# Let's try to design an ATM Machine 💵 💰

## What are the requirements of the system ?

- Single Transaction Support:
- The system should allow only one transaction at a time for a particular user. No concurrent transactions.
Start Button:
- The ATM machine should have a Start button to initiate the transaction.
Card Insertion:
- Once the transaction starts, the machine should prompt the user to insert their card.
The system should validate the card details upon insertion.
Card Validation:
- If the card is invalid, the system should reject it and return it to the user.
Cash Withdrawal:
- After validating the card, the system should ask the user to enter the withdrawal amount.
The system should validate if the withdrawal amount can be dispensed based on the account balance and machine capacity.
Cancellation Support:
- Allowed Scenarios:
    - Before inserting the card.
    - After being prompted to

## Let's clarify the flow of an ATM machine !!

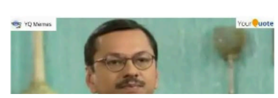
1. Transaction Start
    - Action: User presses the "Start Transaction" button.
    - API Call:
        - An API is triggered to start the transaction.
        - The API returns a unique transaction ID to track the interaction.
    - Next Step:
        - User proceeds or cancels.

2. Card Insertion
    - Action: User inserts their card into the machine.
    - API Call:
        - The card details are read and sent to an API for validation.
    - Validation Flow:
        - If valid: Proceed to the next step (Enter Amount).
        - If invalid:
            - Stop the transaction.
            - Eject the card and return to the initial state.

3. Enter Withdrawal Amount
    - Action: User enters the withdrawal amount on the machine.
    - API Call:
        - Validate if the entered amount can be dispensed (based on account balance and ATM cash availability).
    - Validation Flow:
        - If valid: Proceed to cash dispensing.
        - If invalid:
            - Allow the user to cancel or re-enter the amount.

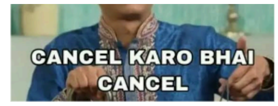
4. Cash Dispensing
    - Action: If the amount is valid, the ATM dispenses the cash.
    - API Call:
        - Close the transaction and record it for tracking purposes.
    - User Feedback:
        - Display a confirmation message indicating successful transaction completion.

5. Cancellation Options
    - Cancellation Points:
        - Before card insertion (API call to stop the transaction).
        - After card insertion but before entering the amount (API call to stop the transaction and eject the card).
        - After entering the amount but before cash dispensing (API call to stop the transaction and reset).
    - Restricted Cancellation:
        - Once cash dispensing has started, the transaction cannot be canceled.



6. Transaction Closure

Action: After cash dispensing or cancellation, the transaction is finalized.
API Call:
- Mark the transaction as completed or canceled.
- Record the transaction details for audit/logging.




## API Overview

→ Start Transaction: Initiated when the user presses the "Start" button, returning a transaction ID.

→ Cancel Transaction: Stops the transaction at any valid point and resets the state.

→ Validate Card: Checks if the card is valid or not.

→ Validate Amount: Ensures the entered amount can be dispensed.

→ Close Transaction: Finalizes the transaction and records the details.

## Basic ATM class overview

```java
public class ATM {

    // Start a transaction and return a transaction ID
    public int startTransaction() {
        // Logic to initiate a transaction and generate a transaction ID
        int transactionId = generateTransactionId();
        System.out.println("Transaction started with ID: " + transactionId);
        return transactionId;
    }

    // Cancel a transaction based on the transaction ID
    public boolean cancelTransaction(int transactionId) {
        // Logic to cancel the transaction
        System.out.println("Transaction with ID: " + transactionId + " canceled.");
        return true; // Return true if cancellation is successful
    }

    // Read card details and validate them
    public boolean readCard(String cardType, long cardNumber, int pin) {
        // Logic to validate card details
        System.out.println("Validating card: " + cardType + ", Card Number: " + cardNumber);
        boolean isValid = validateCardDetails(cardType, cardNumber, pin);
        if (isValid) {
            System.out.println("Card is valid.");
        } else {
            System.out.println("Invalid card details.");
        }
        return isValid;
    }

    // Withdraw a certain amount
    public boolean withdrawAmount(int transactionId, double amount) {
        // Logic to check if withdrawal is possible
        System.out.println("Processing withdrawal of amount: " + amount + " for transaction ID: " +
transactionId);
        boolean canWithdraw = checkWithdrawalLimit(transactionId, amount);
        if (canWithdraw) {
            System.out.println("Withdrawal approved.");
        } else {
            System.out.println("Withdrawal denied.");
        }
        return canWithdraw;
    }

    // Dispense cash
    public void dispenseCash(double amount) {
        // Logic to dispense cash
        System.out.println("Dispensing cash: " + amount);
        deductCashFromATM(amount);
    }

    // Eject the card
    public void ejectCard() {
        // Logic to eject the card
        System.out.println("Ejecting card...");
    }

    // Private helper methods
    private int generateTransactionId() {
        // Simulate generating a transaction ID
        return (int) (Math.random() * 100000);
    }

    private boolean validateCardDetails(String cardType, long cardNumber, int pin) {
        // Simulate card validation logic
        return true; // Assume the card is valid for simplicity
    }

    private boolean checkWithdrawalLimit(int transactionId, double amount) {
        // Simulate checking withdrawal limits
        return amount <= 1000; // Assume a max limit of 1000 for simplicity
    }

    private void deductCashFromATM(double amount) {
        // Simulate deducting cash from ATM balance
        System.out.println("ATM balance updated after dispensing " + amount);
    }

    // Main method for testing
    public static void main(String[] args) {
        ATM atm = new ATM();

        // Simulate an ATM transaction
        int transactionId = atm.startTransaction();
        boolean cardValid = atm.readCard("VISA", 1234567890123456L, 1234);

        if (cardValid) {
            boolean canWithdraw = atm.withdrawAmount(transactionId, 500);
            if (canWithdraw) {
                atm.dispenseCash(500);
            } else {
                atm.cancelTransaction(transactionId);
            }
        } else {
            atm.ejectCard();
        }
    }
}
```

