UML State Machine Diagrams and Their Implementation in Java

Affan Rauf

National University of Computer and Emerging Sciences

December 18, 2024

Introduction to UML State Machine Diagrams

- UML State Machine Diagrams represent the different states of an object throughout its lifecycle.
- They model how an object transitions from one state to another in response to events or conditions.
- The states are depicted as rounded rectangles, while transitions are shown as arrows connecting the states.

Key Components of UML State Machine Diagrams

- **State:** Represents a situation during the life of an object where it satisfies some condition.
- **Transition:** The movement from one state to another due to an event or condition.
- **Event:** A trigger that causes a state transition.
- **Action:** The behavior executed in response to a transition.

The Importance of Internal State in State Machine Diagrams

- One of the primary advantages of UML State Machine Diagrams is their ability to model behavior that is dependent on the internal state of an object.
- **Unlike other behavioral diagrams**, state machines focus on situations where:
  - The object's output or behavior is not just a result of external inputs.
  - The internal state of the object plays a crucial role in determining the outcome of an event or action.
- This is particularly useful in systems where:
  - The same input can result in different outcomes based on the current state.
  - The object must maintain internal memory of its past interactions.

ATM Machine Example: Behavior Dependent on State

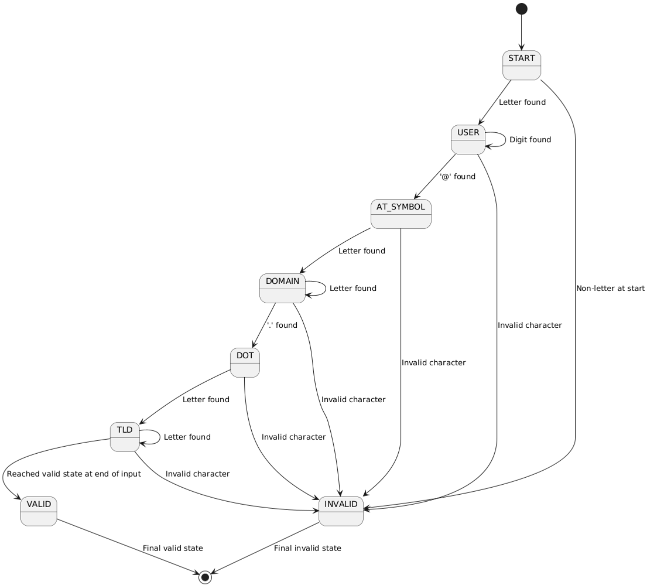
- **ATM Machine Scenario:**
  - The ATM machine has a fixed set of inputs — 12 keys on the keypad.
  - However, the behavior or interpretation of the same key press changes depending on the current screen or state.
- **Key Example:** Pressing the key '1'
  - On the **Home Screen**, pressing '1' may initiate a balance inquiry.
  - On the **PIN Entry Screen**, pressing '1' is part of the user's PIN input.
  - On the **Withdrawal Amount Screen**, pressing '1' selects Rs.100 withdrawal.
- **State Dependency:**
  - The input alone does not determine the action — the internal state (screen) defines what behavior will occur.
  - This dynamic behavior is precisely what UML State Machine Diagrams capture.

Problem: Email Address Validation State Machine

Problem: Design a state machine to check if a string follows the pattern of a valid email address: `[user]@[domain].[TLD]`.

- `[user]` must start with a letter and then can have any number of digits.
- `[domain]` and `[TLD]` must not contain any digits.
- `[domain]` can only contain one dot (".") separating it from `[TLD]`.

State Machine Diagram



EmailValidator Class - Switch-Based State Machine Implementation

State Machine Implementation: A state machine can be implemented using a simple switch-based code structure, where:

- Each `switch` case represents a state in the state machine.
- Transitions between states occur based on the input character and the current state.
- Invalid transitions are caught, and the system is moved to an `INVALID` state.

In this example, the email validation process transitions between various states:

- `START`: Beginning of the email string.
- `USER`: The user part of the email.
- `AT_SYMBOL`: When the '@' character is encountered.
- `DOMAIN`: Domain part of the email.
- `DOT`: Detecting the dot between domain and TLD.
- `TLD`: The Top-Level Domain.
- `INVALID/VALID`: Final states, indicating whether the email is valid or not.

