Article

Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in Object-Oriented Programming

Francesca Fallucchi [1,2,*,†] and Manuel Gozzi [1,†]

[1] Department of Engineering Sciences, Guglielmo Marconi University, 00193 Roma, Italy; m.gozzi@studenti.unimarconi.it
[2] Leibniz Institute for Educational Media, Georg Eckert Institute, Freisestraße 1, 38118 Braunschweig, Germany
* Correspondence: f.fallucchi@unimarconi.it
† These authors contributed equally to this work.

Featured Application: This software design pattern can be used in OOP programming in order to promote conceptual clarity, reduce coupling, and facilitate system scalability.

Abstract: Object-oriented programming (OOP) has long been a dominant paradigm in software development, but it is not without its challenges. One major issue is the problem of tight coupling between objects, which can hinder flexibility and make it difficult to modify or extend code. Additionally, the complexity of managing inheritance hierarchies can lead to rigid and fragile designs, making it hard to maintain and evolve the software over time. This paper introduces a software development pattern that seeks to offer a renewed approach to writing code in object-oriented (OO) environments. Addressing some of the limitations of the traditional approach, the Puzzle Pattern focuses on extreme modularity, favoring writing code exclusively in building blocks that do not possess a state (e.g., Java interfaces that support concrete methods definitions in interfaces starting from version 8). Concrete classes are subsequently assembled through the implementation of those interfaces, reducing coupling and introducing a new level of flexibility and adaptability in software construction. The highlighted pattern offers significant benefits in software development, promoting extreme modularity through interface-based coding, enhancing adaptability via multiple inheritance, and upholding the SOLID principles, though it may pose challenges such as complexity and a learning curve for teams.

Keywords: software engineering; software pattern; object oriented programming

Citation: Fallucchi, F.; Gozzi, M. Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in Object-Oriented Programming. *Appl. Sci.* **2024**, *14*, 5083. https://doi.org/10.3390/app14125083

Academic Editor: Andrea Prati

Received: 30 April 2024
Revised: 29 May 2024
Accepted: 10 June 2024
Published: 11 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Object-oriented programming (OOP) is a programming paradigm centered around the concept of "objects", which encapsulate both state and behaviors. This paradigm enables the modeling of complex systems in a manner that aligns closely with real-world scenarios. However, the perception of this modeling approach can vary among developers. In OOP, objects are instances of classes, which can inherit behaviors and states from one another, allowing for hierarchical structures and code reuse. This approach is widely appreciated for its ability to organize large codebases and enhance maintainability through key principles such as encapsulation, inheritance, and polymorphism. Encapsulation ensures that objects maintain their internal states privately, exposing only the necessary interfaces to interact with other parts of the system. Inheritance allows classes to derive from others, gaining their attributes and behaviors and promoting code reuse while reducing redundancy. Polymorphism enables objects of different classes to be treated uniformly through a common interface, which supports more flexible and dynamic coding. Despite these advantages, OOP faces several challenges. Objects in OOP can become so interdependent that changes to one may require changes to many, hindering modularization and increasing the risk of

bugs when modifying the system, leading to tight coupling. Complex inheritance chains can make the system difficult to navigate and extend due to the intertwined dependencies and behaviors. As systems expand, the tightly coupled nature of OOP makes it challenging to introduce changes without affecting the entire system architecture. From a technical standpoint, tight coupling reduces modularization within the codebase, making it hard to isolate and modify components without affecting others. This complexity not only makes the development process more error-prone and resource-intensive but also inhibits code reusability and extensibility. Components that are tightly intertwined cannot be easily reused or adapted to new requirements without substantial changes. On the non-technical side, tight coupling can affect organizational dynamics and collaboration. Teams working on tightly coupled codebases may struggle with coordination and effective collaboration. The complexity introduced by tight coupling can obscure the understanding and reasoning about the code, leading to communication barriers and reduced productivity. Furthermore, as the codebase grows and becomes more convoluted, onboarding new team members becomes increasingly challenging due to the complex relationships between various components. Addressing these challenges typically involves implementing design patterns and best practices that promote modularity, reduce coupling, and simplify maintenance. The widespread adoption of OOP indicates its fundamental role in software development, yet recognizing and mitigating its limitations remains crucial for developing robust, maintainable software systems.

