Task 1: Research Design Patterns (5 points extra credit)

I participated in the #DesignPattern Slack channel and contributed the following resources/comments:

1

- **Shared a GeeksForGeeks article on the Composite Pattern that provides a clear explanation with diagrams and Java implementation examples. This article helped me understand how to apply the pattern to hierarchical structures like family trees.**
- **Resource: https://www.geeksforgeeks.org/composite-design-pattern/**

2

- **Commented on a discussion about Strategy vs. State patterns, highlighting how Refactoring Guru explains the key differences between these patterns. I found their comparison table particularly helpful for understanding when to use each pattern.**
- **Resource: https://refactoring.guru/design-patterns/strategy/java/example**

3

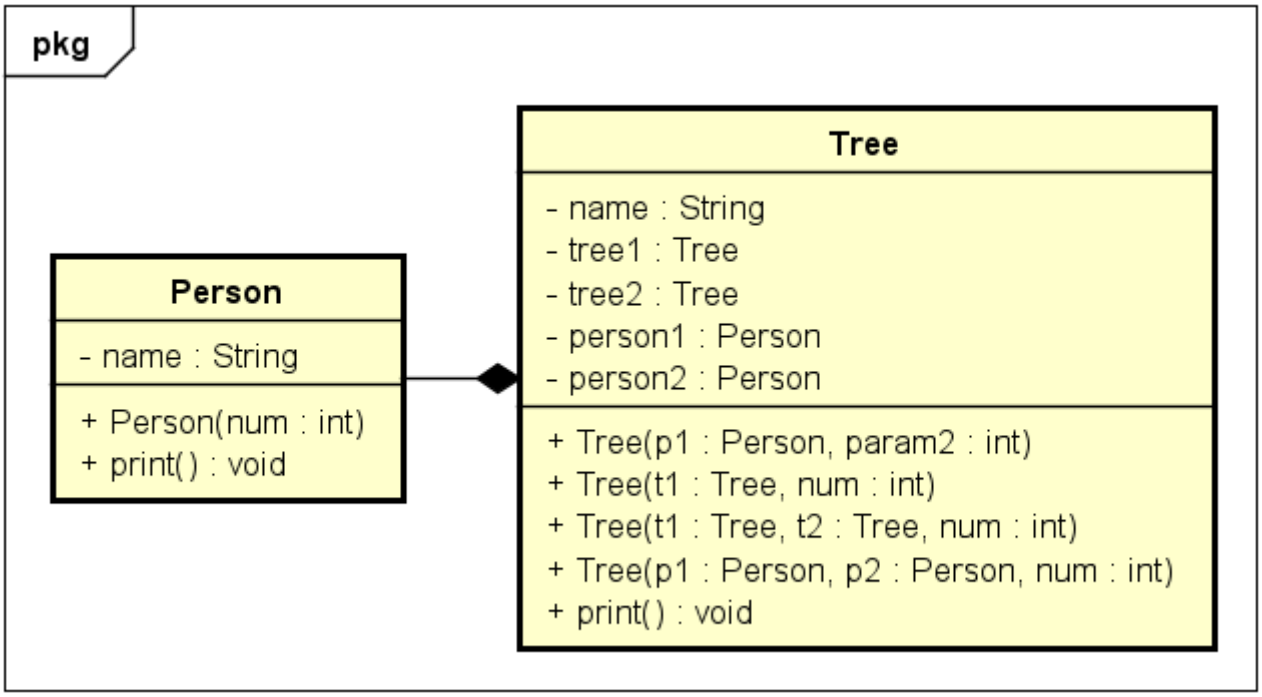
- **Shared a YouTube tutorial from "Christopher Okhravi" explaining the Builder Pattern. His step-by-step coding examples helped me implement this pattern for my assignment.**
- **Resource: https://www.youtube.com/watch?v=9XnsOpjclUg**

