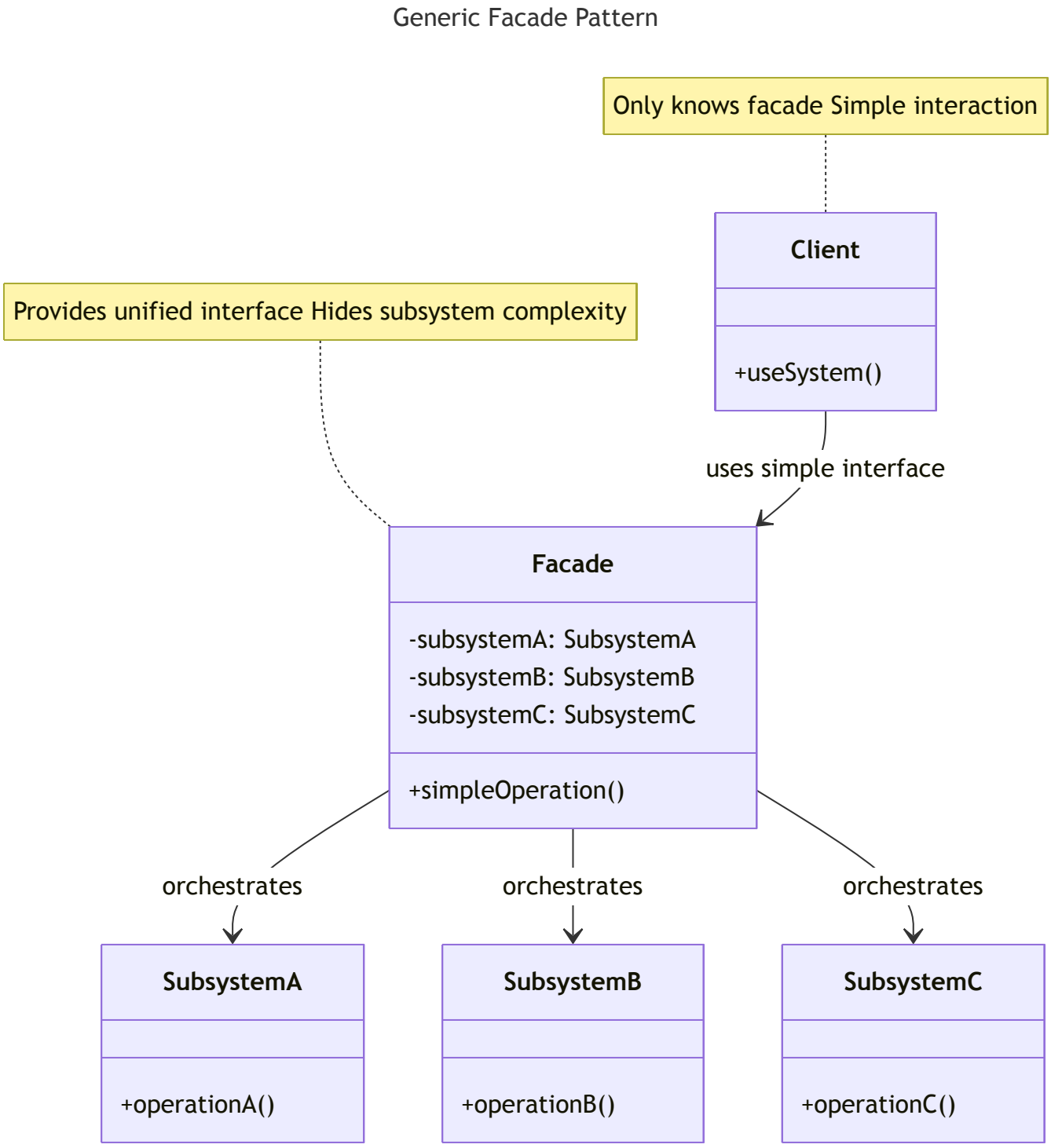