Any problems that we can identify based on an ATM machine functionality ?

Problem 1: Missing Instance Variables
Issue: The class only defines methods but lacks fields to hold specific properties of an ATM, such as a unique identifier (atmId) or its state.
Solution:
Add instance variables to represent the ATM's ID and state.
Use these fields to track the machine's identity and current status.

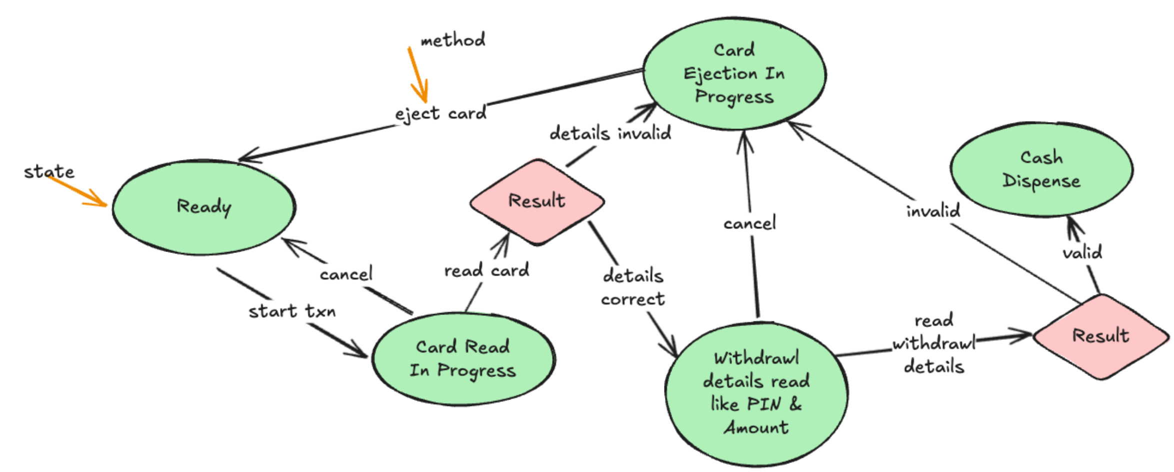
Problem 2: No Support for Multiple ATM Instances
Issue: The current design assumes a single ATM and does not account for multiple ATM instances in different locations.
Solution:
Ensure each ATM object has a unique identifier (atmId) assigned during instantiation.
Pass the atmId as a parameter to the constructor.

Problem 3: Lack of State Management
Issue: The class does not manage the current state of the ATM, leading to potential issues like starting a new transaction while another is in progress.
Solution:
Introduce an enum to define possible ATM states (IDLE, TRANSACTION_IN_PROGRESS, etc.).
Use a state variable to track and enforce state transitions.

Problem 4: Concurrency Issues
Issue: No mechanism to prevent concurrent transactions, leading to possible conflicts or incorrect behavior.
Solution:
Check the ATM's state before starting a transaction.
Throw an exception or error if a transaction is already in progress.

Problem 5: Undefined State Transitions
Issue: The transitions between ATM states are not explicitly defined, making the flow ambiguous.
Solution:
Clearly define state transitions and ensure operations are allowed only in valid states.
Update the state after each successful operation.

