ECE 811 – SOFTWARE ENGINEERING

FINITE STATE MACHINE IN SOFTWATE ENGINEERING –STUDY GUIDE/REVISION

OBJECTIVE

Master the design, implementation, and application of FSMs for robust software systems.

1. INTRODUCTION TO FINITE STATE MACHINES

1. **Definition**

   A Finite State Machine (FSM) is a computational model defining a system's behaviour using a finite number of **states**, **transitions** between states triggered by **events**, and associated **actions**.

2. **Purpose of FSM in Software Engineering**

   - Simplify complex logic (e.g., User Interface(UI) workflows, game AI, network protocols).
   - Ensure predictable behaviour and easier debugging.
   - Model real-world processes (e.g., vending machines, traffic lights).

2. CORE CONCEPTS IN FINITE STATE MACHINES

1. **States**

   - Distinct configurations of the system (e.g., Idle, Processing, Success, Error).
   - **Initial State**: Entry point (e.g., Idle).
   - **Final/Accepting State(s)**: Terminal states (e.g., Success).

2. **Transitions**

   - Directed change from one state to another triggered by an **event** (e.g., onButtonClick).

3. **Events**

   - Inputs or conditions activating transitions (e.g., user actions, system signals).

4. **Actions**

   - Operations executed during transitions (e.g., validateInput(), sendRequest()).

3. TYPES OF FINITE STATE MACHINES

Type	Key Characteristics	Use Cases
Deterministic (DFA)	One transition per event/state pair. Predictable.	Protocol parsing, input validation.
Non-Deterministic (NFA)	Multiple possible transitions per event/state.	Regex engines, complex AI.
Mealy Machine	Actions **depend on transitions** (events + current state).	Network controllers, robotics.
Moore Machine	Actions **depend solely on states**.	Simple UI controllers, hardware.

4. MODELLING FINITE STATE MACHINES

1. **State Diagrams (Visual)**
    - Use circles for **states** and arrows for **transitions**.
    - Label transitions: Event [Guard] / Action.
    - Example:

[Idle] -- buttonClick → [Processing] / validateInput()

[Processing] -- success → [Success] / showMessage()

2. **State Transition Tables (Tabular)**

Current State	Event	Next State	Action
Idle	buttonClick	Processing	Validate Input
Processing	success	Success	Show Message

5. IMPLEMENTING FSMs IN CODE

1. **State Pattern (OOP)**
    - Define an abstract State class/interfaced with event handlers.
    - Concrete states (e.g., IdleState, ProcessingState) implement behavior.
    - Context class holds the current state.

```
// Example in Java
interface State {
    void handleButtonClick(Context context);
}
class IdleState implements State {
```

```
    public void handleButtonClick(Context context) {
        validateInput();
        context.setState(new ProcessingState());
    }
}
```

2. **State Tables (Data-Driven)**
   - Use a 2D array or dictionary mapping (currentState, event) to (nextState, action).
   - Ideal for large FSMs (e.g., game AI):

```
# Python Example
fsm_table = {
    ('Idle', 'button_click'): ('Processing', validate_input),
    ('Processing', 'success'): ('Success', show_message),
}
```

3. **Libraries**:
   - Python: transitions, pytransitions.
   - JavaScript: xstate, machina.js.
   - C#: Stateless.

6. PRACTICAL EXAMPLES

1. **User Interface (UI) Workflow**
   - States: LOGIN_SCREEN → INPUT_VERIFICATION → HOME_SCREEN.
   - Events: submit_form, validation_success.
2. **Payment Gateway**
   - States: PAYMENT_INITIATED → PROCESSING → SUCCESS/FAILED.
3. **Game AI (Enemy Behavior)**
   - States: PATROL → CHASE → ATTACK → RETREAT.
   - Events: player_spotted, health_low.

7. ADVANTAGES & CHALLENGES

1. **Pros**
   - Clarity in complex logic.
   - Easy to extend/modify states.
   - Reusable across components.

2. **Cons**
    - **State explosion**: Too many states become unmanageable (use **hierarchical FSMs**).
    - Not suitable for concurrency (consider **statecharts** or **Petri nets**).

8. ADVANCED TOPICS

1. **Hierarchical FSMs**
    - Nest FSMs within states (e.g., GameLevel state contains Paused/Running sub-states).
2. **Statecharts**
    - Extend FSMs with concurrency, history, and compound states.
3. **Testing FSMs**
    - Validate all transitions via unit tests.
    - Tools: cucumber (Gherkin scenarios), model checkers.

9. EXERCICES

1. Sketch a state diagram for a real-world system (e.g., microwave oven).
2. Code a simple FSM using the state pattern in your preferred language.
3. Explore xstate or transitions to build FSMs declaratively.