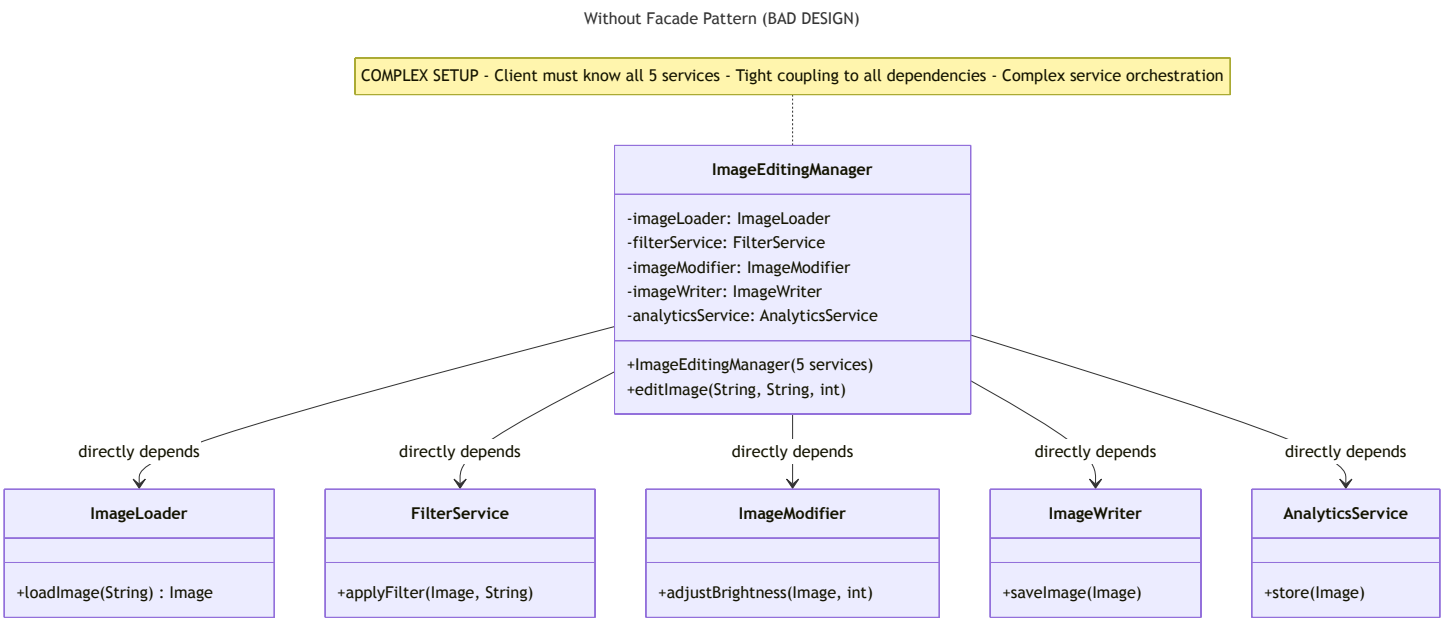
Generic Diagram