In this paper, we introduce the Puzzle Pattern, an innovative approach designed to complement existing software design patterns in addressing challenges in object-oriented programming (OOP). Traditional OOP often relies on concrete classes to encapsulate both state and behavior. However, the Puzzle Pattern advocates for a unique use of interfaces, enhancing extreme modularity and code organization. This pattern encourages developers to think about system design in terms of potential interactions and dependencies delineated by these interfaces, thus guiding the structural design of software systems toward a higher level of modularity. The Puzzle Pattern leverages interfaces not just to define a contract for functionalities via method signatures but also to promote an unusual practical use of interfaces. While traditionally, interfaces in OOP are used to decouple the architecture of software systems and specify operations without implementations, the Puzzle Pattern uses them to enhance modularity and abstraction. However, the level of abstraction of the Puzzle Pattern remains congruent with that of standard patterns, which typically reside at the model level (M1) in the Meta Object Facility (MOF) framework [1]. By advocating for the design of behaviors to be primarily encapsulated in interfaces, the Puzzle Pattern enhances the decoupling of software components. It aligns well with core OOP principles that interfaces should specify operations without providing implementations. Rather than embedding functionality directly within classes, developers are encouraged to define interfaces representing individual “puzzle pieces” of functionality. These interfaces then serve as blueprints for concrete classes, which are assembled through mechanisms such as composition or dependency injection. This modular approach facilitates significant reductions in coupling by separating the implementation details from the interface definitions. Each puzzle piece can be developed, tested, and maintained independently, fostering a modular design that enhances conceptual clarity and communication among team members. The Puzzle Pattern thus introduces a fresh perspective on software construction, aligning seamlessly with principles like Dependency Injection and Multiple Behavioral Inheritance to promote loose coupling and flexibility. The focus on extreme modularity, exclusively composing code in interfaces, offers numerous benefits within software development, including a clearer division of responsibilities and simplified maintenance. However, the Puzzle Pattern also introduces its own set of challenges. Relying on multiple inheritances might introduce complexity, necessitating vigilant management of shared states among interfaces. Furthermore, while aiming to remain compatible with traditional object-oriented programming methods, teams accustomed to more conventional practices may experience a learning curve. Conventional practices in OOP typically include the

use of concrete classes to encapsulate both state and behavior, reliance on inheritance to reuse code, and the implementation of polymorphism through class hierarchies. For example, developers often create base classes with common functionality that other classes inherit from, or they use design patterns like the Singleton or Factory patterns [2]. These practices emphasize a straightforward, hierarchical approach to structuring code. In contrast, the Puzzle Pattern promotes extreme modularity by using interfaces as independent building blocks, which might require developers to shift their mindset from traditional inheritance-based design to composition and interface-driven design. This shift can present a learning curve for those more familiar with orthodox OOP methods.

The best use case for this pattern is incremental refactoring. This approach allows teams to gradually introduce changes by refactoring specific components of a larger monolithic system into loosely coupled, independently manageable modules. Incremental refactoring reduces risk, facilitates easier adoption, and minimizes disruption to existing systems, making the Puzzle Pattern an ideal strategy for modernizing legacy applications in a controlled and effective manner.

The rest of this paper is structured as follows. After the introduction, in Section 2, we want to provide a high-level overview of the current state of the art of software design patterns, giving an overview of the main OOP problems that these methodologies solve. In Section 3, subsequently, we conceptualize the Puzzle Pattern by defining the concepts on which it is based and analyzing its conformity to the software development principles considered to be the most common best practices. Moreover, in Section 4, a use case of application of the Puzzle Pattern is described, analyzing the design and structuring aspect of the source code, discussing the technical–functional aspects in detail. Finally, Section 5 discusses the results of the case study, highlighting the advantages and disadvantages of the Puzzle Pattern compared to other design approaches.

2. Related Work

Software design patterns are indispensable tools in object-oriented programming (OOP), offering structured solutions to mitigate common issues such as tight coupling, which can significantly hinder system flexibility and maintainability. This section delves into a diverse array of sources that discuss various design patterns and evaluates their impact on software architecture, highlighting how these patterns facilitate complex design problem-solving and enhance software reuse and maintenance.

The foundational work by Gamma et al. [2] introduced a classification of 23 design patterns into creational, structural, and behavioral categories. These patterns have had a profound impact on modern software engineering practices by providing a framework that encourages software reuse and systematic problem-solving. For instance, the Dependency Injection pattern [3] effectively decouples classes by externalizing their dependencies, thereby enhancing both modularity and testability. However, Kerievsky [4] raises concerns that without meticulous management, such patterns can introduce a surplus of boilerplate code, complicating the codebase more than simplifying it. Similarly, while the Observer pattern fosters loose coupling between subjects and observers, Hevery [5] points out that it can lead to unpredictable side effects if event management is not handled with care. A notable trend in OOP is the shift from inheritance-based to composition-oriented designs. The principle of "Composition over Inheritance", advocated by experts like Martin [6], posits that using object composition offers more flexibility in system design compared to traditional inheritance. Although inheritance facilitates code reuse, it can also lead to fragile dependencies. Suryanarayana et al. [7] discuss how inheritance hierarchies can become cumbersome and prone to errors over time. Another core principle in OOP, encapsulation, restricts access to an object's internal state, which helps maintain control over how data are accessed and modified. The Principle of Least Privilege, as discussed by Saltzer and Schroeder [8], advocates for minimizing access rights for modules to only what is necessary for their operations, thereby mitigating the risk of unintended interactions and enhancing system security.

