Principles of Software Engineering and Data Bases

Davide Yi Xian Hu

Email: davideyi.hu@polimi.it

Date: 10 December 2024

Exercise Lecture: 08 - Design Patterns

POLITECNICO
MILANO 1863

Exercise 1 - Singleton

Design a class that represents a Singleton hospital

```python
# Wrong implementation
class Hospital:
   def __init__(self):
       self.records = []

   def add_record(self, record):
       self.records.append(record)

   def get_records(self):
       return self.records


# Multiple instances
hospital1 = Hospital()
hospital2 = Hospital()

hospital1.add_record("Patient A")
hospital2.add_record("Patient B")

print(hospital1.get_records())  # Output: ['Patient A']
print(hospital2.get_records())  # Output: ['Patient B']
print(hospital1 is hospital2)   # Output: True
```

Exercise 2 - Factory

Design a factory class to create Doctors and Nurses

```python
# Wrong implementation
class Doctor:
   ...


class Nurse:
   ...


staff_type = "doctor"  # Hardcoded logic
if staff_type == "doctor":
   staff = Doctor("Alice", 101, "Cardiology")
elif staff_type == "nurse":
   staff = Nurse("Bob", 102, "Emergency")
else:
   raise ValueError("Invalid staff type")

print(staff.get_details())
```

Exercise 3 - Adapter

Make ExternalNurse compatible with Nurse

```python
class Nurse(Staff):
   def __init__(self, name, specialization):
       self.name = name
       self.specialization = specialization

   def get_details(self):
       return f"Doctor {self.name} - …"


class ExternalNurse:
   def __init__(self, name, department):
       self.name = name
       self.department = department

   def details(self):
       return f"Nurse {self.name} - …"
```

```python
class Hospital:
   def __init__(self):
       self.staff_list = []

   def add_staff(self, staff):
       self.staff_list.append(staff)

   def show_all_staff(self):
       for staff in self.staff_list:
           print(staff.get_details())
```

Exercise 4 - Strategy

Make the following code more open to extensions

```python
# Wrong implementation
class BillingSystem:
    def calculate_bill(self, patient_type, base_charge):
        if patient_type == "insured":
            return base_charge * 0.8  # 20% discount
        elif patient_type == "uninsured":
            return base_charge
        elif patient_type == "government":
            return base_charge * 0.5  # 50% discount
        else:
            raise ValueError("Unknown patient type")

# Client code
billing_system = BillingSystem()

print(billing_system.calculate_bill("insured", 1000))  # Output: 800
print(billing_system.calculate_bill("uninsured", 1000))  # Output: 1000
print(billing_system.calculate_bill("government", 1000))  # Output: 500
```

Exercise 5 - State

In a hospital room booking system, a room can be in different states like Available, Occupied, or Under Maintenance, and the behavior of booking or releasing the room depends on its current state.

```python
# Wrong implementation
class HospitalRoom:
   def __init__(self, room_number):
       self.room_number = room_number
       self.state = "Available"  # Possible states: Available, Occupied, Maintenance

   def book(self):
       if self.state == "Available":
           print(f"Room {self.room_number} booked successfully.")
           self.state = "Occupied"
       elif self.state == "Occupied":
           print(f"Room {self.room_number} is already occupied!")
       elif self.state == "Maintenance":
           print(f"Room {self.room_number} is under maintenance, cannot be booked!")

   def release(self):
       if self.state == "Occupied":
           print(f"Room {self.room_number} is now available.")
           self.state = "Available"
       elif self.state == "Available":
           print(f"Room {self.room_number} is already available!")
       elif self.state == "Maintenance":
           print(f"Room {self.room_number} is under maintenance, cannot be released!")
```

Exercise 6 - Observer

In a hospital scenario, imagine a Hospital Management System that notifies different departments (e.g., Pharmacy, Billing, and Reception) when a new patient is admitted.

Exercise 7 - Tic Tac Toe

Design and implement a Tic Tac Toe game for two players in the terminal using Python.

Design Patterns to Be Used:

- **State Pattern**: Manage the game's state (e.g., in progress, won, draw) and change the game's behavior based on the current state.
- **Observer Pattern**: Notify components (e.g., console or logging systems) about changes to the board.
- **Strategy Pattern**: Encapsulate the logic for player moves, allowing different player types (e.g., human, AI).

Exercise 8 - Snake

Design and implement a simple Snake game in the terminal using Python. The game should demonstrate clean, maintainable, and extensible software design by utilizing different design patterns.

Design Patterns to Be Used:

- **Command Pattern**: Handle player input to change the snake's direction.
- **Singleton Pattern**: Ensure a single instance of the game state, which tracks the grid, snake, and score.
- **Factory Pattern**: Create different types of food with varying effects (e.g., normal food, special food).