

**CPT304 Software Engineering II**

Beyond the Silver Bullet: Addressing Key Software Challenges with Builder and Strategy Patterns

**Assignment 1**

**Group member:**

Haojie Xu 2144434

Linfeng Li 2145589

Huayuan Zheng 2145549

Zirui Zhu 2141991

Ganmin Wang 2144415

1. **Introduction**

In the field of software engineering, people often hope to find a universal and efficient method to rapidly solve the wide array of problems encountered during development. However, as Brooks famously argued in *No Silver Bulle*t, there is no single breakthrough, or silver bullet that can dramatically enhance software productivity and quality [1]. The most significant obstacles in software development are not the accidental difficulties arising from tools or programming languages, but the essential difficulties inherent to the nature of software itself.

Brooks identifies four such key challenges—complexity, conformity, changeability and invisibility—which remain persistent barriers in software engineering [1]. This report focuses on two of these: complexity and changeability. We will explore how design patterns, specifically the Builder Pattern and Strategy Pattern, can be applied to help address these challenges. Through a case study analysis, we will demonstrate how these patterns can effectively mitigate the intrinsic complexity and frequent change found in modern software systems. Ultimately, this report aims to show that while there is no single silver bullet, adopting appropriate design patterns can significantly improve software development practices.

1. **Key challenges**

Before examining design patterns in depth, we will first review the key challenges outlined in No Silver Bullet to provide a conceptual foundation for our discussion.

Brooks draws a critical distinction between accidental difficulties and essential difficulties, a classification that echoes Aristotle’s distinction between the accidental and the essential. Accidental difficulties arise from the tools, languages and environments used in software development. They are problems created by engineers and, in principle, solvable by engineering advancements. For example, the invention of high-level programming languages such as Ada, which introduced the philosophy of modularization, abstract data types, and hierarchical structuring, significantly reduced accidental complexity. However, as Brooks notes, the remaining accidental challenges are relatively minor, and the productivity payoff from further eliminating them is limited [1].

In contrast, essential difficulties stem from the nature of the problem being solved. They cannot be eliminated by technical improvements alone. For instance, when a user requests thirty distinct features, these requirements define the essence of the system—they are part of the conceptual task itself, not byproducts of implementation. Brooks argues that although there is no one silver bullet, a series of coordinated innovations that address these essential difficulties could lead to significant progress [1]. Indeed, this is where most modern software engineers now spend most of their effort.

This report adopts Brooks’s perspective and identifies the four essential difficulties—complexity, conformity, changeability, and invisibility—as the key challenges in contemporary software engineering.

* **Complexity** arises from the intricate interdependencies within software systems. Unlike physical structures, software lacks repetitive components, leading to an exponential growth in complexity as systems scale.
* **Conformity** refers to the need for software to adapt to arbitrary and often inconsistent external constraints, such as hardware, protocols, or organizational requirements, which software must conform to without simplifying them.
* **Changeability** reflects the constant demand for software to evolve with new user needs, technologies, and environments. Unlike physical products, software is endlessly malleable and thus expected to change continuously.
* **Invisibility** highlights the abstract, intangible nature of software. Unlike blueprints or circuit diagrams, software lacks a natural visual representation, making it difficult to fully grasp or communicate its structure.

1. **Design pattern**

In *No Silver Bullet*, Brooks envisions the potential of expert systems to encapsulate expert knowledge and assist developers in making informed design decisions through inference mechanisms [1]. While expert systems focus on automation and reasoning, a similar philosophical approach can be observed in the emergence of design patterns.

Formally introduced by the “*Gang of Four*” in 1994, design patterns serve as documented, reusable solutions to common software design challenges [2]. Unlike expert systems, which depend on inference engines, design patterns rely on the developer’s intuition and experience to identify and apply the appropriate structure during the design phase.

In the following content, we will examine the Builder and Strategy design patterns, which are widely used and particularly relevant to the challenges of complexity and changeability, as outlined by Brooks.

