Somaiya Vidyavihar University
K J Somaiya School of Engineering

Batch: A2 Roll No.: 16010322014
Experiment / assignment / tutorial No. 03
Grade: AA / AB / BB / BC / CC / CD /DD
Signature of the Staff In-charge with date

TITLE: System Design Analysis using UML
AIM: Design an Embedded System Model using UML/FSM Simulator

1. **OUTCOME:** Understand the concepts of co-design and co-simulation and testing methodology for Embedded system.

Simulator Used: -

- **https://app.diagrams.net/**
- **Draw.io (diagrams.net)**

Hardware Architecture: -

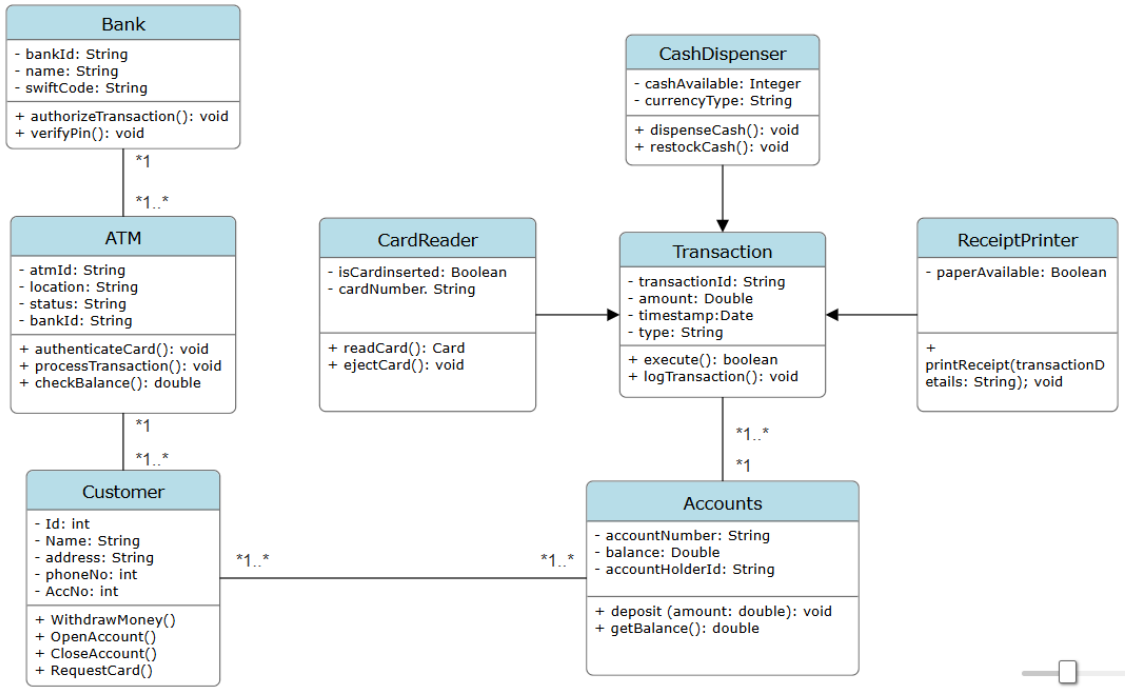
- Processor: A microcontroller would serve as the "heart" of the ATM, running all its software and coordinating the hardware.
- Memory: This includes Program Memory to hold the ATM's operating software and Data Memory for transaction details.
- Input Devices Interfacing Circuits: These circuits would connect the processor to the ATM's keypad and card reader.
- Outputs Interfacing Circuits: These would drive the ATM's LCD display, receipt printer, and the mechanical cash dispenser.
- Communication Ports: A serial or network port would be used to connect the ATM to the bank's central servers.
- Interrupt Controller and Timers: These manage hardware requests and ensure operations happen within specific time periods, which is critical for a real-time system like an ATM

Software Architecture: -

- Application Software: This is the top layer and would be the main program that the user interacts with. It includes the logic for
- `processTransaction()`, `checkBalance()`, and displaying menus on the screen.
- API (Application Programming Interface): This acts as the middleman between your ATM application and the Operating System. For example, when your application needs to print a receipt, it makes a call to the OS through the API.
- Operating System: For an ATM, this would be a Real-Time Operating System (RTOS) because it must respond to events within strict deadlines. This layer is made of several components:
    - Kernel: The core of the RTOS. It manages all the different software tasks (e.g., one task for the keypad, another for the dispenser), schedules them, and handles memory.
    - Device Manager: This uses specific device drivers to control the ATM's physical hardware, like the `CashDispenser` and `ReceiptPrinter`.

- Communication Software: This manages the secure network connection to the bank for authorizing transactions.
- File System: Could be used for storing transaction logs or other data locally on the ATM

UML Model for the proposed System



Procedure: -

1. **Identify the Class**: Determine the class you want to model, representing a real-world object or concept.
2. **Draw the Class Box**: Create a rectangle divided into three sections:
   - **Top Section**: Name of the class (e.g., `Person`).
   - **Middle Section**: List attributes with visibility (e.g., `- name: String`) and data types.
   - **Bottom Section**: List methods with visibility, return types, and parameters (e.g., `+ getName(): String`).
3. **Define Visibility**:
   - + for public, **-** for private, # for protected.
4. **Add Relationships**: Show interactions with other classes:
   - **Association**: Solid line (can include an arrow).
   - **Aggregation**: Hollow diamond at the line end.
   - **Composition**: Filled diamond at the line end.
   - **Inheritance**: Solid line with a hollow triangle.
   - **Dependency**: Dashed line with an arrow.
5. **Specify Multiplicity**: Add number constraints to relationships (e.g., `1..*`, `0..1`).
6. **Refine and Review**: Double-check for completeness and correctness, ensuring the diagram clearly represents class relationships and structure.

