Design Patterns and Functional Requirements

Overview

This document provides a comprehensive list of the design patterns adopted in the AIU Trips & Events Management System and their corresponding functional requirements. The design patterns were carefully selected to improve code maintainability, scalability, and adherence to SOLID principles.

Summary of Adopted Design Patterns

A total of **11 design patterns** were implemented across the system:

Creational Patterns (4)

1. Factory Pattern
2. Abstract Factory Pattern (integrated with Builder)
3. Builder Pattern
4. Prototype Pattern

Structural Patterns (3)

5. Adapter Pattern
6. Bridge Pattern
7. Decorator Pattern

Behavioral Patterns (4)

8. Command Pattern
9. Chain of Responsibility Pattern
10. State Pattern
11. Strategy Pattern

Additional Pattern:

- Memento Pattern (for state management)

---

1. Factory Pattern

Location

`com.aiu.trips.factory` - Repository Layer

Components

- `IModelFactory` - Factory interface
- `ModelFactory` - Concrete factory implementation with registry system
- `IBaseModel<T>` - Base model interface

- `IReadModel<T>` - Read operations interface
- `IWriteModel<T>` - Write operations interface

Purpose

Centralized creation and registration of repository models to decouple model instantiation from business logic.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-1.1	User Registration	Factory creates UserModel for registration operations
FR-2.1	Event Management	Factory creates ActivityModel for event CRUD operations
FR-3.1	Booking Management	Factory creates BookingModel for booking operations
FR-4.1	Notification System	Factory creates NotificationModel for notification operations
FR-5.1	Reporting & Analytics	Factory creates ReportModel for report generation

Implementation Details

```
// Register models
modelFactory.register("userModel", userModel);
modelFactory.register("activityModel", activityModel);

// Retrieve and use models
IBaseModel<User> userModel = modelFactory.get("userModel");
```

---

2. Abstract Factory Pattern

Location

`com.aiu.trips.builder` - Activity Management (integrated with Builder)

Components

- `IActivityFactory` interface concept (realized through builders)
- `EventBuilder` - Creates Event entities
- `TripBuilder` - Creates Trip entities

Purpose

Provide an interface for creating families of related objects (Events and Trips) without specifying their concrete classes.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-2.1	Create Events	Factory provides consistent event creation interface
FR-2.2	Create Trips	Factory provides consistent trip creation interface
FR-2.3	Activity Type Management	Factory handles different activity types polymorphically

Implementation Details

```
// Different builders for different activity types
IActivityBuilder eventBuilder = new EventBuilder();
IActivityBuilder tripBuilder = new TripBuilder();
```

---

3. Builder Pattern

Location

`com.aiu.trips.builder` - Activity Management

Components

- **`IActivityBuilder`** - Builder interface
- **`EventBuilder`** - Concrete builder for Events
- **`TripBuilder`** - Concrete builder for Trips
- **`IActivityDirector`** - Director interface
- **`ActivityDirector`** - Build orchestrator

Purpose

Construct complex Activity objects step-by-step, separating the construction of complex objects from their representation.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-2.4	Complex Event Creation	Builder handles multi-step event construction
FR-2.5	Trip Itinerary Building	Builder constructs trips with multiple attributes
FR-2.6	Activity Validation	Director ensures all required fields are set
FR-2.7	Different Activity Variants	Director can create different activity configurations

Implementation Details

```
IActivityBuilder builder = new EventBuilder();
IActivityDirector director = new ActivityDirector();
```

```
director.setBuilder(builder);
ActivityDTO event = director.makeNormalEvent(data);
```

4. Prototype Pattern

Location

`com.aiu.trips.prototype` - Activity Management

Components

- **`IPrototype<T>`** - Interface with `clone()` method
- Implemented by **`EventEntity`** and **`Trip`**

Purpose

Enable cloning of Activity objects for template-based creation and duplication.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-2.8	Duplicate Events	Clone existing events as templates
FR-2.9	Recurring Activities	Create similar activities from prototypes
FR-2.10	Template Management	Maintain activity templates for quick creation

Implementation Details

```java
public class EventEntity extends Activity implements IPrototype<EventEntity> {
    @Override
    public EventEntity clone() {
        // Deep copy implementation
        return clonedEvent;
    }
}
```

5. Adapter Pattern

Location

`com.aiu.trips.adapter` - Notification Component

Components

- **`IEmailService`** - Target interface
- **`SmtpEmailAdapter`** - Adapter wrapping JavaMailSender