**3.1 Builder Pattern**

The Builder pattern is a creational design pattern that separates the construction of a complex object from its representation, allowing the same construction process to create different representations. The following Figure 1 illustrates the structure of the Builder pattern. As described by Gamma et al. in *Design Patterns* [2], the Builder pattern is particularly useful when an object requires numerous steps to be instantiated and assembled, especially when some components are optional or need to vary across contexts.

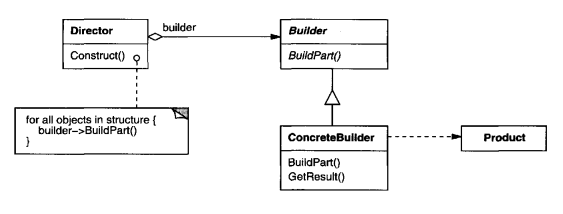


Figure 1. UML class diagram for the builder pattern

As Chung analogizes, building a house reflects the Builder Pattern: the client expresses their requirements to a contractor (Director), who then delegates the actual construction work to specialized professionals (Builders), such as plumbers, electricians, and carpenters [3]. This separation of responsibilities makes the construction process more flexible, modular, and controllable. It decouples construction logic from product representation and facilitates reusable, maintainable object creation.

The Builder pattern is widely used in frameworks and libraries. A notable enterprise-level example is the use of HttpClientBuilder in Apache HttpComponents, which allows developers to fluently build HTTP clients with a variety of configurations [4]. Additionally, in data warehouse engineering, the metadata-driven builder pattern is used to systematically construct ETL pipelines, where metadata acts as the blueprint and builder classes dynamically generate components such as data extractors, transformers, and loaders [5].

**3.2 Strategy pattern**

The Strategy pattern is a behavioral design pattern that defines a family of algorithms, encapsulates each one as a separate object, and allows them to be interchangeable at runtime. The following Figure 2 illustrates the structure of the Strategy pattern. Gamma et al. state that the Strategy pattern enables the selection of an algorithm’s behavior without altering the code that uses it [2]. This pattern is particularly useful in scenarios where multiple variants of an operation exist and need to be swapped dynamically depending on context or user input.

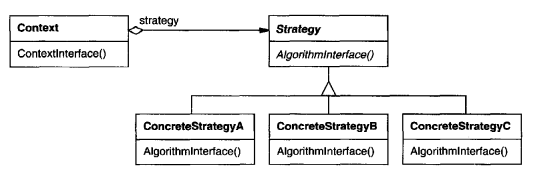


Figure 2. UML class diagram for the strategy pattern

The Strategy pattern offers several notable advantages in the design of flexible and maintainable software systems. One of the main benefits of the Strategy pattern is that it promotes adherence to the Open/Closed Principle, allowing new strategies to be introduced without modifying existing code. It also eliminates the need for complex if-else or switch statements, leading to cleaner and more maintainable code. Additionally, because each strategy is encapsulated in its own class, the pattern makes unit testing and debugging easier. The context class becomes simpler and focuses solely on delegating tasks to the selected strategy, improving overall code readability and flexibility.

The Strategy Pattern is widely used in modern software systems. For instance, in e-commerce platforms, different payment methods (credit card, PayPal, bank transfer) can be implemented as strategies that conform to a common payment interface. This allows the system to switch payment providers without changing the core logic [6]. Another common use is game development, strategy-based AI behaviors (aggressive, defensive, evasive) are implemented using this pattern to create modular, interchangeable behaviors [7].

1. **Case study analysis**

Game development is a software engineering domain where Brooks’s essential difficulties—particularly complexity and changeability—are especially evident. Intense market competition imposes tight deadlines, often leading to rushed design and poor maintainability. This makes thoughtful design decisions critical for long-term adaptability [8]. In this context, applying design patterns such as Builder and Strategy becomes essential. This report uses an RPG game case study to show how these patterns help manage object construction complexity and support evolving gameplay behaviors.

**4.1 Addressing Complexity**

Brooks emphasizes that software complexity emerges from the intricate and nonlinear interactions between distinct components [1]. This complexity increases cognitive load and hinders effective communication within development teams. In the context of RPG development, character creation is a clear example. A character consists of multiple interrelated attributes such as profession, name, weapon and armor which are shown in figure 3. Many of these attributes are optional or dependent on specific conditions. Without a well-structured approach, the object construction process becomes convoluted and error-prone. This leads to reduced clarity in design and creates barriers for new developers to understand the system.