The concepts of mixins and traits have been extensively explored to address issues related to code reuse and the limitations of single inheritance in object-oriented programming. Mixins, initially conceptualized in languages like Flavors and CLOS, have evolved to offer a flexible mechanism for composing behaviors in class-based languages without the complexities of multiple inheritances. Researchers such as Bracha and Cook have detailed the benefits and semantic models of mixins, emphasizing their role in providing a structured yet flexible approach to object composition [9]. Similarly, traits, introduced by Schärli et al., provide a fine-grained mechanism to compose classes from reusable components. Traits differ from mixins in that they allow for an explicit composition model that can resolve method conflicts and provide better modularity [10]. Both concepts have been integrated into modern programming languages, such as Scala, which uses traits to allow for multiple inheritances and dynamic behavior composition. The academic discourse surrounding these features continues to explore their implications for software design, particularly how they can be leveraged to create more maintainable and adaptable software architectures. Studies like Ducasse et al. have further investigated how traits can be used in large-scale systems to improve the robustness and clarity of code, suggesting a significant shift towards adopting these modular design paradigms in industry practices [11].

The landscape of software design patterns is rich and varied, reflecting the diverse challenges faced in software development across different domains and technologies. The seminal classification by Gamma et al. [2] laid the groundwork for this, but the evolution of software engineering has seen the introduction of new patterns and the adaptation of existing ones to meet modern needs. For instance, the introduction of cloud computing and microservice architectures has led to the development of patterns like the Circuit Breaker or the API Gateway, which address issues specific to distributed systems [12]. Additionally, the rise of reactive programming has popularized patterns such as Event Sourcing and CQRS (Command Query Responsibility Segregation) that deal with real-time data processing and state management in systems designed to be highly responsive and scalable [13]. The academic community continues to explore these patterns not only to tackle technical challenges but also to address issues of system maintainability and adaptability in the face of rapid technological changes [14]. Notable contributions like those by Newman [14] and Vernon [13] discuss these modern patterns in the context of their practical application, showcasing how they cater to the specific needs of contemporary software architectures and thus broadening the spectrum of design patterns beyond the traditional ones identified in earlier studies. This diversity highlights the dynamic nature of software engineering and underscores the importance of continuous learning and adaptation within the field.

Our work introduces the Puzzle Pattern, a new software pattern that excels particularly in the context of refactoring, where it fits seamlessly into the process of transforming legacy systems or reorganizing existing codebases. The Puzzle Pattern builds on the strengths of established patterns and promotes a highly modular and dynamically adaptable architecture, making it especially beneficial in the refactoring process. By enabling the incremental modularization of systems, the Puzzle Pattern allows for the step-by-step restructuring of a software system. This methodological approach not only facilitates easier maintenance and better adaptability but also minimizes disruption during the refactoring process. In refactoring scenarios, the Puzzle Pattern enables developers to isolate and redefine functionalities into distinct, interface-defined modules ("puzzle pieces"). These modules can then be developed, tested, and maintained independently of one another, which is crucial for managing large and complex software systems. This pattern's emphasis on interface-based modularity aligns well with the principles of loose coupling and high cohesion, further fostering the creation of flexible and maintainable software architectures.

3. Puzzle Pattern Conceptualization

The Puzzle Pattern in software development draws a compelling analogy from a physical puzzle, which consists of two main elements: a board and a set of pieces. In a puzzle, the pieces are assembled on the board and together, they form a complex design.

Each piece, while enclosing only a small part of the overall picture, is crucial to completing the puzzle. Similarly, in the Puzzle Pattern, we can associate the puzzle pieces with interfaces and the puzzle board with a concrete class. A concrete class, much like a puzzle board, gains a definite meaning only when all the pieces are correctly positioned. However, each piece retains a degree of independence from the board itself. This relationship mirrors the dynamic between interfaces and concrete classes in software design. Each interface, representing a puzzle piece, is designed to implement a specific behavior and to declare all the requirements that need to be met to realize this behavior. These requirements are typically represented by abstract methods within the interface.

The separation between contract (interfaces) and implementation (concrete classes) is a fundamental tenet of object-oriented programming (OOP), crucial for the functionality of many frameworks such as Java Enterprise and Spring Boot. This principle ensures flexibility, maintainability, and scalability by decoupling the specification of behavior from its implementation. In our discussion of the Puzzle Pattern, we emphasize this separation by advocating for the use of interfaces to define behaviors and concrete classes to provide the actual implementations that consists of state. The Puzzle Pattern does not contradict this tenet but rather builds upon it by encouraging the creation of highly modular and interchangeable components. Each "puzzle piece" (interface) represents a distinct contract, while the concrete classes that implement these interfaces provide the state needed. This clear delineation enhances the flexibility and maintainability of the system, similar to how frameworks like Spring Boot leverage interfaces and dependency injection to manage complexity and promote clean architectures. The Puzzle Pattern treats interfaces as independent, self-contained units of functionality. Each interface defines a specific and atomic behavior enhancing its reusability and facilitating testing. By focusing on small, interchangeable components, the Puzzle Pattern enables the dynamic assembly of applications. Developers can mix and match different "puzzle pieces" to tailor the functionality of an application to specific needs without altering the underlying architecture. The clear separation of concerns ensures that changes in one part of the system have minimal impact on others. This approach not only aligns with fundamental OOP principles but also enhances the flexibility and maintainability of software applications.

