State Pattern Implementation in SkillForge

1. Introduction

SkillForge is a skill exchange platform where users can teach and learn from each other. A core workflow involves **learning sessions**, which translates to a flow where a student requests a session from a teacher, credits are held in escrow, and upon completion, the credits transfer to the teacher. And that's only for Requested to Accepted state flow.

Managing session lifecycle presents a challenge: a session transitions through multiple states (Requested, Accepted, Completed, etc.), each with distinct allowed operations and behaviors. Rather than scattering conditional logic throughout the codebase, we apply the **State Pattern** to encapsulate state-specific behavior within dedicated classes.

---

2. Pattern Overview

The **State Pattern** allows an object to alter its behavior when its internal state changes. The object appears to change its class.

Pattern Structure

Role	SkillForge Mapping
Context	`Session` — the object whose behavior varies by state
State	`SessionState` — interface defining state-dependent operations
ConcreteState	`RequestedState`, `AcceptedState`, `RejectedState`, `CompletedState`, etc.

This approach:

- **Eliminates conditionals** — no switch-case or `if (state == "REQUESTED")` checks scattered throughout the codebase
- **Encapsulates behavior** — each state class handles its own logic
- **Enforces valid transitions** — invalid operations throw exceptions

---

3. Domain Model

3.1 Core Entities

The session lifecycle involves three key entities working together:

```
public class Session {
    private String id;
    private String skillListingId;
    private String studentId;
    private String escrowTransactionId;
```

```
    private LocalDateTime scheduledTime;

    private SessionState state;    // Current state object (behavior)
    private String stateString;    // Persisted for DB queries

    // Delegate operations to current state
    public void accept(SessionContext ctx) { state.accept(this, ctx); }
    public void reject(SessionContext ctx) { state.reject(this, ctx); }
    public void complete(SessionContext ctx) { state.complete(this, ctx);
}
}
```

```
public class Wallet {
    private String userId;
    private BigDecimal balance;
    private BigDecimal lockedCredits;  // Held in escrow

    public void lockCredits(BigDecimal amount) { /* ... */ }
    public void releaseCredits(BigDecimal amount) { /* ... */ }
    public void transferLockedCredits(BigDecimal amount, Wallet recipient)
{ /* ... */ }
}
```

```
public class EscrowTransaction {
    private String id;
    private String fromWalletId;   // Student
    private String toWalletId;     // Teacher
    private BigDecimal amount;
    private EscrowStatus status;   // HELD, RELEASED, TRANSFERRED,
REFUNDED
}
```

3.2 Concrete Example

Consider a scenario where **Bob** (student) requests a Java session from **Alice** (teacher):

Object	Key Attributes
`javaListing`	creditCost = 50, teacher = Alice
`session1`	state = REQUESTED, scheduledTime = null
`bobWallet`	availableCredits = 50, lockedCredits = 50
`escrow1`	amount = 50, status = HELD

When Bob requests the session, 50 credits are locked from his wallet and held in escrow until the session outcome is determined.

4. Session Request Flow

When a student requests a session, three coordinated steps occur:

```
public Session requestSession(String studentId, String listingId,
LocalDateTime proposedTime) {
    SkillListing listing = listingRepository.findById(listingId);
    Wallet studentWallet = walletRepository.findByUserId(studentId);

    // Step 1: Lock credits in student's wallet
    studentWallet.lockCredits(listing.getCreditCost());

    // Step 2: Create escrow transaction (credits held until resolution)
    EscrowTransaction escrow = new EscrowTransaction();
    escrow.setAmount(listing.getCreditCost());
    escrow.setStatus(EscrowStatus.HELD);
    escrowRepository.save(escrow);

    // Step 3: Create session with initial state
    Session session = new Session();
    session.setState(new RequestedState(proposedTime));
    session.setEscrowTransactionId(escrow.getId());

    return sessionRepository.save(session);
}
```

At this point, the session exists in `RequestedState`, awaiting the teacher's response.

5. State Interface and Transitions

5.1 State Interface

The `SessionState` interface defines all possible operations. Each concrete state implements only the valid ones:

```
public interface SessionState {
    void accept(Session session, SessionContext context);
    void reject(Session session, SessionContext context);
    void counterOffer(Session session, SessionContext context,
LocalDateTime newTime);
    void complete(Session session, SessionContext context);
    void dispute(Session session, SessionContext context, String reason);

    String getStateName();
}
```