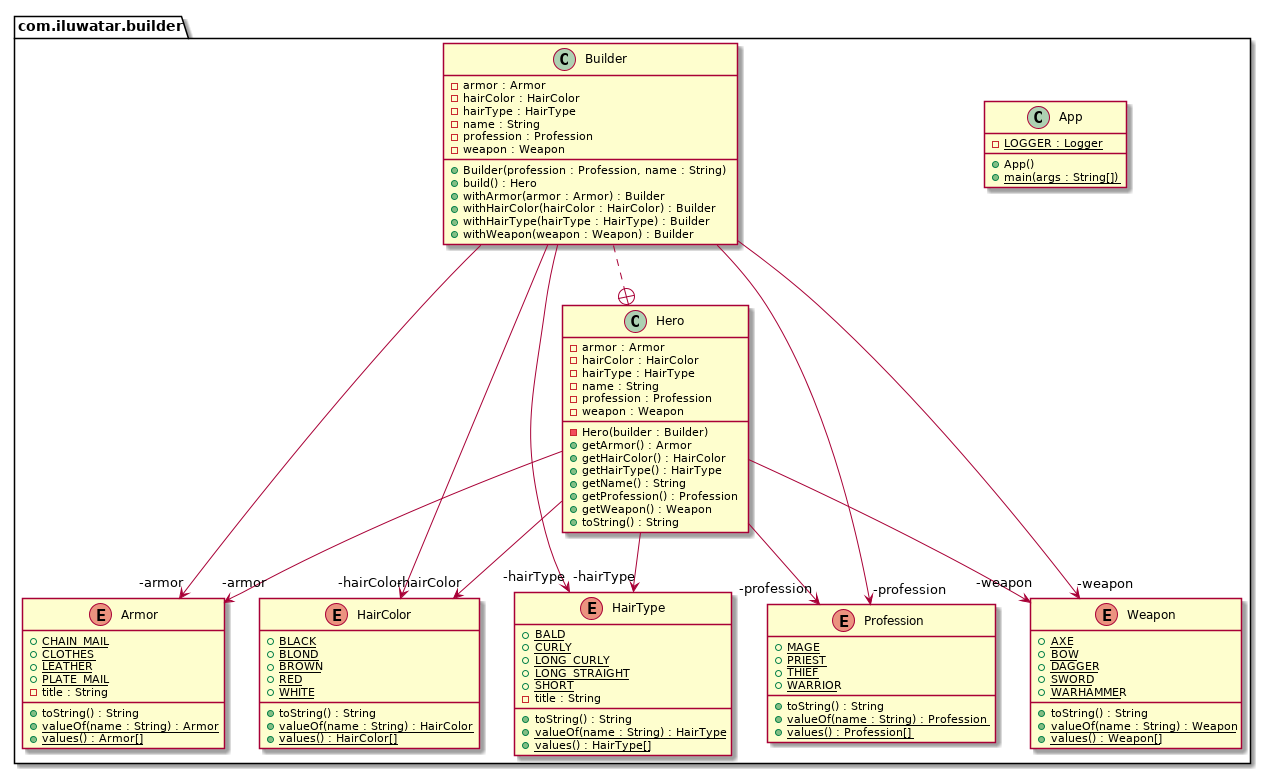


Figure 3. UML class diagram for the builder pattern in game

The Builder pattern offers an effective solution by decoupling the process of object construction from its representation. It allows character objects to be assembled step by step in a modular and consistent manner. Through a fluent and readable interface, it ensures the logical order of construction while making it easier to introduce new character features without disrupting existing code. The figure 4 shows the details of the sample code. This modularity simplifies the mental model required to work with the object structure and improves collaboration across the team.