Task 2: Understand Design Patterns and their Application

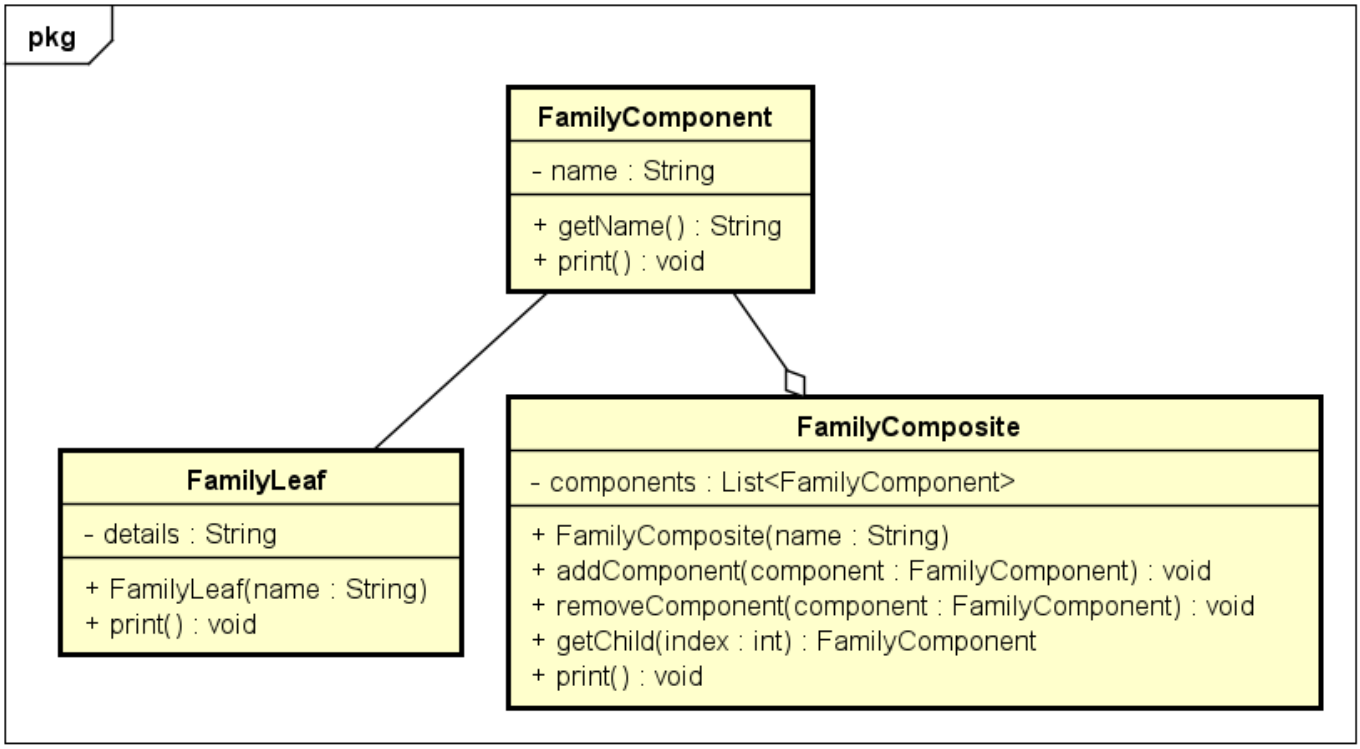
For this task, I chose to apply design patterns to the Tree/Person project and the Character project.