Code Snippet: EmailValidator Class

```
public class EmailValidator {
    // Enum to represent different states of the state machine
    enum State {
        START, USER, AT_SYMBOL, DOMAIN, DOT, TLD, INVALID, VALID
    }
    public static boolean validateEmail(String email) {
        State state = State.START; // Initial state
        char[] chars = email.toCharArray();
        int i = 0;
        while (i < chars.length) {
            char c = chars[i];
            switch (state) {
                case START:
                    if (Character.isLetter(c)) {
                        state = State.USER;
                    } else {
                        state = State.INVALID;
                    }
                    break;
                case USER:
                    if (Character.isDigit(c)) {
                        // Stay in USER state as long as valid chars
                    } else if (c == '@') {
                        state = State.AT_SYMBOL;
                    } else {
                        state = State.INVALID;
                    }
                    break;
```

Code Snippet: EmailValidator Class

```
case AT_SYMBOL:
    if (Character.isLetter(c)) {
        state = State.DOMAIN;
    } else {
        state = State.INVALID;
    }
    break;
case DOMAIN:
    if (Character.isLetter(c)) {
        // Stay in DOMAIN state as long as valid domain letters
    } else if (c == '.') {
        state = State.DOT;
    } else {
        state = State.INVALID;
    }
    break;
case DOT:
    if (Character.isLetter(c)) {
        state = State.TLD;
    } else {
        state = State.INVALID;
    }
    break;
case TLD:
    if (Character.isLetter(c)) {
        // Stay in TLD state as long as valid chars
    } else {
        state = State.INVALID;
    }
    break;
```

Code Snippet: EmailValidator Class

```
                case INVALID:
                    return false; // Once invalid, we can exit
                default:
                    state = State.INVALID;
                    break;
            }
            i++;
        }
        // Check if we ended up in the valid state
        if (state == State.TLD) {
            state = State.VALID;
        }
        return state == State.VALID;
    }
}
```

State Design Pattern: An Alternative to Switch-Case

What is the State Design Pattern?

- The State Design Pattern is a behavioral design pattern.
- It allows an object to alter its behavior when its internal state changes.
- It is an alternative to using switch-case statements or conditionals for managing state transitions.

Key Concept: Instead of using conditionals, each state is represented as a separate class, and state transitions are handled by delegating to the appropriate state object.

How the State Design Pattern Works

Key Components:

- **Context:** The class that contains the state and delegates behavior to the current state object.
- **State Interface:** Defines the behavior that each state should implement.
- **Concrete States:** Different classes representing each possible state. Each class implements the behavior associated with that state.

State Transitions:

- The context holds a reference to the current state object.
- When an event occurs, the context delegates the task to the current state, which decides the next state.

Benefits of Using the State Design Pattern

- **Clear Separation of Concerns:** Each state has its own class, so state-specific behavior is encapsulated.
- **Open/Closed Principle:** Adding new states doesn't require modifying existing states. You just add a new class.
- **Simplified Maintenance:** Reduces complex switch-case logic or conditionals in the main class.
- **Extensibility:** Easy to extend the state machine by adding new states without affecting the existing logic.

The pattern is particularly useful when there are many possible states and transitions between them.

State Design Pattern vs Switch-Case

Switch-Case Approach:

- `Pros:` Simple and effective for small state machines.
- `Cons:` Becomes difficult to maintain as the number of states increases. Tight coupling between states and logic.

State Design Pattern:

- `Pros:` Better separation of concerns and follows the Open/Closed Principle. Easier to extend and maintain.
- `Cons:` More complex to set up initially. Involves more classes and boilerplate code.

State Design Pattern Example: Email Validator

Context: The `EmailValidator` class holds the current state and delegates behavior.

- **State Interface:** Defines methods like `handleChar(char c)`.
- **Concrete States:** Classes like `UserState`, `DomainState`, `TLDState` that implement the behavior for handling email validation in their respective states.

State Transitions: Each state object determines the next state based on the character provided.