Let's discuss different states of the ATM machien




🌐https://pastebin.com/LVvrFfST

```java
public class ATM {

    // Enum for ATM states
    private enum ATMState {
        IDLE, TRANSACTION_IN_PROGRESS, CARD_INSERTED, AMOUNT_ENTERED, DISPENSING_CASH
    }

    // Instance variables
    private String atmId;         // Unique identifier for the ATM
    private ATMState state;       // Current state of the ATM

    // Constructor
    public ATM(String atmId) {
        this.atmId = atmId;
        this.state = ATMState.IDLE; // Default state is IDLE
    }

    // Start a transaction
    public int startTransaction() {
        if (state != ATMState.IDLE) {
            throw new IllegalStateException("ATM is currently busy with another transaction.");
        }
        state = ATMState.TRANSACTION_IN_PROGRESS;
        int transactionId = generateTransactionId();
        System.out.println("Transaction started with ID: " + transactionId + " on ATM: " +
atmId); return transactionId;
    }

    // Cancel a transaction
    public void cancelTransaction() {
        if (state == ATMState.IDLE) {
            throw new IllegalStateException("No transaction to cancel.");
        }
        state = ATMState.IDLE;
        System.out.println("Transaction canceled on ATM: " + atmId);
    }

    // Read card details and validate
    public boolean readCard(String cardType, long cardNumber, int pin) {
        if (state != ATMState.TRANSACTION_IN_PROGRESS) {
            throw new IllegalStateException("Cannot read card. Start a transaction first.");
        }
        state = ATMState.CARD_INSERTED;
        boolean isValid = validateCardDetails(cardType, cardNumber, pin);
        if (!isValid) {
            state = ATMState.IDLE;
        }
        return isValid;
    }

    // Enter withdrawal amount
    public boolean enterAmount(double amount) {
        if (state != ATMState.CARD_INSERTED) {
            throw new IllegalStateException("Card must be inserted before entering an amount.");
        }
        state = ATMState.AMOUNT_ENTERED;
        return checkWithdrawalLimit(amount);
    }

    // Dispense cash
    public void dispenseCash(double amount) {
        if (state != ATMState.AMOUNT_ENTERED) {
            throw new IllegalStateException("Amount must be entered before dispensing cash.");
        }
        state = ATMState.DISPENSING_CASH;
        System.out.println("Dispensing cash: " + amount + " on ATM: " + atmId);
        state = ATMState.IDLE; // Reset state after dispensing
    }

    // Eject card
    public void ejectCard() {
        if (state == ATMState.IDLE) {
            throw new IllegalStateException("No card to eject.");
        }
        state = ATMState.IDLE;
        System.out.println("Card ejected from ATM: " + atmId);
    }

    // Private helper methods
    private int generateTransactionId() {
        return (int) (Math.random() * 100000); // Generate random transaction ID
    }

    private boolean validateCardDetails(String cardType, long cardNumber, int pin) {
        // Simulate card validation
        return true; // Assume valid for simplicity
    }

    private boolean checkWithdrawalLimit(double amount) {
        // Simulate withdrawal limit check
        return amount <= 1000; // Assume a max limit of 1000
    }
}
```

Any problems that we can identify based on an ATM class ?

| SOLID Principle | Violation | Impact |
| --- | --- | --- |
| Single Responsibility (SRP) | `ATM` class handles multiple concerns (state management, business logic, error handling). | Hard to maintain and scale; any change in one responsibility might affect others. |
| Open/Closed (OCP) | Adding new states requires modifying existing methods with additional `if` checks. | Code is not extensible and prone to bugs during changes. |
| Liskov Substitution (LSP) | Polymorphism is not leveraged for state-specific logic; tightly coupled logic limits substitutability. | Future extensions or substitutions (e.g., new ATM types) are not supported. |
| Interface Segregation (ISP) | Single class handles all functionality, leading to a bloated interface. | Testing and extending functionality becomes difficult. |
| Dependency Inversion (DIP) | No abstraction for state management; tightly coupled methods and state logic. | Poor scalability; hard to modify or extend the behavior without touching core functionality. |

Lack of Fine-Grained State Management

Problem:

The state variable transitions are hard-coded into the methods.

The code requires multiple if checks for each possible state in every operation. Suppose later startTransaction function can work on more than one states so more if checks will be needed.
If new states are introduced, each method will require updates, leading to cluttered and error-prone code.
Impact: Adding new states or transitions will result in significant refactoring.

Missing State-Specific Logic Encapsulation
Problem:
Each operation (init, cancelTransaction, dispenseCash, etc.) has to handle behavior for multiple states, making the methods cluttered and hard to maintain.
There is no clear separation of behavior for different states.
Impact: The logic for state management is spread across multiple methods instead of being encapsulated.