Project 1: Tree/Person - Applying the Composite Pattern

Original UML Class Diagram



Modified UML with Composite Pattern



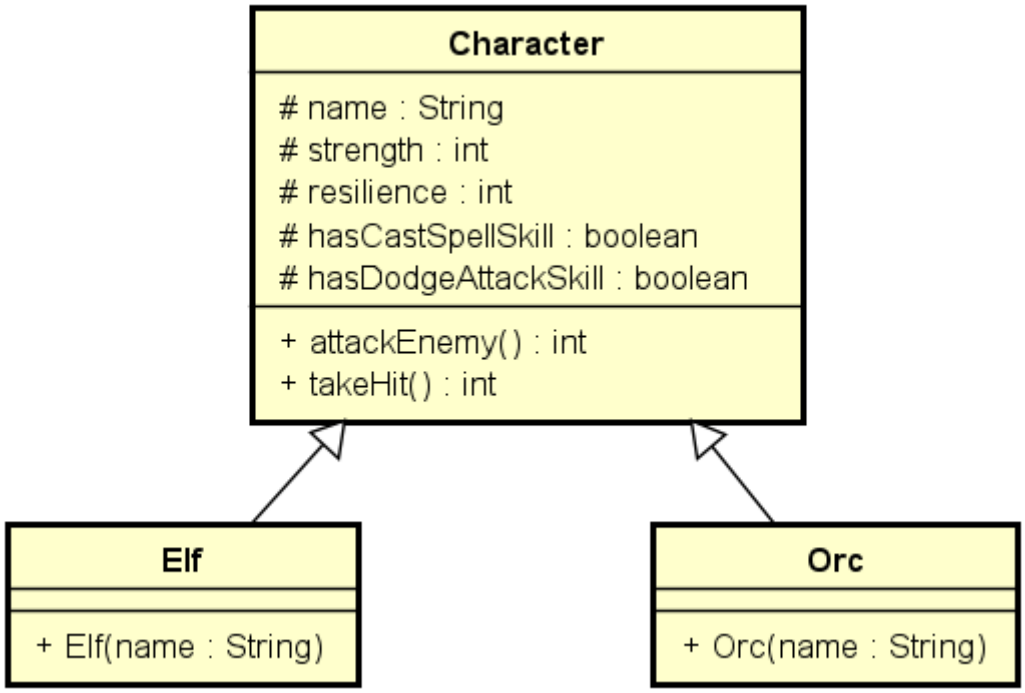
Explanation of Composite Pattern for Tree/Person Project

I chose the Composite Pattern for the Tree/Person project because:

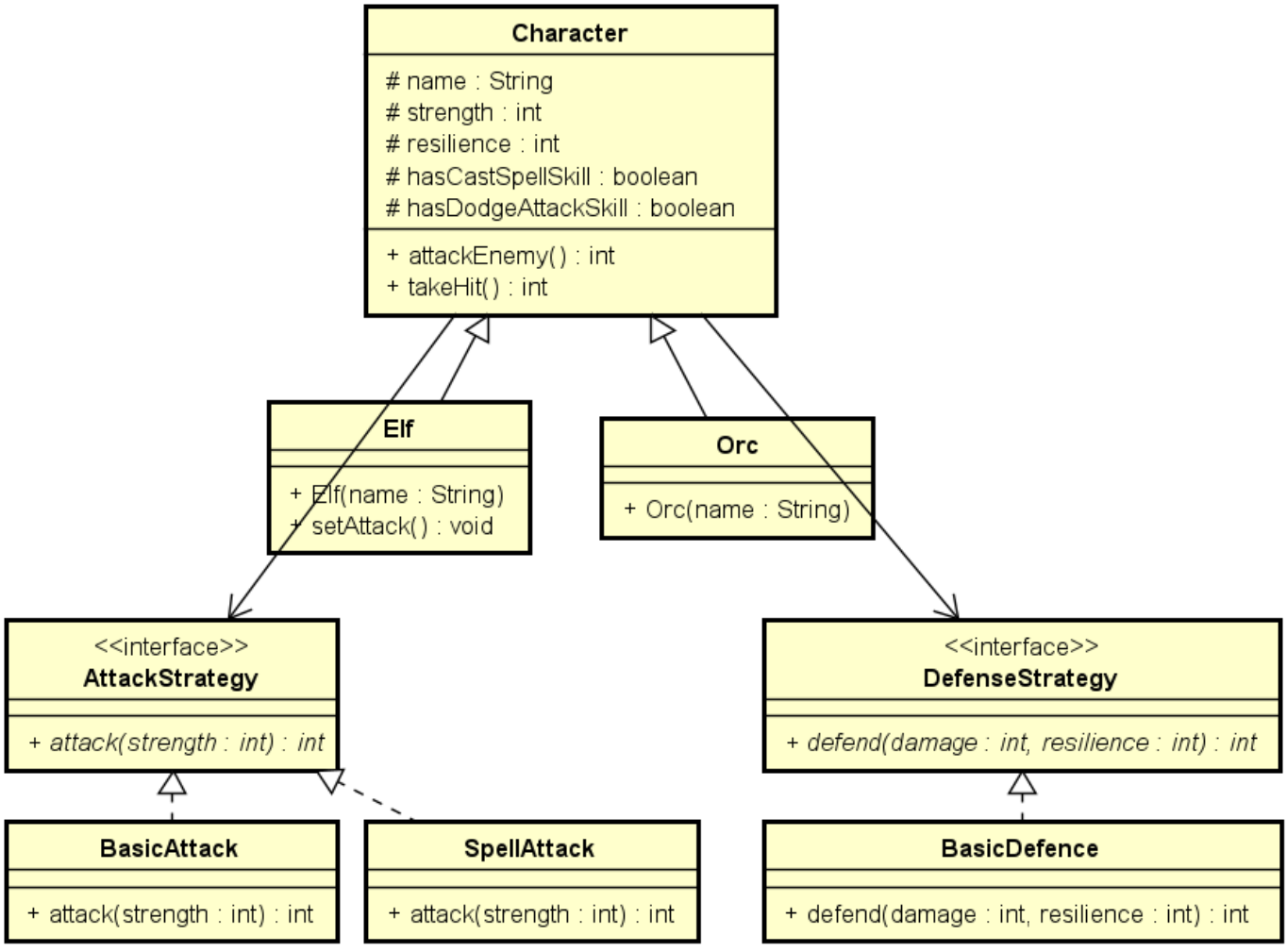
1. **Problem in original design**: The original design has a rigid structure where Tree can only contain exactly two trees or two persons. This limits flexibility and makes it difficult to represent more complex family structures.
2. **How Composite Pattern improves it**: The Composite Pattern allows for a more flexible tree structure where each node (FamilyComposite) can have any number of children (either other composites or leaf nodes). This makes the structure more adaptable to real-world family trees.
3. **Learning**: By applying this pattern, I learned how the Composite Pattern creates a unified interface for both simple and complex objects in a tree structure, making it easier to work with hierarchies.