Behavioral code placed within interfaces offers a way to provide shared functionality without forcing all implementing classes to duplicate code, but their use should be carefully considered. The Puzzle Pattern leverages default methods to promote code reuse and reduce boilerplate, but this is done with the awareness of the potential risks. This approach should be limited only to small, atomic, and idempotent functionalities. Also, the cardinality of "puzzle pieces" assembled into a concrete class should be chosen with parsimony. Exceeding the cardinality of pieces could negatively impact the code maintainability. The concrete class that assembles puzzle pieces must be congruent with the pieces themselves. This means that a complex functionality should be broken in puzzle pieces when there is the need to reuse some of the pieces. Furthermore, a single piece should never rely on the concrete class's behavioral code, except for the state dependencies' provisioning.

3.1. Formalizing Puzzle Pattern

To provide a clear and standardized description of the Puzzle Pattern, we adopt the Gang of Four pattern specification template [15]. This ensures consistency and facilitates a better understanding of the pattern's applicability and implementation.

- Intent: the Puzzle Pattern aims to modularize software components by integrating functionalities implemented in dispersed interfaces (puzzle pieces) into a single cohesive unit (integration board).
- Applicability: This pattern is applicable in scenarios where modularity, flexibility, and maintainability are critical. It is particularly useful in systems undergoing refactoring or in environments with diverse architectural styles.

- Structure: the Puzzle Pattern consists of puzzle pieces (independent modules or interfaces that encapsulate specific functionalities) and a board (a composite unit that integrates and coordinates the puzzle pieces).
- Participants: a puzzle piece defines a modular functionality, while the puzzle board integrates multiple puzzle pieces into a cohesive unit.
- Collaborations: puzzle pieces interact through well-defined interfaces, and the integration board coordinates their interactions.
- Consequences: The Puzzle Pattern enhances modularity, supports incremental development, and facilitates integration with external systems. However, it requires careful design to ensure compatibility and interoperability.
- Implementation: the pattern can be implemented using object-oriented principles, with puzzle pieces represented as interfaces or classes and the integration board as a composite class.
- Known uses: the Puzzle Pattern has been successfully implemented in various domains, including software refactoring projects, microservice architectures, and modular application development.
- Verification: the pattern has been verified across three unrelated real-world domains, demonstrating its versatility and applicability.

3.2. Implementing Puzzle Pattern

The primary idea behind the Puzzle Pattern is to develop code within these interfaces, the puzzle pieces. Concrete classes—or the puzzle boards—are then assembled by implementing these interfaces. This method of organizing software components enhances modularity and encourages the development of maintainable and scalable software systems. Each interface mandates a behavior through its methods, using abstract methods to outline the necessary services or state required from the concrete class. In other words, each abstract method in an interface simply returns the instance needed to provide the functionality required in the concrete methods of that interface. This state or functionality is provided by the concrete class that implements the interface, exploiting the state implemented by itself. This approach not only facilitates a clear division of responsibilities but also enhances the system's adaptability by allowing individual components to be developed, tested, and maintained independently. By fostering a modular design, the Puzzle Pattern promotes a structure where the integration of new functionality or the modification of existing features becomes more manageable, mirroring the way puzzle pieces can be individually added or replaced without disrupting the entire system.

Let us consider a simplistic example in which the aim is to implement a Java service that is responsible for handling digitalized documents. The OOM that represents documents is already defined (the class Document is defined by a content, a timestamp, and a type), as well as the DatabaseService that is responsible for database interactions. Two functionalities need to be implemented: the document deletion and the document acquisition. The approach to follow is the following one:

1. Define one puzzle piece for each functionality that needs to be implemented.
2. Implement the behavior of each puzzle piece. If the single puzzle piece needs to rely on further functionalities, in this scenario, the DatabaseService, then declare abstract "getters" methods that are responsible for those third-party services provisioning.
3. Assemble the "board" by defining a concrete class that implements all the puzzle pieces needed.

The implementation of the Document Management System using the Puzzle Pattern approach, as depicted in the provided UML diagram (Figure 1) and subsequent Java code 1 in Appendix A, offers a clear demonstration of how to architect software to promote modularity and loose coupling. By segregating different functionalities such as the acquisition and deletion of documents into separate interfaces, the system achieves a high level of modularity. This separation allows for distinct behaviors to be developed, tested, and maintained independently of one another, enhancing the system's manageability and scalability.

