6-1 Journal: Explore Object-Oriented Programming

Marie-Chantal Foster

Cybersecurity, Southern New Hampshire University

CS-500-10018-M01 Introduction to Programming

Dr. Shadha Tabatabai

December 21, 2025

Explore Object-Oriented Programming

A fundamental paradigm in modern software development is object-oriented programming (OOP), which enables developers to create reusable, scalable, and efficient applications (Zybooks, 2025). Organizing code into modular, maintainable components makes it ideal for game development. It helps manage and simplify code complexity by modeling game elements, such as characters, game mechanics, and objects, with their own attributes and behaviors (Rihnafi, Sohail, & B, n.d.). I was drawn to its uses in game design because one of my favorite games is Mortal Kombat, which I still enjoy whenever time permits.

To make OOP work effectively in game design, development begins with *classes* and *objects*; they are the building blocks of the game world. A *class* is the blueprint that defines shared attributes, such as health and speed, as well as behaviors, such as `attack()` and `defend(),` for all characters in the game. In Zybooks (2025), an *object* is an instance of a *class,* such as a specific fighter in a match. This structure sets the stage for applying the OOP pillars, each of which supports scalable, modular development. The first pillar is *Inheritance*, which lets you reuse and extend those blocks. It is what you call a “is-a” or “has-a” relation between levels. We can view the relationship as a parent-child one, meaning the child inherits from the parent (Zybooks, 2025). The *class* can inherit from another *class.* In a game, A `fighter` object inherits everything from `Character`. The second pillar, *encapsulation,* protects and organizes the data by keeping member variables private and exposing only controlled methods for interaction. This principle, termed information hiding, ensures that sensitive values, such as the fighter’s health, cannot be altered directly. Therefore, the game interacts with these values through safe methods such as `takedamage()`and `gethealth()`, which prevent unintended

changes and maintain game balance (Navodya, 2025). The third pillar, *polymorphism,* adds flexibility by allowing different objects to respond uniquely to the same method, keeping the call the same while the behavior varies by object. When calling `playsound()` during an attack, it triggers different audio depending on the fighter: Scorpion's punch would produce a fiery impact, while Subzero's would produce an icy crack (Navodya, 2025).

The benefits of OOP concepts for game design applications for end users, developers, and program structure and performance, based on the Mortal Kombat (MK) scenario. Effects on end user: Gameplay experience is enhanced by ensuring clarity, variety, and engagement. Since the player never interacts with the blueprints, they benefit from the developer's object-oriented structure. The Character class defines what a fighter can do by specifying shared attributes like health and speed, as well as behaviors such as attack and defend. When a player chooses a fighter, e.g., Scorpion, they select an object of that class. Because each object is built from the same, all fighters follow consistent rules and mechanics. *Inheritance* provides uniformity across different entities. If a player learns how to interact with one object (e.g., a Scorpion's spear), they will intuitively know how to interact with all other objects (e.g., a Cub) because the "is-a" relationship ensures continuity. E*ncapsulation* ensures data integrity by ensuring that a fighter's internal state, such as health, is managed strictly by the game's core rules, preventing unintended glitches or unauthorized changes. *Polymorphism* allows different objects to respond to the same command in unique ways, enabling players to use a single interaction button to trigger "talk" or "pick up" behavior (Navodya, 2025; Premasiri, 2020). Effects on the developer: *Classes* such as `character`, `specialmove`, and `arena` define the structure of the game world, and each *object*, e.g., Scorpion, is an instance of these classes, creating a predictable and manageable system (Premasiri, 2020). *Inheritance* reduces workload by allowing new fighters to inherit shared

attributes and behaviors from their `character` class, while *encapsulation* prevents accidental changes when multiple developers work on the same codebase; therefore, it supports teamwork among developers when refining and designing characters (Premasiri, 2020). *Polymorphism* allows methods like `specialmove()` to behave differently depending on the character, e.g., Scorpion's version triggers a spear, making the code easier to extend and update. By modeling data and behaviors after real-world interactions, developers gain a more intuitive framework for creating meaningful, reusable game components (Rihnafi, Sohail, & B, n.d.). Effect on the program structure and performance: MK supports a clean, modular architecture to improve performance and maintainability. The system begins by defining core *classes*, such as the `character` class, in which individual fighters, like Scorpion, are created as *objects*. *Inheritance* eliminates redundant code, improves performance, and reduces memory usage, since the game engine does not need separate movement logic for each fighter, as all fighters share a common base implementation. *Encapsulation* ensures each class controls its own data, thereby preventing bugs caused by unintended interactions between systems (Navodya, 2025). *Polymorphism* streamlines performance by allowing the game engine to call the same method name across all characters, e.g., `performattack()` or `specialmove()`, while each object responds with its own behavior (Navodya, 2025; Premasiri, 2020). This structure improves debugging because issues can be traced to specific classes, such as `collision()`, rather than requiring a search of the entire codebase (Mohov, S. (n.d.).

OOP demonstrates how game design's modular complexity can be managed through *Classes* and *objects,* which form the foundation; *inheritance,* which enables reusability; *encapsulation*, which protects data; and *polymorphism*, which adds flexibility. These concepts support scalability and maintainability in modern gaming design.

References

Navodya, M. (2025, June 5). *The Power of OOP in Game Development. Medium.* https://medium.com/@mikaznavodya/the-power-of-oop-in-game-development-f5de5300b739

Premasiri, T. (CodeLikeMe). (2020, March 28). Lecture 7 - Object-Oriented Principles in Game Development. YouTube [video]. https://youtu.be/OspQ-T1q5jo?si=hIW5YgzyaDSFVteK

Rihnafi, A., Sohail, H., B, H. (n.d.) *What is procedural programming and how does it differ from object-oriented programming in game development?* https://www.linkedin.com/advice/0/what-procedural-programming-how-does-differ-from-gkocc

ZyBooks. (2022). Introduction to programming. Wiley. http://www.zybooks.com