5.2 RequestedState

The initial state after a student requests a session:

```
public class RequestedState implements SessionState {
    private LocalDateTime proposedTime;

    @Override
    public void accept(Session session, SessionContext ctx) {
        // Teacher accepts → schedule session, transition to AcceptedState
        session.setScheduledTime(this.proposedTime);
        session.setState(new AcceptedState(this.proposedTime));
        ctx.getNotificationService().notify(session.getStudentId(),
"Session accepted!");
        // Escrow remains HELD (credits transfer only on completion)
    }

    @Override
    public void reject(Session session, SessionContext ctx) {
        // Teacher rejects → refund student, transition to RejectedState
        EscrowTransaction escrow =
ctx.getEscrowRepository().findById(session.getEscrowTransactionId());
        Wallet studentWallet =
ctx.getWalletRepository().findById(escrow.getFromWalletId());

        studentWallet.releaseCredits(escrow.getAmount());  // Unlock
credits
        escrow.setStatus(EscrowStatus.REFUNDED);
        session.setState(new RejectedState());
    }

    @Override
    public void complete(Session session, SessionContext ctx) {
        // Invalid: cannot complete a session that hasn't been accepted
        throw new IllegalStateTransitionException("Cannot complete a
requested session");
    }
}
```

5.3 AcceptedState

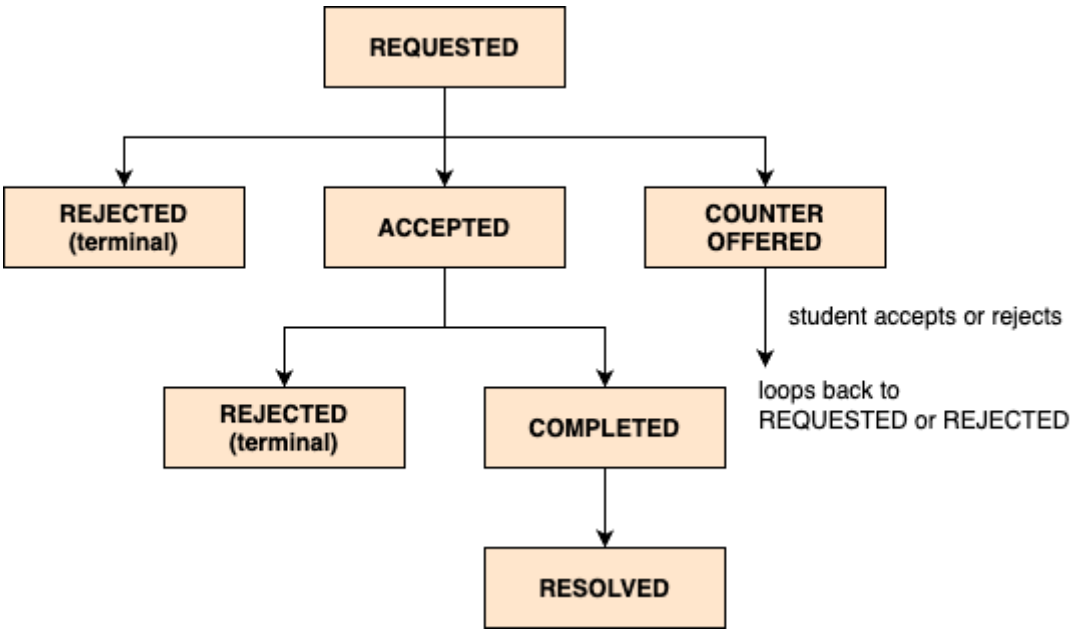
After the teacher accepts, the session can be completed or disputed:

```
public class AcceptedState implements SessionState {

    @Override
    public void complete(Session session, SessionContext ctx) {
        // Session finished → transfer credits to teacher
        EscrowTransaction escrow =
ctx.getEscrowRepository().findById(session.getEscrowTransactionId());
        Wallet studentWallet =
ctx.getWalletRepository().findById(escrow.getFromWalletId());
```

```
        Wallet teacherWallet =
ctx.getWalletRepository().findById(escrow.getToWalletId());

        studentWallet.transferLockedCredits(escrow.getAmount(),
teacherWallet);
        escrow.setStatus(EscrowStatus.TRANSFERRED);
        session.setState(new CompletedState());
    }

    @Override
    public void accept(Session session, SessionContext ctx) {
        throw new IllegalStateTransitionException("Session already
accepted");
    }

    /* ... */
}
```

---

6. State Transition Diagram



Escrow Status by State

Transition	Escrow Action
Request → **Requested**	HELD (credits locked)
Accept → **Accepted**	HELD (unchanged)
Reject → **Rejected**	REFUNDED (credits returned to student)
Complete → **Completed**	TRANSFERRED (credits sent to teacher)
Dispute → **Disputed**	HELD (frozen until resolution)

---

7. Conclusion

The State Pattern provides an elegant solution for managing session lifecycle in SkillForge:

- **Maintainability** — State-specific logic is isolated in dedicated classes
- **Extensibility** — Adding new states (like `RescheduledState`) requires minimal changes
- **Safety** — Invalid state transitions throw exceptions, preventing inconsistent states
- **Clarity** — The escrow/wallet integration is explicit within each state transition

This pattern is particularly well-suited for workflows with well-defined states and transitions, such as order processing, booking systems, or—as demonstrated here—learning session management.