Implementing functionalities as interchangeable puzzle pieces (interfaces) enables easier adjustments and additions down the line. For instance, introducing a new operation, such as document modification, can be done by adding another interface and implementing it in the DocumentService class or a new class altogether without disrupting existing code. The design promotes reusability through interfaces. Different parts of the application can reuse the same interface for similar operations, or new projects can adopt the established interfaces, reducing the need to rewrite common functionality. The interfaces ensure that concrete implementations are loosely coupled, minimizing the impact of modifications on other parts of the system and easing maintenance.

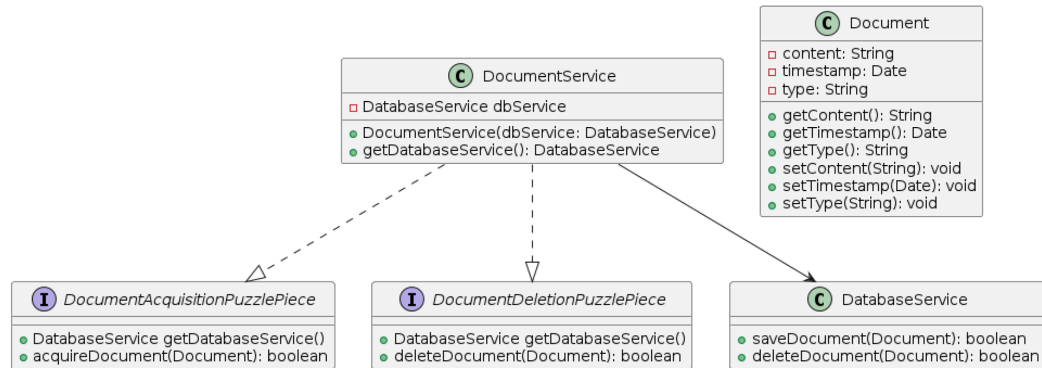

Figure 1. UML class diagram that formalizes the example described above.

While the system promotes modularity and flexibility, it also introduces additional complexity. Developers must understand how interfaces interact and ensure that concrete classes correctly implement all required interfaces. This learning curve might slow down initial development, especially for teams unfamiliar with interface-driven designs. The abstraction and separation of concerns require more classes and interfaces, potentially leading to increased system overhead. This can complicate the architecture unnecessarily if the application scope does not justify such a granular level of separation. Managing dependencies can become more challenging as the number of interfaces and their implementations grows. Developers must carefully manage these dependencies to ensure that the system remains stable and that service implementations are correctly provided where needed. While modular design simplifies some aspects of testing by allowing individual components to be tested in isolation, it also requires comprehensive integration testing to ensure that all pieces work together as expected. This can increase the complexity and scope of test planning and execution.

3.3. SOLID Principle Compliance

The SOLID principles —Single Responsibility [16], Open–Closed [16], Liskov Substitution [17], Interface Segregation [2], and Dependency Inversion [6] —serve as a benchmark for assessing the robustness and adaptability of software architectures. The objective here is to critically examine how well the presented pattern aligns with each SOLID principle, elucidating both areas of strong adherence and potential challenges. By evaluating the pattern through the lens of these principles, we aim to provide a comprehensive understanding of its compatibility with established best practices in object-oriented design.

The Single Responsibility principle is a cornerstone in object-oriented design, emphasizing the need for a class or interface to have only one reason to change. In Puzzle Pattern, this principle is diligently satisfied as each interface takes on a singular responsibility, encapsulating it comprehensively within its scope. This adherence ensures that each interface remains focused on a specific aspect of functionality, promoting a modular and easily maintainable structure. By adhering to the Single Responsibility Principle, the pattern not only aligns with established design principles but also contributes to a codebase that is more resilient to changes and easier to comprehend. This adherence to singular responsibility at

the interface level lays a solid foundation for achieving clarity and maintainability, crucial aspects in the development of robust software systems.

The adherence to the Open–Closed principle is a pivotal strength of the proposed Puzzle Pattern. It establishes a development environment where the introduction of new interfaces is a seamless process, devoid of the need to rectify or modify existing code within the class responsible for their implementation. This design philosophy imbues the class with an "open to extensions, closed to modifications" characteristic, a key tenet of the Open–Closed principle. This means that the system can readily accommodate new functionality through the addition of interfaces without necessitating changes to the existing codebase. By strictly adhering to the Open–Closed principle, the Puzzle Pattern promotes an extensible and adaptable architecture, enabling developers to enhance the system's capabilities without compromising the stability and integrity of the core code. This principled approach not only aligns with best practices but also contributes to a software design that is resilient to evolving requirements and scalable in the face of future enhancements.