Facade Pattern for Image Editing Application

Without Facade Pattern



SOLID Principles Violated Without Facade Pattern

🔴 Dependency Inversion Principle (DIP)

- Client directly depends on all concrete service implementations
- High coupling between client and low-level modules

🔴 Single Responsibility Principle (SRP)

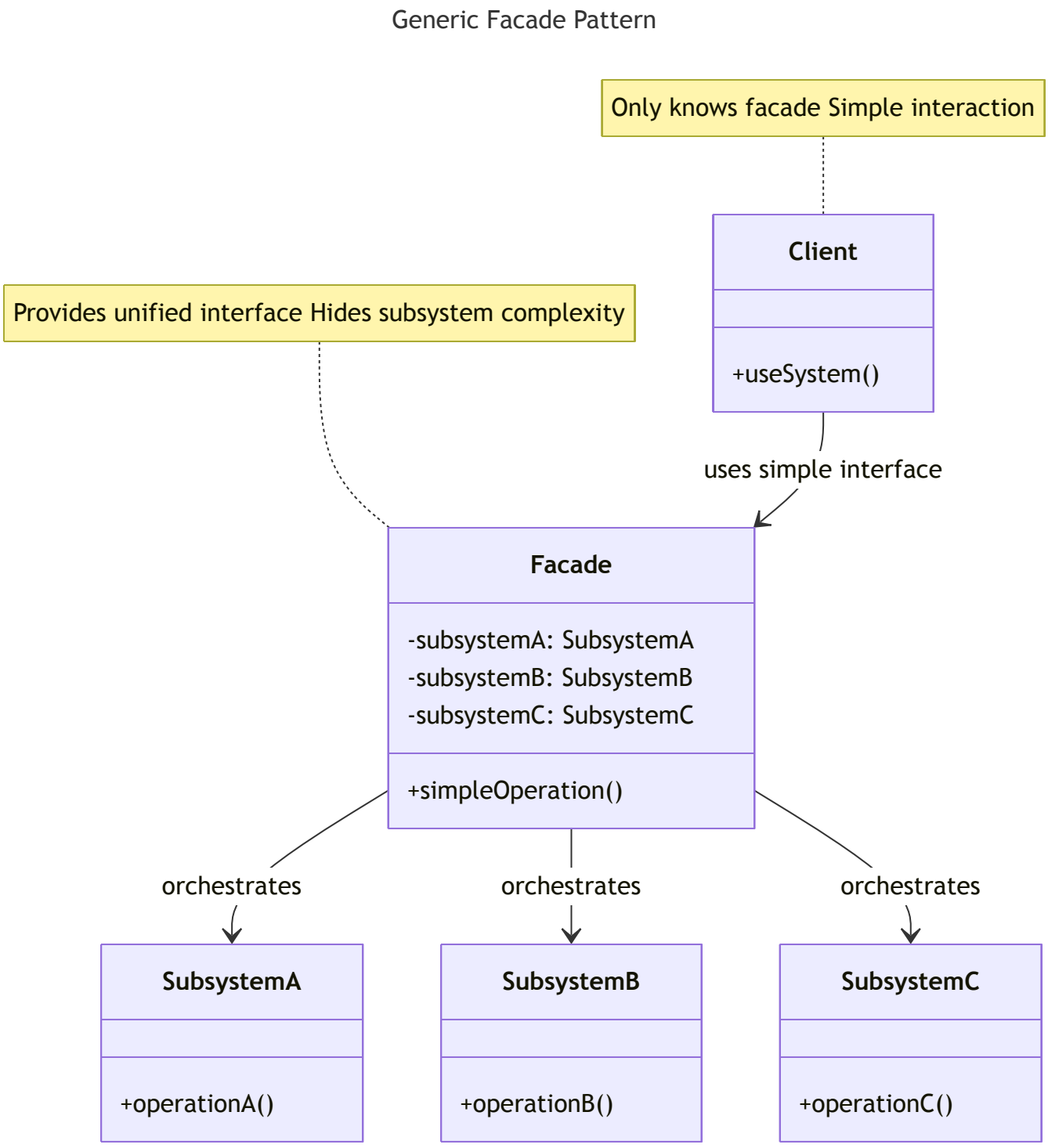
- Client responsible for creating and managing all services
- Mixed concerns: business logic + dependency management

🔴 Open/Closed Principle (OCP)