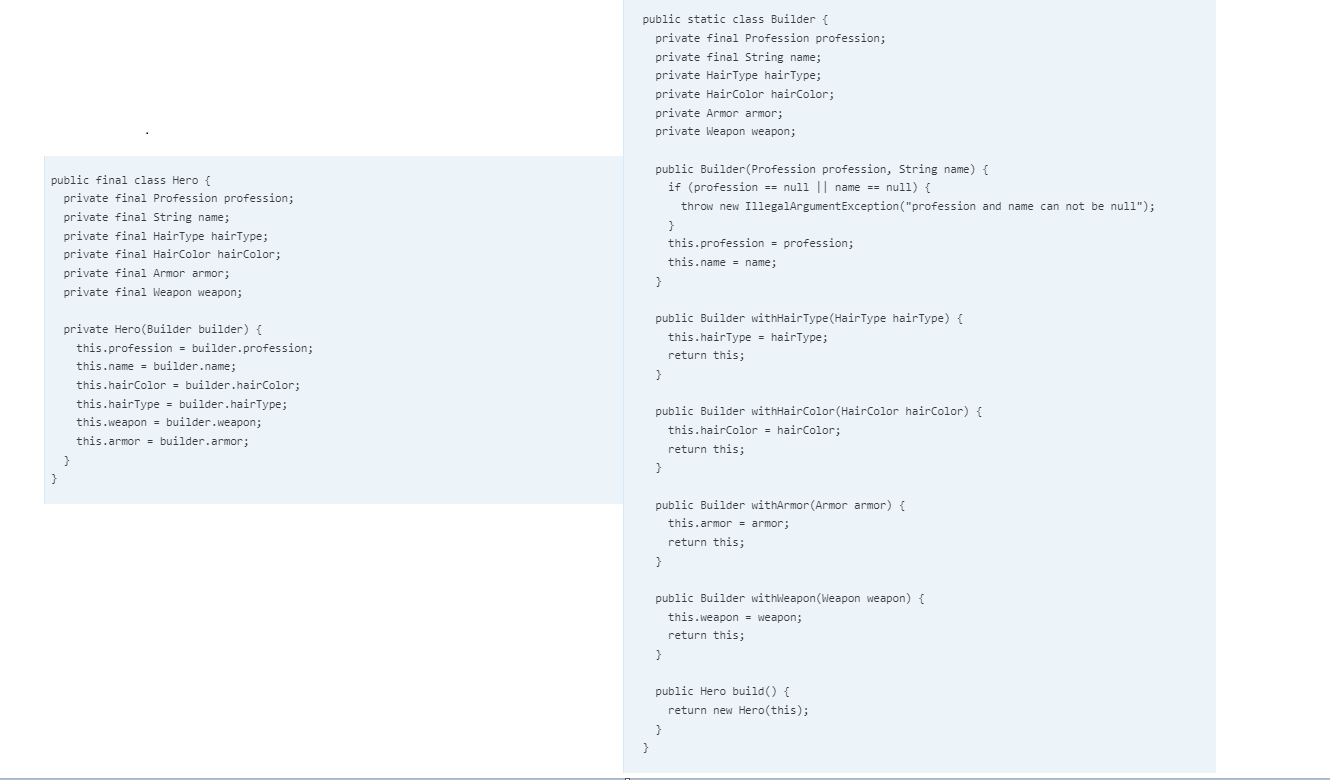


Figure 4. The sample code of the builder pattern

**4.2 Addressing Changeability**

According to Brooks, software must constantly adapt because it embodies functionality, which is the aspect most susceptible to evolving requirements [1]. In game development, this often manifests through the introduction of new combat mechanics, AI behaviors, or skill types as player expectations shift. These additions reflect a common source of change, where users seek to apply the software beyond its original scope. When all behavior is tightly integrated into a single character class, the system becomes rigid and increasingly difficult to modify or extend.