The pattern's alignment with the Liskov Substitution Principle (LSP) is inherent in the assembly operation it employs. The class's behavior is intricately tied to the implementation of interfaces, providing a solid foundation for LSP compliance. This design approach guarantees that the behavior of the class is exclusively defined through the interfaces it implements. Consequently, the addition or removal of an implementation has no cascading impact on the pre-existing behavior. The LSP, which asserts that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program, is elegantly upheld. The pattern's commitment to maintaining the expected behavior even in the face of changes in implementation ensures a robust and dependable software system, consistent with the principles of Liskov Substitution.

The Interface Segregation principle is meticulously fulfilled through the deliberate creation of atomic interfaces within the proposed pattern. Each interface is crafted to be independent of methods it does not utilize, aligning with the essence of Interface Segregation. This intentional segregation ensures that each interface is inherently cohesive, containing only the functionality that is strictly necessary for its specific responsibility. By adhering to the Interface Segregation principle, the pattern fosters a modular and granular design, where each interface encapsulates a specific set of related functionalities. This segregation not only enhances the clarity and readability of the codebase but also contributes to a system that is flexible and adaptable, allowing for the construction of tailored solutions by selectively combining these atomic interfaces. The conscientious application of Interface Segregation in Puzzle Pattern underscores its commitment to best practices in interface design and encapsulation.

The Dependency Inversion principle is rigorously adhered to in the proposed pattern, as concrete classes exhibit a dependency on abstractions provided by interfaces rather than relying on concrete implementations. This strategic inversion of dependencies introduces a level of flexibility that is instrumental in managing the intricacies of the overall system. By establishing a framework where high-level modules are not dependent on low-level modules but both rather both depending on abstractions, the pattern facilitates a decoupled and modular architecture. This inversion of dependencies not only aligns with the Dependency Inversion principle but also contributes to a system that is more adaptable to change. The abstraction of dependencies through interfaces enables a seamless interchangeability of components, allowing for easier maintenance, extensibility, and scalability. In essence, the Dependency Inversion principle is a cornerstone of the Puzzle Pattern, fostering a design paradigm that promotes flexibility and robust dependency management.

3.4. Differences between Puzzle Pattern, Mixins, and Traits

The Puzzle Pattern, traits [10], and mixins [9] are all techniques used in software development to enhance modularity, reusability, and manage complexity in object-oriented and functional programming environments. Understanding the nuances between these

concepts can help in choosing the appropriate approach for specific software design requirements. In this section, we provide a detailed comparison of these three similar but distinct concepts. The Puzzle Pattern is primarily a software design approach that advocates for the use of interfaces to define behavior, with concrete classes assembled by implementing these interfaces to provide complex functionalities. The pattern focuses on extreme modularity, encouraging the design of small, interchangeable interface-based components, akin to puzzle pieces, that can be assembled in various ways to form a complete application. This pattern is particularly useful in scenarios requiring high flexibility and modularity, allowing for the dynamic assembly of applications from a set of well-defined and loosely coupled components.

Traits are a language-level concept used primarily in Scala and other programming languages like PHP and Rust. They are conceptually similar to interfaces with partial implementation, used to define object types by specifying the signature of the supported methods and can contain method implementations. Traits can be composed to extend the functionality of classes in a flexible and reusable manner. They are ideal for sharing methods across multiple classes in languages that do not support multiple inheritances, providing a means to avoid code duplication and promote clean inheritance hierarchies. Traits allow the implementation of methods that can be shared across different classes, providing a controlled mechanism to combine behaviors in classes without forcing a class hierarchy. Mixins are classes that contain methods for use by other classes without needing to be the parent class of those other classes. They provide a mechanism to add additional functionality to a class and are typically used in a context where they inject additional behavior without affecting inheritance. Mixins are often used in languages like Python, Ruby, and JavaScript and are useful when you need to augment the behavior of a class in environments where multiple inheritance might be necessary but is undesirable or impossible due to language constraints. They enable the sharing of functionalities across unrelated classes, allowing behaviors to be added to classes without modifying the class hierarchy. Mixins provide significant flexibility in extending the functionality of classes without the need for inheritance, helping to avoid the diamond problem of multiple inheritances by enabling selective behavior sharing. The Puzzle Pattern shares some conceptual similarities with mixins and traits, as all three aim to enhance code reusability and modularity. Mixins, particularly in Java, leverage multiple inheritance through interfaces to inject additional behavior into classes. This approach, however, often results in complex inheritance hierarchies and tightly coupled systems. Traits, found in languages like Scala, provide a more controlled form of code reuse by allowing the composition of behaviors without the pitfalls of multiple inheritance. While mixins and traits primarily focus on adding functionalities to classes, the Puzzle Pattern emphasizes extreme modularity and decoupling. The pattern treats interfaces as independent building blocks, each encapsulating distinct behaviors that can be combined to form a cohesive system. This approach reduces dependencies and enhances system scalability, promoting a clear separation of concerns. Unlike mixins, which can lead to tangled inheritance structures, the Puzzle Pattern maintains a simpler and more maintainable codebase by strictly defining interactions through interfaces.