Code: -

```
// Sample UML class diagram for a banking system, written in Umple.
// --- Class Definitions ---

class Bank {
  String bankId;
  String name;
  String swiftCode;

  // Methods
  authorizeTransaction() {}
  verifyPin() {}
}

class ATM {
  String atmId;
  String location;
  String status;
  String bankId;

  // Methods
  authenticateCard() {}
  processTransaction() {}
  Double checkBalance() { return 0.0; }
}

class Customer {
  Integer id;
  String name;
  String address;
  Integer phoneNo;
  Integer accNo;

  // Methods
  withdrawMoney() {}
  OpenAccount() {}
  CloseAccount() {}
  RequestCard() {}
}

class Accounts {
  String accountNumber;
  Double balance;
  String accountHolderId;

  // Methods
  deposit(Double amount) {}
```

```
  Double getBalance() { return 0.0; }
}

class Transaction {
  String transactionId;
  Double amount;
  Date timestamp;
  String type;

  // Methods
  Boolean execute() { return true; }
  logTransaction() {}
}

class CardReader {
  Boolean isCardInserted;
  String cardNumber;

  // The diagram shows "readCard(): Card", but there is no Card class defined.
  // The return type is omitted here for simplicity as per Umple syntax.
  readCard() {}
  ejectCard() {}
}

class CashDispenser {
  Integer cashAvailable;
  String currencyType;

  // Methods
  dispenseCash() {}
  restockCash() {}
}

class ReceiptPrinter {
  Boolean paperAvailable;

  // Methods
  printReceipt(String transactionDetails) {}
}

// --- Associations between Classes ---

// A Bank is associated with one or more ATMs.
association {
  1 Bank -- 1..* ATM;
}

// An ATM is associated with one or more Customers.
// Note: This relationship in a real system might be modeled differently,
// but this reflects the provided diagram.
```

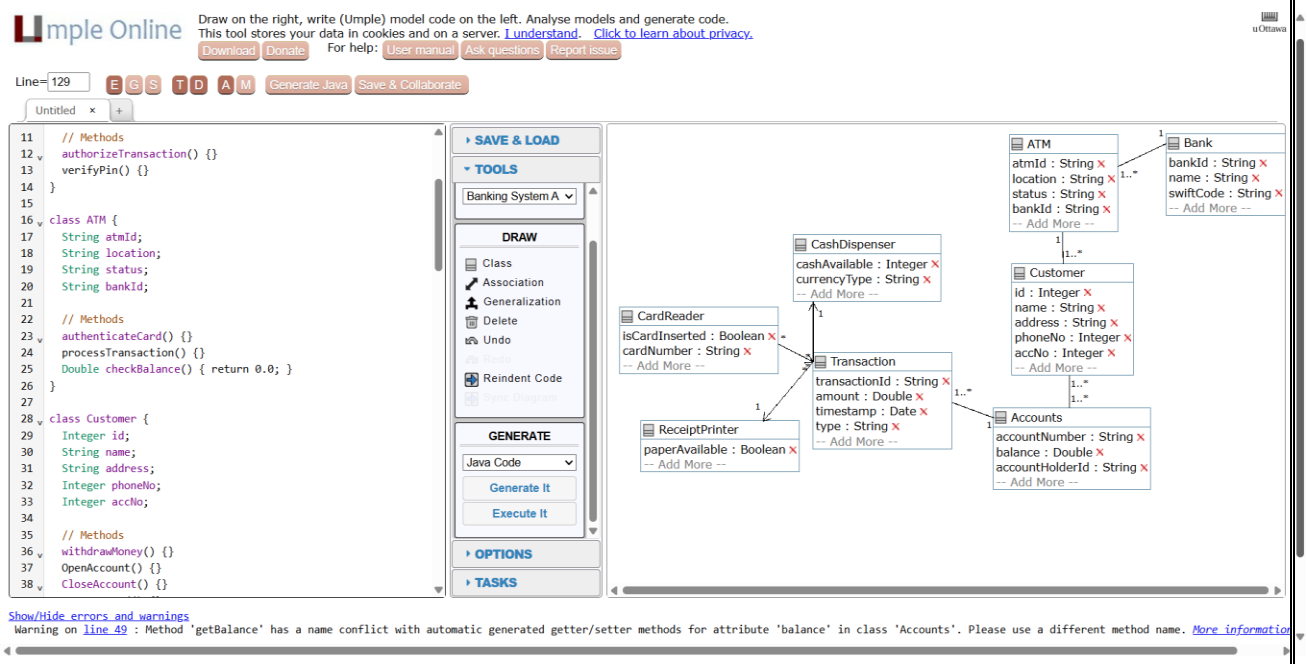
```
association {
  1 ATM -- 1..* Customer;
}
// One or more Customers can have one or more Accounts (e.g., joint accounts).
association {
  1..* Customer -- 1..* Accounts;
}
// An Account is linked to one or more Transactions.
association {
  1 Accounts -- 1..* Transaction;
}

// --- Directed Associations (Dependencies) ---

// The CardReader initiates or is linked to a Transaction.
association {
  * CardReader -> * Transaction;
}
// A Transaction uses a CashDispenser to dispense cash.
association {
  * Transaction -> 1 CashDispenser;
}

// A Transaction uses a ReceiptPrinter to print a receipt.
association {
  * Transaction -> 1 ReceiptPrinter;
}
```

Observations: -



Conclusion:

In this experiment, we explored co-design and co-simulation for embedded system modelling using UML and FSM simulators. The process helped us understand how to design and test both hardware and software components together for optimal performance. By leveraging UML for system modelling and FSM for simulating state transitions, we learned how to validate embedded systems early in development, improving reliability and efficiency before physical implementation.

Signature of faculty in-charge with date