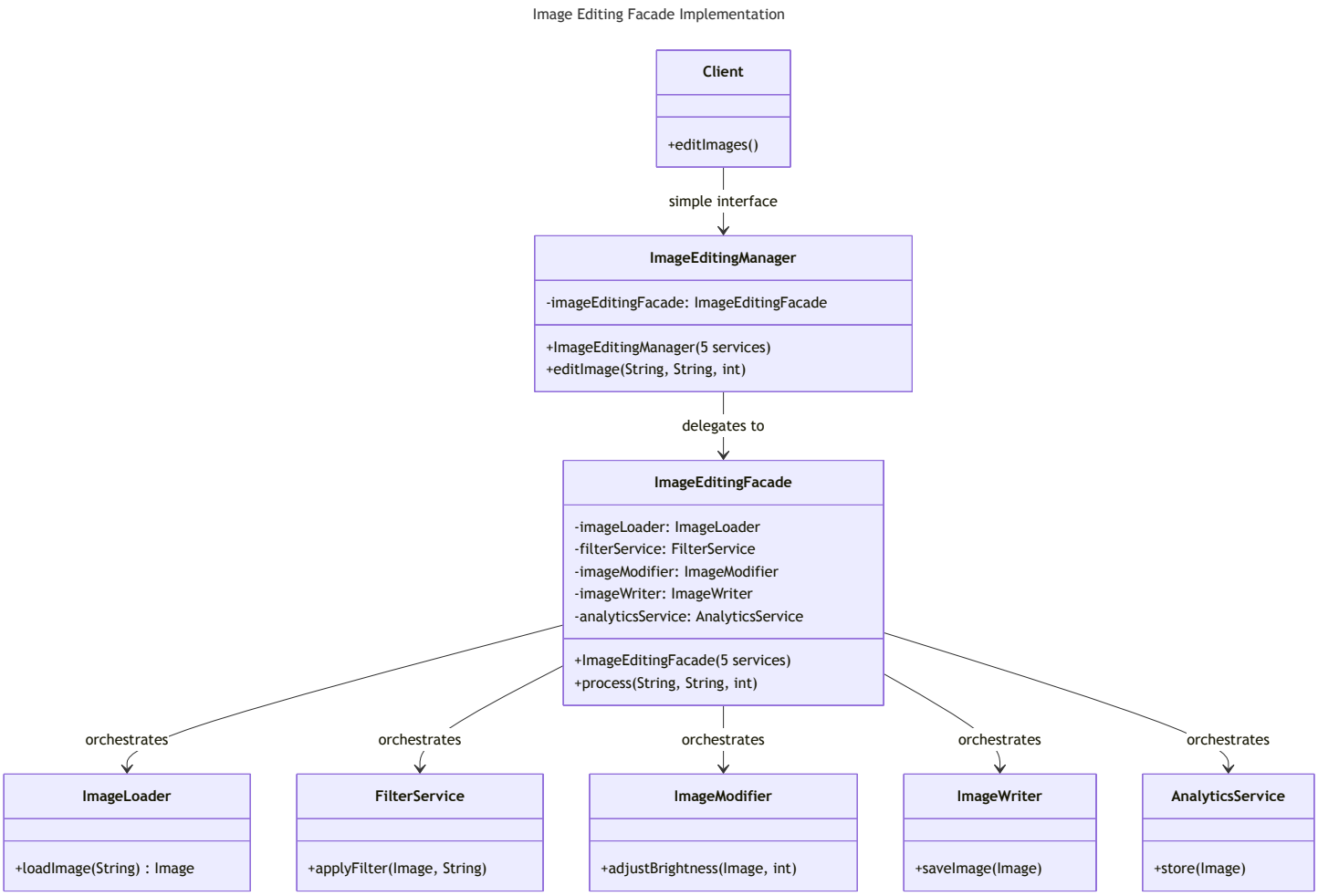
- Adding new services requires modifying client code
- Client must know about internal service structure

With Facade Pattern

Generic Diagram



Specific Diagram



How Facade Pattern Helps Here

Key Benefits:

- **Simplified Interface**: Single point of contact for complex image editing operations
- **Hidden Complexity**: Client doesn't need to know about internal service orchestration
- **Reduced Dependencies**: Client only depends on facade, not individual services
- **Service Coordination**: Facade handles proper sequence of operations (load → filter → modify → save → analytics)
- **Easy Testing**: Can mock facade instead of individual services
- **Maintenance**: Changes to internal services don't affect client code

SOLID Principles Now Followed ✅

- **SRP**: Facade handles service coordination, client handles business logic

- **DIP**: Client depends on facade abstraction, not concrete services
- **OCP**: Can add new services without changing client code

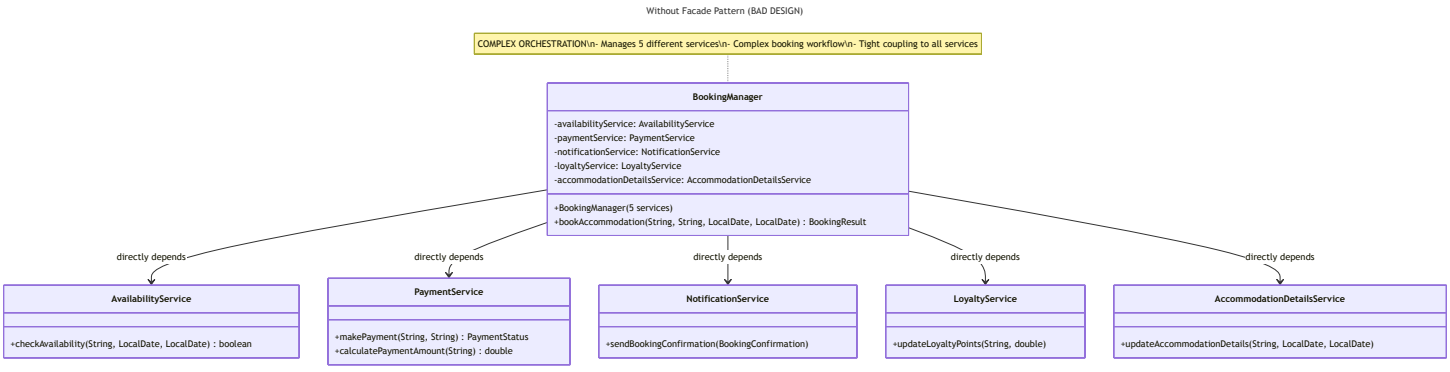
Facade Pattern Flow

1. **Client calls** `manager.editImage()`
2. **Manager delegates** to `facade.process()`
3. **Facade orchestrates** all services in correct sequence:
    - Load → Filter → Modify → Save → Analytics
4. **Complex workflow** hidden from client

The refactored design provides a **clean, simplified interface** while maintaining full functionality and test compatibility!

Facade Pattern for Travel Booking Platform

Without Facade Pattern



```
// BAD - Complex service orchestration exposed
public class BookingManager {
    // 5 service dependencies ❌
    private AvailabilityService availabilityService;
    private PaymentService paymentService;
    private NotificationService notificationService;
    private LoyaltyService loyaltyService;
    private AccommodationDetailsService accommodationDetailsService;

    public BookingResult bookAccommodation(String userId, String accommodationId,
                                           LocalDate checkInDate, LocalDate checkOutDate)
        // Complex workflow with multiple service calls ❌
        boolean isAvailable = availabilityService.checkAvailability(accommodationId, ch

        if (!isAvailable) {
            return BookingResult.notAvailable("Accommodation not available for the give
        }

        PaymentStatus paymentStatus = paymentService.makePayment(userId, accommodationI
        if (paymentStatus != PaymentStatus.SUCCESS) {
            return BookingResult.paymentFailed("Payment failed with status: " + payment
        }

        BookingConfirmation confirmation = new BookingConfirmation(userId, accommodatio
        notificationService.sendBookingConfirmation(confirmation);

        loyaltyService.updateLoyaltyPoints(userId, paymentService.calculatePaymentAmoun
        accommodationDetailsService.updateAccommodationDetails(accommodationId, checkIn

        return BookingResult.success(confirmation);
        // All service coordination logic mixed in one method ❌
    }
}
```