Table 1 provides a detailed comparison of platform-dependent solutions (such as traits and mixins) with the proposed platform-independent Puzzle Pattern. The comparison highlights several unique benefits of the Puzzle Pattern:

- Modularity: the Puzzle Pattern enhances modularity by allowing functionalities to be encapsulated within discrete "puzzle pieces" that can be easily integrated into different systems.
- Interoperability: unlike traits and mixins, which are often language-specific, the Puzzle Pattern is designed to be platform-independent, facilitating easier integration across various programming environments.
- Maintainability: by adhering to common interfaces, the Puzzle Pattern simplifies the maintenance of complex systems, enabling easier updates and modifications.

This table and accompanying discussion provide a clear rationale for the preference of the Puzzle Pattern over traditional multiple inheritance implementations, demonstrating its practical advantages in real-world scenarios.

Table 1. Comparison of Puzzle Pattern, traits, and mixins.

Puzzle Pattern	Traits	Mixins
Focuses on extreme modularity using interfaces for defining behaviors and concrete classes for implementations.	Offer method implementations in interfaces for sharing functionalities across classes.	Allow behavior injection into classes without extending them, avoiding direct inheritance.
Ideal for applications requiring high modularity and flexibility, particularly in dynamic systems.	Best suited for environments where code reusability across classes is needed without multiple inheritance.	Useful for adding functionalities to classes in languages that restrict or do not support multiple inheritances.
Enhances modularity and decouples components, facilitating independent development and maintenance.	Enable sharing of functionality and composition over inheritance, reducing redundancy.	Provide flexibility in extending class functionalities and help avoid the complexities of multiple inheritance.
Uses interfaces to define required operations and concrete classes to perform them.	Traits are partially implemented interfaces that can be composed into classes.	Mixins are classes that provide methods to other classes without requiring inheritance.

3.5. Differences between Puzzle Pattern and Command Pattern

The Puzzle Pattern and the Command Pattern are both design patterns that enhance modularity and maintainability, but they serve different purposes. The Puzzle Pattern focuses on integrating dispersed functionalities into cohesive units by assembling modular components, or “puzzle pieces”, into an “integration board”. This approach promotes enhanced modularity and interoperability across different platforms, making it particularly suitable for complex systems that require a high degree of modularity and flexibility. On the other hand, the Command Pattern encapsulates a request as an object, thereby allowing for the parameterization of clients with queues, requests, and operations. It decouples the sender and receiver of a request, enabling features like undoable operations and deferred execution. While the Command Pattern is primarily concerned with executing commands and maintaining action history, the Puzzle Pattern is designed to modularize system components and facilitate their integration. The Command Pattern deals with behavioral aspects of software design, whereas the Puzzle Pattern addresses structural concerns, focusing on the organization and integration of system components. Despite their differences, both patterns aim to improve system flexibility and maintainability, each addressing unique aspects of software design. In this subsection, a comparative analysis Table 2 is provided.

In summary, while the Puzzle Pattern and Command Pattern serve different roles within software design, they both contribute significantly to enhancing system flexibility and maintainability. The Puzzle Pattern is ideal for the structural organization and integration of modular components, making it suitable for complex, cross-platform systems. Conversely, the Command Pattern excels in scenarios requiring a decoupling of request handling and managing action histories, providing powerful behavioral control mechanisms.

Table 2. Comparison of Puzzle Pattern and Command Pattern.

Aspect	Puzzle Pattern	Command Pattern
Purpose	Integrates dispersed functionalities into cohesive units through modular components.	Encapsulates a request as an object, allowing for parameterization of clients with queues, requests, and operations.
Focus	Structural design, enhancing modularity and interoperability.	Behavioral design, managing actions, and maintaining history.
Key components	Puzzle pieces (modular components) and integration board (assembly unit).	Command (encapsulated request), Receiver (executes the command), Invoker (calls the command), and Client (creates the command).
Primary Use	Modularizing and integrating system components across different platforms.	Decoupling sender and receiver of a request, enabling features like undoable operations and deferred execution.
Advantages	Enhanced modularity, platform independence, and improved system maintainability.	Decouples command execution, allows for dynamic command management, and supports undo/redo functionality.
Example use case	Complex systems requiring high modularity and flexibility, such as enterprise applications.	Applications needing flexible command execution and action history management, such as text editors or transaction systems.