Lack of Validation for Concurrent Transactions
Problem:
The init method does not prevent concurrent transactions effectively.
Even if the ATM is in a state like CASH_DISPENSING, a new transaction can still be initiated unless explicit checks are added.
Impact: This can lead to conflicts and unpredictable behavior.

Risk of Code Duplication
Problem:
Repeated state checks in every method (e.g., checking if the state is READY, TRANSACTION_IN_PROGRESS, etc.).
Each new feature or state will require redundant checks in all related methods.
Impact: Leads to bloated code and higher maintenance costs.

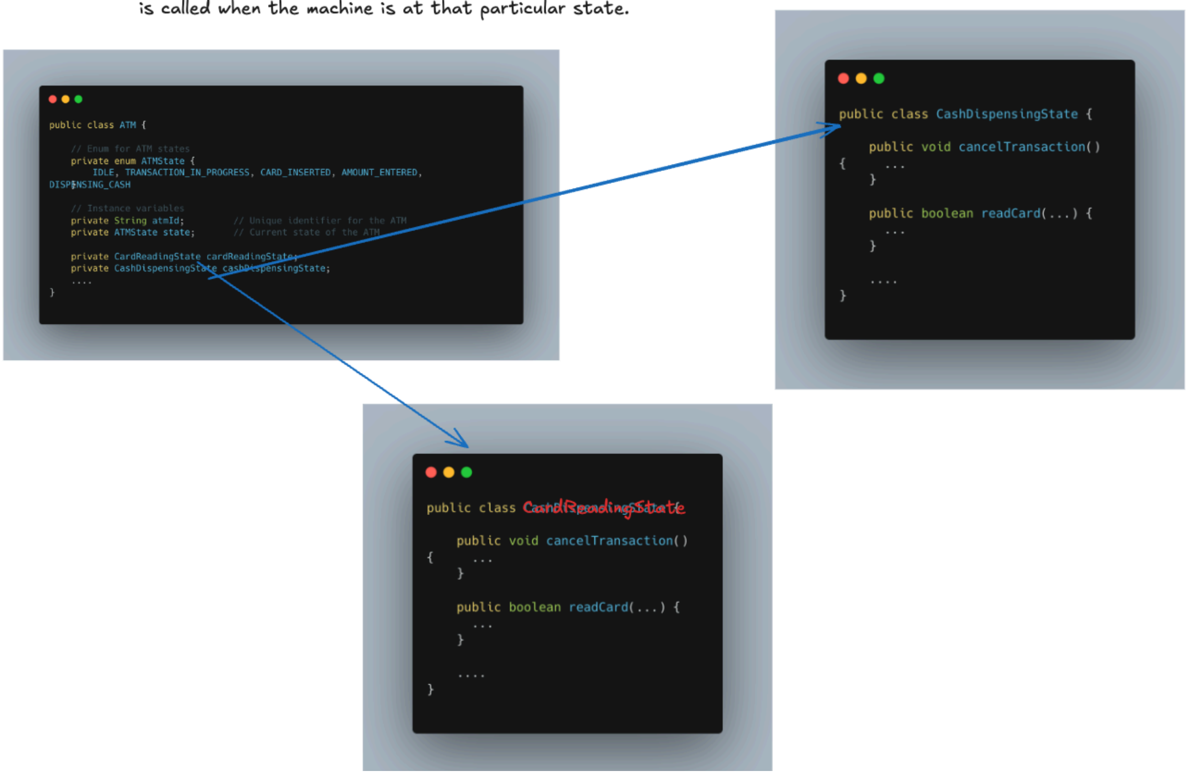
Inefficient Handling of Future State Extensions
Problem:
Adding new states (e.g., ERROR_STATE, OUT_OF_SERVICE) will require modifying every method that deals with state transitions.
This tightly couples the state logic with the core business logic.
Impact: Poor scalability and increased risk of introducing bugs during future enhancements.

The ATM machine is a typical case of creating a state machine

How can we improve more on the solution ?

- Currently our ATM class has a lot of responsibility, how about we create separate classes for each state.

- In each state class we will be having the functionality of how the state machine will react when a function is called when the machine is at that particular state.



## Are we now violating any SOLID principle ?

- We are violating the Dependency inversion principle as now our ATM class depends on a lot of concrete state classes.

- How can we resolve it ?

- May be we can have a State interface and then multiple classes can implement this interface and our ATM class will be only depending on one State object which will be getting the instance of the corresponding state based on what is going on in the txn.