Purpose

Integrate third-party email services (JavaMailSender) with the system's notification interface.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-4.2	Email Notifications	Adapter integrates SMTP service
FR-4.3	Multiple Email Providers	Adapter allows switching email providers
FR-4.4	Email Template Support	Adapter handles email formatting

Implementation Details

```
@Component
public class SmtpEmailAdapter implements IEmailService {
    @Autowired
    private JavaMailSender mailSender;

    @Override
    public void sendEmail(String to, String subject, String body) {
        // Wrap JavaMailSender functionality
    }
}
```

---

6. Bridge Pattern

Location

`com.aiu.trips.bridge` - Notification Component

Components

- `NotificationChannel` interface (Implementor)
    - `EmailChannel`
    - `InAppChannel`
- `NotificationMessage` abstract class (Abstraction)
    - `NewEventMessage`
    - `EventUpdateMessage`
    - `ReminderMessage`

Purpose

Decouple notification channels from message types, allowing independent variation of both.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-4.5	Multi-Channel Notifications	Bridge separates channels from content
FR-4.6	Dynamic Channel Selection	Channels can be switched at runtime
FR-4.7	Message Type Flexibility	New message types don't affect channels
FR-4.8	Channel-Specific Formatting	Each channel formats messages appropriately

Implementation Details

```
NotificationChannel emailChannel = new EmailChannel(emailService);
NotificationMessage message = new NewEventMessage();
message.setChannel(emailChannel);
message.send(user, event);
```

---

7. Decorator Pattern

Location

`com.aiu.trips.decorator` - Booking & Ticketing

Components

- **`ITicketService`** - Component interface
- **`BaseTicketService`** - Concrete component
- **`TicketServiceDecorator`** - Abstract decorator
  - `SignedQrDecorator` - Adds signed QR codes
  - `AuditLogDecorator` - Adds audit logging

Purpose

Dynamically add responsibilities to ticket services without modifying their structure.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-3.2	QR Code Generation	SignedQrDecorator adds secure QR codes
FR-3.3	Ticket Security	Decorators add security features
FR-3.4	Audit Trail	AuditLogDecorator tracks ticket operations
FR-3.5	Flexible Ticket Features	Decorators can be combined as needed

Implementation Details

```
ITicketService ticketService = new BaseTicketService();
ticketService = new SignedQrDecorator(ticketService);
ticketService = new AuditLogDecorator(ticketService);
Ticket ticket = ticketService.generateTicket(booking);
```

8. Command Pattern

Location

`com.aiu.trips.command` - Controller Layer

Components

- **`IControllerCommand`** - Command interface
- **`ControllerCommandInvoker`** - Command queue manager
- Concrete Commands:
  - `RegisterCommand`
  - `LoginCommand`
  - `CreateEventCommand`
  - `UpdateEventCommand`
  - `DeleteEventCommand`
  - `BookEventCommand`
  - `SendNotificationCommand`
  - `GenerateReportCommand`

Purpose

Decouple request handling from execution, enabling command queuing, logging, and undo operations.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-1.2	User Authentication	LoginCommand encapsulates authentication
FR-2.11	Event CRUD Operations	Commands encapsulate event operations
FR-3.6	Booking Operations	BookEventCommand handles booking logic
FR-4.9	Notification Dispatch	SendNotificationCommand manages notifications
FR-5.2	Report Generation	GenerateReportCommand handles report requests
FR-6.1	Request Logging	Invoker logs all commands
FR-6.2	Operation History	Commands can be stored for audit

Implementation Details

```
@Autowired
private ControllerCommandInvoker invoker;

IControllerCommand loginCommand = new LoginCommand(authService);
invoker.pushToQueue(loginCommand);
ResponseEntity<?> response = invoker.executeNext(requestData);
```

9. Chain of Responsibility Pattern

Location

`com.aiu.trips.chain` - Controller Layer

Components

- **`RequestHandler`** - Abstract handler
- Handler Chain:
  - `AuthenticationHandler` - Validates JWT tokens
  - `AuthorizationHandler` - Checks user permissions
  - `ValidationHandler` - Validates request data
  - `RateLimitHandler` - Enforces rate limits

Purpose

Process requests through a chain of handlers, each handling specific concerns.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-1.3	Token Validation	AuthenticationHandler validates JWT
FR-1.4	Permission Checking	AuthorizationHandler checks roles
FR-6.3	Input Validation	ValidationHandler validates requests
FR-6.4	Rate Limiting	RateLimitHandler prevents abuse
FR-6.5	Security Filters	Chain ensures multiple security checks

Implementation Details

```
RequestHandler chain = new AuthenticationHandler(jwtUtil, userService);
chain.setNext(new AuthorizationHandler())
     .setNext(new ValidationHandler())
     .setNext(new RateLimitHandler());

chain.handle(request);
```