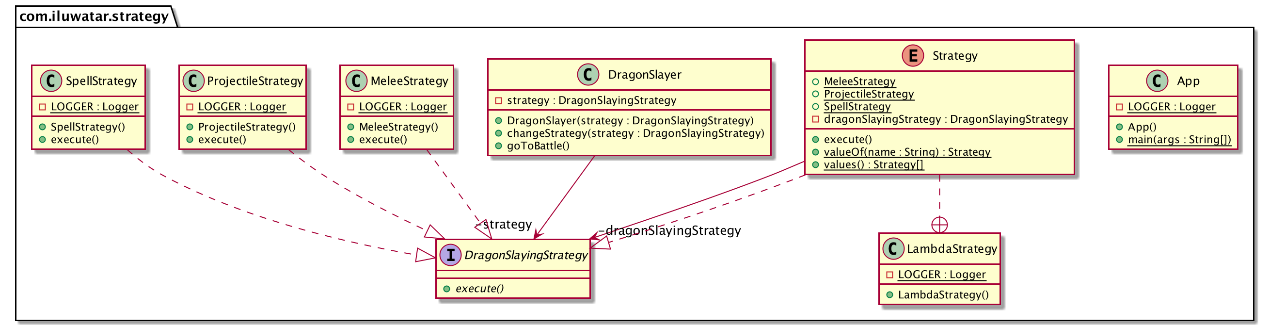


Figure 5. UML class diagram for the strategy pattern in game

The Strategy pattern addresses this issue by encapsulating each combat skill or behavior as a separate strategy class. Character objects delegate actions to a selected strategy, allowing different abilities such as melee, projectile or spell shown in figure 5 to be applied dynamically at runtime. The figure 6 represents the detailed implementation code in this background. This approach removes the need for hardcoded conditionals and supports flexible extension without altering the core system. By separating behavior from the object structure, the Strategy Pattern supports maintainability and aligns with the Open/Closed Principle, making it an ideal solution for managing ongoing functional changes.

