**Sarah Lorenzen 6-1 Journal: Object-Oriented Programming**

Practical application: **Fantasy Wizard Game Development**

I chose to develop a multiplayer fantasy wizard game as my practical application for Object-Oriented Programming (OOP). This application showcases OOP concepts with:

* Wizards, magical objects, and spells naturally translate into classes and objects
* Player actions, item usage, and spell casting require polymorphic behavior
* Different types of wizards and magical items benefit from inheritance
* Player health, magical power, and inventory require proper encapsulation
* The game can easily expand with new wizard types, spells, and magical objects

**Classes and Objects:**  
**Classes** are blueprints that define the structure and behavior of entities. **Objects** are instances of classes, which are the actual entities that exist in our program, each with specific attribute values.

In this game:

* The Player class defines what all wizard players have in common
* Each wizard object represents a specific player with unique attributes

**Practical Benefits:**

**For End Users**: Players interact with distinct game entities (their wizard, their wand) that maintain consistent behavior and state.

**For Developers**: Classes provide reusable templates, reducing code duplication. Wizard behavior is defined once and then can create multiple player instances.

**For Program Structure**: Objects encapsulate related data and methods together, creating logical, maintainable code units. Performance benefits include efficient memory management through shared class methods.

**Inheritance**

**Inheritance** allows classes to inherit attributes and methods from parent classes, creating hierarchical relationships. Child classes extend or modify parent behavior while reusing common code.

In the game:

* MagicalObject is a base class for all magical items
* Wand and CrystalOrb inherit from MagicalObject
* Specialized wizard types can inherit from the base Player class

**Practical Benefits**

**For End Users**: Player’s gameplay experience is varied with different item types that share common interfaces but have unique abilities.

**For Developers**: Inheritance eliminates code duplication. Common item behavior is defined once in the parent class, while specialized behavior is added in child classes.

**For Program Structure**: Creates logical hierarchies that mirror real-world relationships. Inheritance promotes code reuse and makes the codebase easier to extend with new item types.

**Encapsulation**

**Encapsulation** bundles data and methods within a class and restricts direct access to some components. It uses access modifiers to hide internal implementation details.

In the game:

* Player attributes like \_health and \_magical\_power are private
* Public methods like boost\_health() and get\_health() control access to attributes
* Prevents invalid state changes (like negative health)

**Practical Benefits**

**For End Users**: Ensures game rules are enforced consistently. Players can't accidentally break the game by setting invalid values (like negative health or exceeding maximum power).

**For Developers**: Protects internal implementation. We can change how health is stored without breaking code that uses the class. Provides clear interfaces for interacting with objects.

**For Program Structure**: Reduces bugs by preventing invalid states. Improves maintainability by creating clear boundaries between public interface and private implementation. Adds minimal performance overhead with significant reliability benefits.

**Polymorphism**

**Polymorphism** allows objects of different classes to be treated as objects of a common parent class. The same method name can behave differently based on the object type calling it.

In the game:

* All MagicalObject subclasses implement the use() method differently
* Wands boost magical power, while Crystal Orbs boost both health and magical power
* The Spell hierarchy demonstrates different spell behaviors with the same interface

**Practical Benefits**

**For End Users**: Players interact with different magical objects using the same action ("use"), but each item provides unique effects. This creates intuitive, consistent gameplay with variety.

**For Developers**: Write code that works with any MagicalObject without knowing its specific type. Adding new item types doesn't require changing existing code that uses items.

**For Program Structure**: Enables flexible, extensible systems. New spell types or items can be added without modifying core game logic. Promotes loose coupling and high cohesion.