

TP1: Advanced Object-Oriented Programming

SOLID Principles & Design Patterns

Room Reservation System - Complete Implementation

Table of Contents

1. Executive Summary
2. SOLID Principles - Detailed Explanations
 - 2.1 Single Responsibility Principle (SRP)
 - 2.2 Open/Closed Principle (OCP)
 - 2.3 Liskov Substitution Principle (LSP)
 - 2.4 Interface Segregation Principle (ISP)
 - 2.5 Dependency Inversion Principle (DIP)
3. Design Patterns Implemented
 - 3.1 Repository Pattern
 - 3.2 Factory Pattern
 - 3.3 Observer Pattern
 - 3.4 Strategy Pattern
 - 3.5 Dependency Injection
 - 3.6 Composite Pattern
4. Architecture Overview
5. Key Features
6. Testing & Quality Assurance
7. Conclusion

1. Executive Summary

This document presents the implementation of a Room Reservation System that demonstrates the SOLID principles and multiple design patterns. The system is organized in a layered architecture with clear separation of concerns.

Metric	Count
Total Classes	25
SOLID Principles Applied	5
Design Patterns	6
Test Cases	20+
Architecture Layers	4

2. SOLID Principles - Detailed Explanations

2.1 Single Responsibility Principle (SRP)

Definition: A class should have only ONE reason to change - it should have only one responsibility or job.

Application in Project: Each class in our system has a focused, single purpose:

- Room classes manage room properties only
- Reservation manages booking data only
- Repository handles data persistence only
- Service orchestrates business logic only
- Validator validates business rules only

Example: The `ReservationService` class is responsible ONLY for orchestrating reservation operations. It doesn't handle storage (that's `Repository`'s job) or notifications (that's `Notifier`'s job). This separation makes the code easier to test, maintain, and modify.

Benefits:

- Easier to understand and maintain
- Changes in one area don't affect others
- Higher cohesion, lower coupling
- Simpler testing (mock single responsibility)

```
class ReservationService: # ONLY orchestrates reservations # Doesn't store
data (Repository does that) # Doesn't send notifications (Notifier does that)
def create_reservation(...): # Business logic only
```

2.2 Open/Closed Principle (OCP)

Definition: Software entities should be OPEN for extension but CLOSED for modification. You should be able to add new functionality without changing existing code.

Application in Project: Our system allows extension through inheritance and interfaces:

- New room types: Extend `Room` class
- New storage: Implement `IReservationRepository` interface
- New notifications: Implement `INotifier` interface
- Factory pattern enables adding new types dynamically

Example: To add a new room type (e.g., `Auditorium`), we create a new class that extends `Room`. The rest of the system works with it automatically through polymorphism - no modifications needed.

Benefits:

- Reduces risk of breaking existing functionality
- Encourages modular design
- Facilitates continuous integration
- Makes codebase more stable

```
# Adding new room type - NO changes to existing code
class AuditoriumRoom(Room):
    def get_equipment(self): return {"stage": True, "sound": True}
    def get_room_type(self): return "Auditorium"
```

2.3 Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of its subclasses without breaking the application. Subtypes must be substitutable for their base types.

Application in Project: All concrete classes can substitute their abstractions:

- Any Room subtype works anywhere Room is expected
- Any IReservationRepository implementation is interchangeable
- Any INotifier implementation works identically

Example: The ReservationService works with ANY room type (Classroom, Laboratory, ConferenceRoom, ComputerLab) without knowing the specific type. Each subtype fulfills the contract of the Room base class.

Benefits:

- True polymorphism
- Flexible, reusable code
- Easier to swap implementations
- Reduces conditional logic

```
# All room types are substitutable rooms = [ Classroom(...),  
ConferenceRoom(...), Laboratory(...) ] # Works with ANY room type -  
polymorphism for room in rooms: service.create_reservation(room, ...)
```

2.4 Interface Segregation Principle (ISP)

Definition: Clients should not be forced to depend on interfaces they don't use. Create specific, focused interfaces rather than one general-purpose interface.

Application in Project: We use small, focused interfaces:

- IReservationRepository: 5 storage methods only
- INotifier: 2 notification methods only
- ReservationObserver: 3 event methods only

Each interface is focused on ONE specific capability.

Example: INotifier interface has only 2 methods: notify_confirmed() and notify_cancelled(). It doesn't include methods for storage, validation, or other unrelated operations.