Project 2: Character/Elf/Orc - Applying the Strategy Pattern

Original UML Class Diagram



Modified UML with Strategy Pattern



Explanation of Strategy Pattern for Character Project

I chose the Strategy Pattern for the Character project because:

1. **Problem in original design**: The original design has attack and defense behaviors hardcoded in the Character class, with conditional logic based on boolean flags (hasCastSpellSkill and hasDodgeAttackSkill). This makes it difficult to add new attack or defense types without modifying the base class.
2. **How Strategy Pattern improves it**: The Strategy Pattern separates the attack and defense behaviors into their own interfaces and implementations. This allows:
    - Dynamic changing of behavior at runtime
    - Adding new behaviors without modifying existing code
    - More maintainable and extensible design
3. **Learning**: By applying this pattern, I learned how the Strategy Pattern enables the encapsulation of algorithms in separate classes, allowing them to be interchangeable at runtime.

Task 3: Apply Two Design Patterns to a System from the Course

For this task, I chose to apply the Builder Pattern and Observer Pattern to the Tutoring System.

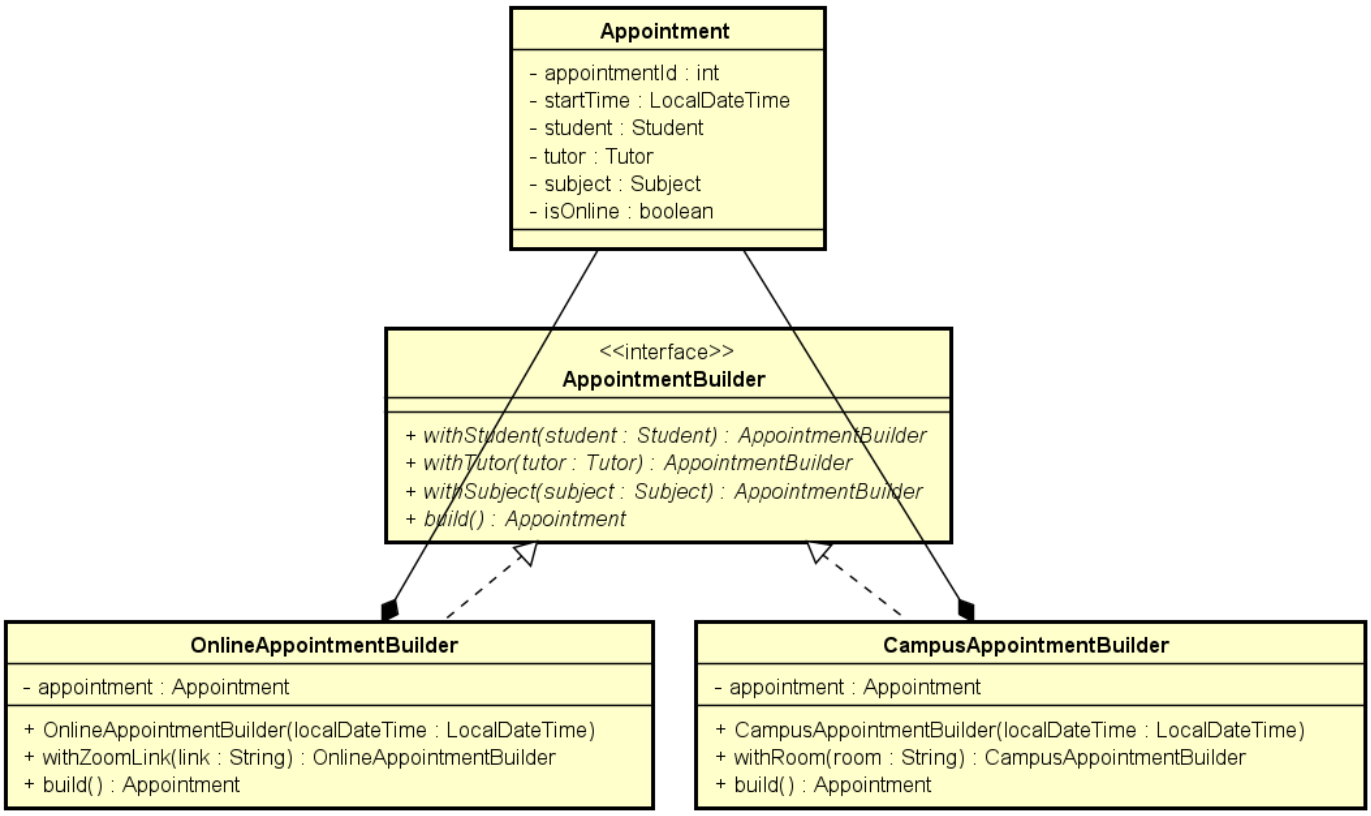
Pattern 1: Builder Pattern for Appointment Creation

Description of Builder Pattern

The Builder Pattern is a creational design pattern that allows for the step-by-step construction of complex objects. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

In this implementation for the Tutoring System, I'm using the Builder Pattern to create appointments. The original system had multiple constructors for different types of appointments (online vs. campus), and setting optional parameters was cumbersome. The Builder Pattern allows for a more flexible and readable way to create appointments with only the parameters needed.

UML Class Diagram for Builder Pattern



Explanation of Application

The Builder Pattern improves the Tutoring System in several ways:

1. **Readability**: Creating appointments becomes more intuitive with method chaining like new OnlineAppointmentBuilder(dateTime).withStudent(student).withTutor(tutor).build().
2. **Flexibility**: We can have different builders for different types of appointments (online vs. campus) that handle specific attributes.
3. **Parameter Management**: We avoid the telescoping constructor problem (having multiple constructors with different parameter combinations).
4. **Validation**: The builder can validate parameters before constructing the final appointment object.

This pattern is especially useful in the Tutoring System because appointments can have many optional attributes (student, tutor, subject, location, etc.) that may or may not be set at creation time.