SOLID Principles Violated Without Facade Pattern

🔴 Single Responsibility Principle (SRP)

- BookingManager handles service coordination + business logic + error handling
- One class responsible for multiple service orchestration concerns

🔴 Open/Closed Principle (OCP)

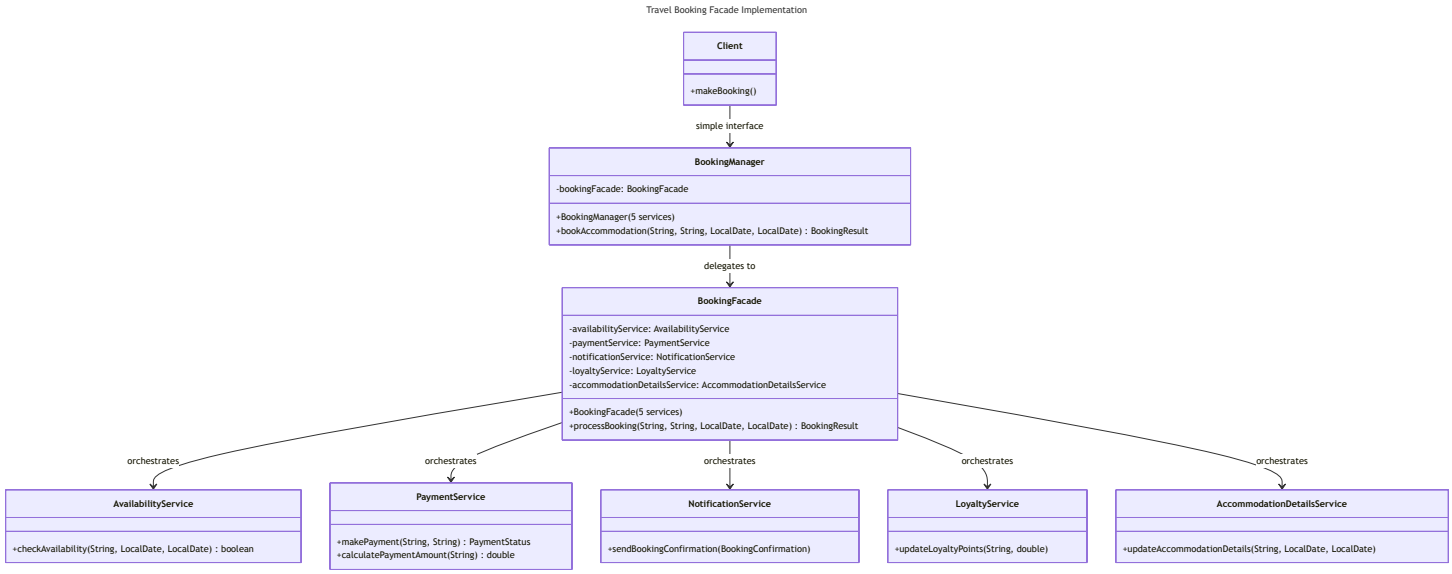
- Adding new services requires modifying BookingManager
- Workflow changes require touching core booking logic

🔴 Dependency Inversion Principle (DIP)

- BookingManager directly depends on all concrete service implementations
- High coupling to low-level service modules

With Facade Pattern

Specific Diagram



How Facade Pattern Helps Here

Key Benefits:

- **Complex Workflow Encapsulation**: Multi-step booking process hidden behind simple interface
- **Service Coordination**: Facade manages the sequence (availability → payment → notification → loyalty → accommodation)
- **Simplified BookingManager**: No longer manages 5 services directly
- **Error Handling Centralization**: All booking-related errors handled in one place
- **Easy Testing**: Mock facade instead of 5 individual services
- **Future Extensibility**: Add booking features without changing BookingManager