Benefits:

- Cleaner, more focused interfaces
- Easier implementation
- Better separation of concerns
- Reduces coupling

```
# Small, focused interface - ISP compliant class INotifier(ABC): def
notify_reservation_confirmed(...) def notify_reservation_cancelled(...)
```

2.5 Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces). Depend on interfaces, not concrete implementations.

Application in Project: ReservationService depends on abstractions:

- Depends on IReservationRepository (not InMemoryRepository)
- Depends on INotifier (not EmailNotifier)
- Dependencies injected via constructor

This allows swapping implementations without changing the service code.

Example: ReservationService is injected with repository and notifier interfaces in its constructor. It doesn't create these dependencies itself, and it doesn't care about the concrete implementation.

Benefits:

- Loose coupling between modules
- Easy to test with mocks
- Flexible configuration
- Runtime polymorphism

```
class ReservationService: def __init__(self, repository:
IReservationRepository, notifier: INotifier): self._repository = repository
self._notifier = notifier
```

3. Design Patterns Implemented

Pattern	Purpose	Implementation
Repository	Data access abstraction	IReservationRepository
Factory	Object creation	RoomFactory, etc.
Observer	Event notifications	ReservationObserver
Strategy	Interchangeable algorithms	Multiple notifiers
Dependency Injection	Loose coupling	Constructor injection
Composite	Treat many as one	MultiNotifier

3.1 Repository Pattern

Abstracts data access layer from business logic. The service doesn't care WHERE or HOW data is stored - it just uses the repository interface.

3.2 Factory Pattern

Centralizes object creation logic. Factories handle creation, making it easy to add new types and keeping creation logic in one place.

3.3 Observer Pattern

Event-driven notifications. When reservations are created/cancelled, observers are automatically notified. Implemented for statistics tracking and audit logging.

3.4 Strategy Pattern

Defines family of algorithms and makes them interchangeable. EmailNotifier, SMSNotifier, ConsoleNotifier all implement the same interface with different strategies.

4. Architecture Overview

The system follows a layered architecture with clear separation of concerns:

Layer	Responsibility	Classes
Models	Domain entities	Room, Reservation
Repositories	Data persistence	InMemory, File
Notifications	User alerts	Email, SMS, Console
Services	Business logic	Service, Validators, Factories

Benefits of This Architecture:

- Clear separation of concerns
- Dependencies point toward abstractions (DIP)
- Easy to test with mocked dependencies
- Extensible without modifying existing code (OCP)

5. Key Features

Core Functionality:

- 4 Room Types: Classroom, Conference Room, Laboratory, Computer Lab
- Conflict Detection: Prevents double-booking
- Multiple Storage Options: In-memory and file-based
- Multi-channel Notifications: Console, Email, SMS

Additional Features:

- CLI interface using Rich library
- Advanced validation (business hours, duration limits)
- Real-time statistics via Observer pattern
- Comprehensive error handling

Code Quality:

- Type hints throughout codebase
- Comprehensive docstrings
- Inline comments explaining design decisions
- Professional naming conventions

6. Testing & Quality Assurance

The project includes comprehensive testing to ensure code quality:

Test Category	Coverage
Model Tests	Room creation, equipment, types
Reservation Tests	Creation, cancellation, overlap detection
Repository Tests	Save, find, delete operations
Service Tests	Create, cancel, conflict detection
Validation Tests	Business rules, time ranges
Factory Tests	Object creation patterns

Testing provides confidence in code correctness and enables safe refactoring.

7. Conclusion

This Room Reservation System implements the five SOLID principles through a layered architecture. The system demonstrates:

Core Requirements:

- Application of all SOLID principles
- Multiple room types using inheritance
- Conflict detection mechanism
- Multiple storage implementations
- Multiple notification channels
- Architecture documentation via UML diagrams

Design Patterns: The implementation includes Repository, Factory, Observer, Strategy, Dependency Injection, and Composite patterns, demonstrating their practical application in a reservation management context.

Testing: The codebase includes unit tests covering models, repositories, services, and validation logic to ensure functionality and enable safe refactoring.

Technical Details:

- 25 classes organized across 4 architectural layers
- Interface-based design enabling loose coupling
- Type hints and documentation throughout
- Validation of business rules (hours, duration, capacity)

The architecture follows industry practices for separating concerns and managing dependencies, making the system maintainable and extensible.