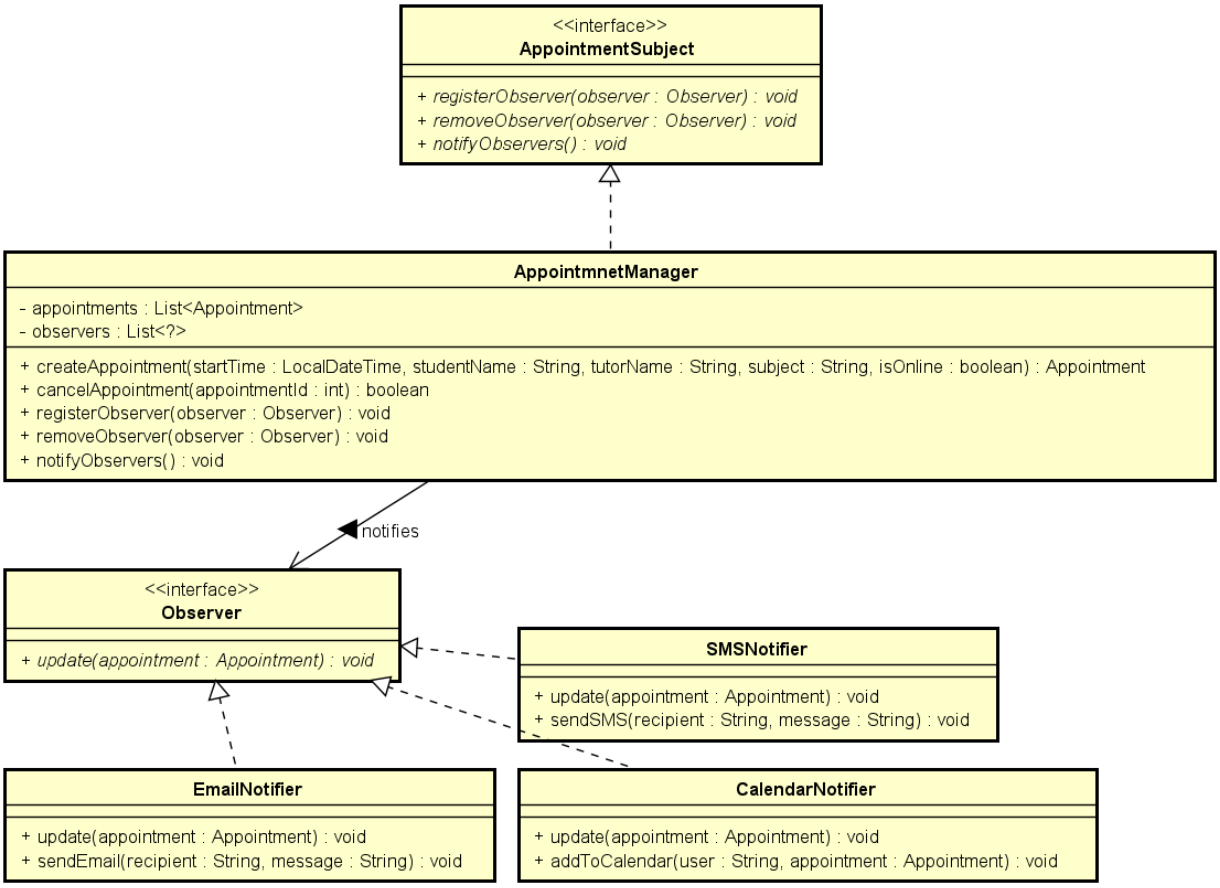
Pattern 2: Observer Pattern for Appointment Notifications

Description of Observer Pattern

The Observer Pattern is a behavioral design pattern where an object (called the subject) maintains a list of dependents (called observers) and notifies them automatically of any state changes, usually by calling one of their methods. This pattern is commonly used to implement distributed event handling systems.

In this implementation for the Tutoring System, I'm using the Observer Pattern to create a notification system for appointment changes. When an appointment is created, canceled, or modified, various observers (like email notifiers, SMS notifiers, or calendar systems) will be automatically notified and can take appropriate actions.

UML Class Diagram for Observer Pattern



Explanation of Application

The Observer Pattern improves the Tutoring System in several ways:

1. **Decoupling**: The appointment management system doesn't need to know the details of notification systems. It simply announces changes, and interested observers respond accordingly.
2. **Extensibility**: We can add new notification types (like mobile app notifications or administrative alerts) without changing the core appointment management code.
3. **Configurability**: Different users can subscribe to different notification methods based on their preferences.
4. **Separation of Concerns**: The appointment management logic is separate from the notification logic, leading to cleaner, more maintainable code.

This pattern is especially useful in the Tutoring System because there are multiple stakeholders (students, tutors, administrators) who need to be informed about appointment changes, and their preferred notification methods may differ.

Conclusion

In this assignment, I've applied and demonstrated four important design patterns:

1. **Composite Pattern** for the Tree/Person project to create a more flexible family tree structure.
2. **Strategy Pattern** for the Character project to separate attack and defense behaviors into interchangeable strategies.
3. **Builder Pattern** for the Tutoring System to provide a flexible way to create appointments.
4. **Observer Pattern** for the Tutoring System to implement a notification system for appointment changes.

Each pattern addresses specific design challenges and improves code flexibility, maintainability, and extensibility by following object-oriented design principles like encapsulation, loose coupling, and separation of concerns.