10. State Pattern

Location

`com.aiu.trips.state` - Activity Management

Components

- **`ActivityState`** - State interface
- States:
  - `UpcomingState` - Activity is scheduled
  - `CompletedState` - Activity has finished
  - `CancelledState` - Activity is cancelled
- **`ActivityLifecycle`** - State context

Purpose

Manage activity lifecycle transitions with state-specific behavior.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-2.12	Activity Status Management	States represent activity lifecycle
FR-2.13	Status Transitions	State pattern enforces valid transitions
FR-2.14	State-Specific Behavior	Each state has appropriate operations
FR-2.15	Event Completion	CompletedState handles post-event logic
FR-2.16	Event Cancellation	CancelledState manages cancellation

Implementation Details

```java
ActivityLifecycle lifecycle = new ActivityLifecycle();
lifecycle.initializeFromActivity(activity);
activity = lifecycle.complete(activity);  // Transition to CompletedState
```

11. Strategy Pattern

Location

`com.aiu.trips.strategy` - Booking & Ticketing

Components

- **`PricingStrategy`** - Strategy interface

- Strategies:
  - `StandardPricingStrategy` - No discount
  - `EarlyBirdPricingStrategy` - 15% early booking discount
  - `BulkGroupDiscountStrategy` - 20% discount for 5+ tickets

Purpose

Enable dynamic pricing calculation based on different business rules.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-3.7	Dynamic Pricing	Strategy enables runtime price calculation
FR-3.8	Early Bird Discounts	EarlyBirdPricingStrategy handles date-based discounts
FR-3.9	Group Discounts	BulkGroupDiscountStrategy handles quantity-based discounts
FR-3.10	Flexible Pricing Rules	New strategies can be added without changing booking logic

Implementation Details

```
PricingStrategy strategy = new EarlyBirdPricingStrategy();
BigDecimal finalPrice = strategy.calculatePrice(basePrice, bookingDate, quantity);
```

---

12. Memento Pattern (Bonus)

Location

`com.aiu.trips.memento` + Data Layer

Components

- **Activity Memento:**
  - `ActivityMemento` - Stores activity state
  - `ActivityMementoFactory` - Creates mementos
  - `ActivityHistoryCaretaker` - Manages history
- **Booking Memento:**
  - `BookingMemento` - Stores booking state
  - `BookingMementoFactory` - Creates mementos
  - `BookingHistoryCaretaker` - Manages history

Purpose

Capture and restore object states for undo/redo and history tracking.

Functional Requirements Addressed

FR ID	Functional Requirement	How Pattern Helps
FR-2.17	Activity History	ActivityMemento stores historical states
FR-2.18	Undo Operations	Caretaker enables state restoration
FR-3.11	Booking History	BookingMemento tracks booking changes
FR-3.12	Rollback Support	Memento enables transaction rollback

Implementation Details

```
ActivityHistoryCaretaker caretaker = new ActivityHistoryCaretaker();
ActivityMemento memento = activityMementoFactory.createMemento(activity);
caretaker.saveState(activityId, memento);

// Restore later
ActivityMemento restored = caretaker.getState(activityId, version);
```

Pattern Benefits Summary

Code Quality Improvements

1. **Modularity** - Each pattern encapsulates specific concerns
2. **Maintainability** - Changes are localized to specific components
3. **Testability** - Patterns enable easier unit testing
4. **Scalability** - New features can be added without modifying existing code

SOLID Principles Adherence

- **Single Responsibility** - Each class has one reason to change
- **Open/Closed** - Open for extension, closed for modification
- **Liskov Substitution** - Subtypes are substitutable for base types
- **Interface Segregation** - Clients depend only on interfaces they use
- **Dependency Inversion** - Depend on abstractions, not concretions

Business Value

1. **Faster Development** - Reusable components reduce development time
2. **Reduced Bugs** - Well-tested patterns minimize errors
3. **Better Collaboration** - Common vocabulary improves team communication
4. **Future-Proofing** - System can adapt to changing requirements

---

Conclusion

The implementation of these 11 design patterns (plus Memento) has significantly improved the architecture of the AIU Trips & Events Management System. Each pattern addresses specific functional requirements while contributing to overall system quality, maintainability, and extensibility.

The patterns work together cohesively:

- **Creational patterns** manage object creation complexity
- **Structural patterns** organize relationships between components
- **Behavioral patterns** define communication and responsibility distribution

This design pattern foundation ensures the system can scale and evolve to meet future requirements while maintaining code quality and developer productivity.