Transaction");
    }
}
```

DispenseItem

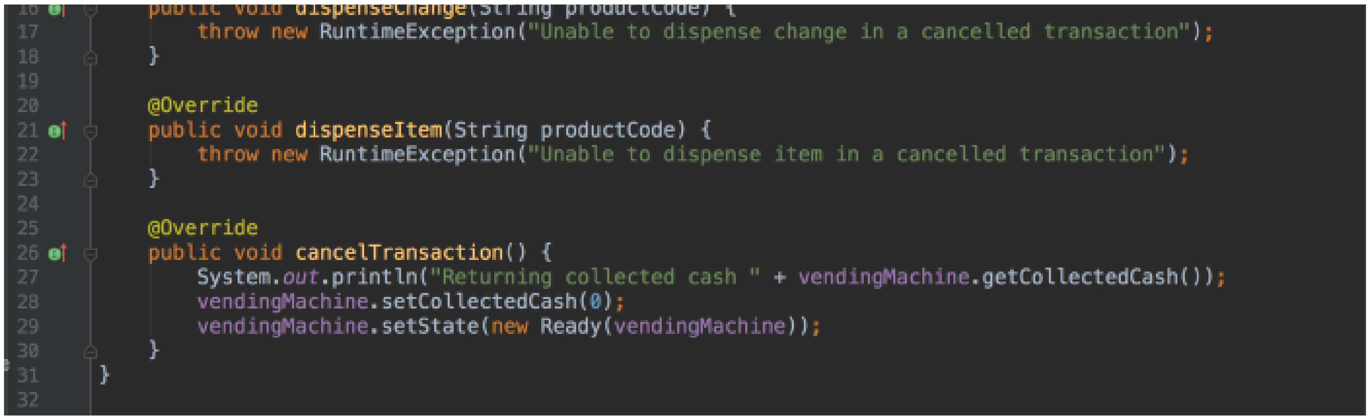


```java
package vendingMachine;

public class Ready implements State {
    private VendingMachine vendingMachine;

    Ready(VendingMachine vendingMachine) {
        this.vendingMachine = vendingMachine;
    }

    @Override
    public void collectCash(int cash) {
        this.vendingMachine.addCollectedCash(cash);
    }

    @Override
    public void dispenseChange(String productCode) {
        this.vendingMachine.setState(new DispenseChange(this.vendingMachine));
        this.vendingMachine.dispenseChange(productCode);
    }

    @Override
    public void dispenseItem(String productCode) {
        throw new RuntimeException("Txn not initiated.Unable to dispense item");
    }

    @Override
    public void cancelTransaction() {
        this.vendingMachine.setState(new TransactionCancelled(vendingMachine));
        this.vendingMachine.cancelTransaction();
    }
}
```

Ready



```java
package vendingMachine;

public class TransactionCancelled implements State {
    private VendingMachine vendingMachine;

    TransactionCancelled(VendingMachine vendingMachine) {
        this.vendingMachine = vendingMachine;
    }

    @Override
    public void collectCash(int cash) {
        throw new RuntimeException("Unable to collect cash in a cancelled transaction");
    }

    @Override
```



```
    public void dispenseChange(String productCode) {
        throw new RuntimeException("Unable to dispense change in a cancelled transaction");
    }

    @Override
    public void dispenseItem(String productCode) {
        throw new RuntimeException("Unable to dispense item in a cancelled transaction");
    }

    @Override
    public void cancelTransaction() {
        System.out.println("Returning collected cash " + vendingMachine.getCollectedCash());
        vendingMachine.setCollectedCash(0);
        vendingMachine.setState(new Ready(vendingMachine));
    }
}
```

TransactionCancelled

With the above code, a new state can be easily defined and plugged into the existing implementation with minimal change. Further, individual states are decoupled from each other. Finally, I had a vending machine codebase which was reusable, extensible, readable & clean.

Mission Accomplished

Advantages

- The design pattern moves all state-related logic to a separate class thus reducing the coupling with the main context class & is following the Single Responsibility Principle
- State-related behaviour is declared in an interface. New states can be easily introduced without the need to modify & add conditional blocks of code. Code becomes open for extension & closed for modification

Disadvantages

- Individual states must be aware of the next states and those states need to be hardcoded
- The pattern becomes an overkill if the design only has one or two states or the state behaviour rarely changes

Note:- You might find the State design pattern similar to the Strategy design pattern which was discussed in my last post here. The only difference is that in Strategy, the concrete strategy classes are not aware of each other whereas, in State pattern, the current state should be aware of the next state.

References:-

- https://en.wikipedia.org/wiki/State_pattern
- https://springframework.guru/gang-of-four-design-patterns/
- https://giphy.com/

Software Development Programming Design Patterns Software Engineering Best Practices