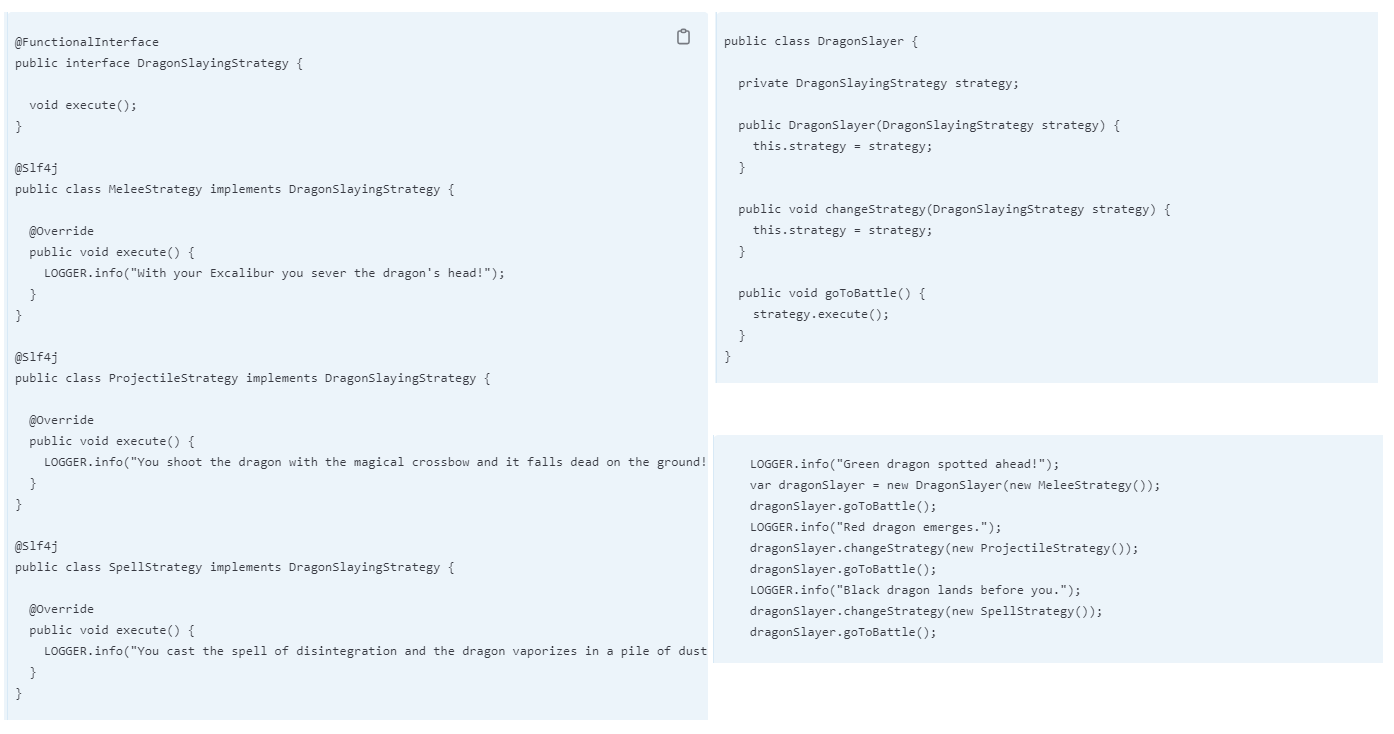


Figure 6. The sample code of the strategy pattern

**Conclusion**

This report has examined two of the fundamental challenges identified by Brooks in *No Silver Bullet*, namely complexity and changeability, and explored how mature design patterns can be applied to address these issues in practical software development. The Builder Pattern has been shown to reduce construction-related complexity by promoting modular design, improving clarity, and enhancing maintainability. On the other hand, the Strategy Pattern supports better management of behavioral variability by encapsulating algorithms as independent components and aligning with the Open/Closed Principle.

The RPG game case study demonstrated that the considered use of design patterns can strengthen system architecture while also improving collaboration among developers and responsiveness to evolving requirements. One important insight gained from this study is that design patterns are not merely theoretical concepts. Instead, they represent proven, reusable knowledge that can inform and support real-world development decisions.

Although no single solution can fully overcome the inherent challenges of software engineering, design patterns can be regarded as a toolbox of expert knowledge that addresses these essential difficulties in a practical and reusable way. Furthermore, their application encourages developers to approach problems with greater abstraction and design discipline. In this sense, design patterns represent one of the incremental yet valuable steps toward more effective software engineering practice.

**Task allocation**

**Reference**

[1] F. P. Brooks, “No Silver Bullet—Essence and Accident in Software Engineering,” in *Proc. IFIP 10th World Comput. Conf.*, 1986, pp. 1069–1076.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, 1995.

[3] C. Chung, “Builder,” in *Pro Objective-C Design Patterns for iOS*, Berkeley, CA: Apress, 2011, pp. 85–96.

[4] Apache Software Foundation, "Chapter 5. Fluent API," Apache HttpComponents, Apr. 2025. [Online]. Available: <https://hc.apache.org/httpcomponents-client-4.5.x/current/tutorial/html/fluent.html>.

[5] H. Wang, J. Zhang, and J. Guo, “Constructing data warehouses based on operational metadata-driven builder pattern,” in *Proc. 2015 Int. Conf. Logistics, Informatics and Service Sciences (LISS)*, Barcelona, Spain, 2015, pp. 1–4, doi: 10.1109/LISS.2015.7369630.

[6] Refactoring.Guru, "Strategy in Java," [Online]. Available: <https://refactoring.guru/design-patterns/strategy/java/example>.

[7] H. Patel, “A research paper on software design patterns,” *World J. Adv. Eng. Technol. Sci.*, vol. 13, no. 1, pp. 803–813, 2024.

[8] M.-E. Paschali, C. Volioti, A. Ampatzoglou, A. Gkagkas, I. Stamelos, and A. Chatzigeorgiou, “Implementing game requirements using design patterns,” *J. Softw. Evol. Process*, vol. 33, no. 12, p. e2399, 2021, doi: 10.1002/smr.2399.

[9] Java Design Patterns, “Builder,” [Online]. Available: <https://java-design-patterns.com/zh/patterns/builder>. [Accessed: Apr. 10, 2025].

[10] Java Design Patterns, “Strategy,” [Online]. Available: <https://java-design-patterns.com/zh/patterns/strategy/>. [Accessed: Apr. 10, 2025].