3.6. Unit Testing Considerations

Testing software components developed with the Puzzle Pattern offers significant benefits, particularly in enhancing the clarity and efficiency of the testing process. This pattern's structure supports a systematic approach that effectively isolates and verifies the behavior of individual components. The modular nature of the Puzzle Pattern allows each component (or "puzzle piece") to be tested in isolation. This separation simplifies test creation because developers can focus on one small, manageable piece at a time without interference from other parts of the system. Each interface defines clear boundaries and responsibilities, which means that test cases can be very specific and targeted. This not only makes tests easier to write but also enhances their effectiveness in catching issues. Testing code structured according to the Puzzle Pattern involves several key steps that focus on interface behavior and dependency management:

1. Stub implementation: Begin by defining a stub class that implements the interface you wish to test. This stub class serves as a controlled test subject that directly embodies the interface's contract, allowing you to focus on testing behavior without the complexities of the full application context.
2. Mocking state dependencies: The next crucial step is to mock the state dependencies within the stub class. This involves creating mock objects for any services or components that the interface's methods depend on to function. By mocking these dependencies, you can simulate a wide range of conditions and edge cases, ensuring that the interface can handle various scenarios robustly. This step is vital as it allows the testing of interface methods under controlled conditions that mimic expected operational scenarios.
3. Testing behavioral methods: With the stub class set up and dependencies mocked, the focus shifts to testing the behavioral methods defined at the interface level. This testing ensures that all behaviors prescribed by the interface are correctly implemented in the stub, adhering to the specified behaviors and interactions. Any class that later

plugs into this "puzzle piece" of the architecture will automatically adhere to the tested behaviors, thus ensuring consistent functionality across different implementations.

While testing with a stub provides a robust framework for ensuring that interface behaviors are correct and stable, it does not eliminate the need to test the real concrete classes. The primary aim of testing with a stub is to validate the behavior as specified by the interface. However, the concrete classes, which implement these interfaces, also require testing, specifically focusing on the "state getters" methods. These methods, often crucial for the correct operation of the class within the broader application, need to be verified to ensure they correctly manage and provide access to the class's state. This dual approach to testing—focusing on both the abstract behaviors via stubs and the concrete implementations' state management—ensures comprehensive coverage. It validates not only the theoretical design as specified by the interfaces but also the practical implementation in the concrete classes. Due to the standardized approach to implementing and interfacing in the Puzzle Pattern, tests written for one module can often be adapted or reused for another module with similar functionalities. This reuse can significantly reduce the effort required to write new test cases, as common testing logic for interface behavior does not need to be rewritten from scratch. Instead, it can be slightly modified to fit the nuances of different implementations, which accelerates the testing process and ensures consistency across tests. The clear compartmentalization of functionalities within interfaces facilitates more straightforward and efficient regression testing. When changes are made to a particular module, only the tests associated with that module need to be rerun to verify that no existing functionalities have been broken. This focused approach reduces the scope and complexity of regression tests, making them quicker to execute and easier to manage.

Finally, testing within the framework of the Puzzle Pattern encourages adherence to best practices such as the Single Responsibility Principle and the Open/Closed Principle. Each component being responsible for a single functionality simplifies the creation of specific, focused tests. Additionally, since components are closed for modification but open for extension, tests can remain valid and require fewer updates as new functionalities are added via new modules rather than changes to existing code. This methodical testing strategy underpins the reliability and maintainability of software developed with the Puzzle Pattern, offering clear pathways to diagnose issues and validate functional integrity before deployment.

4. Use Case: The Puzzle Pattern Applied to Incremental Refactoring

The evolving landscape of software development demands continual adaptation and improvement of existing systems. As businesses strive to modernize their legacy applications, the challenge lies in doing so without disrupting ongoing operations. This section explores how the Puzzle Pattern can be applied to object-oriented programming (OOP)-based systems to transform them into modular, flexible architectures. Such transformations are critical for organizations aiming to update and enhance their software applications, which are often hindered by tightly coupled components, sprawling codebases, and outdated technologies. The integration of the Puzzle Pattern into such environments offers a structured approach to reducing maintenance costs and enhancing system adaptability. This is particularly beneficial in scenarios where the legacy system's rigidity prevents integration with modern infrastructures like cloud services or microservices, thereby escalating maintenance costs. The intent here is to showcase practical applications of the Puzzle Pattern in creating more modular and flexible system architectures.

Integrating innovative design patterns into established systems poses significant challenges, particularly when dealing with legacy systems characterized by tight coupling and complex inheritance hierarchies. The Puzzle Pattern provides a modular approach that can be gradually introduced to enhance flexibility and maintainability. This approach is composed of three main phases:

1. Incremental refactoring: Begin by thoroughly analyzing the system to identify the core components that exhibit high complexity and tight coupling. These components usu-

ally benefit most from enhanced modularity and reduced interdependencies. Instead of opting for a complete overhaul, which can be disruptive and risky, incrementally refactor these components. Implement the Puzzle Pattern by breaking down the components into smaller, more manageable, and loosely coupled pieces. This step-by-step approach minimizes operational disruption and enables a continuous assessment of the pattern's impact on system performance and maintenance.

2. Interface abstraction: In systems that are predominantly based on concrete class implementations, it is beneficial to gradually introduce interfaces that define coherent sets of behaviors. This strategic insertion of interfaces helps in defining clear contracts for interactions within the system. Existing classes can be refactored to implement these new interfaces, effectively segregating responsibilities. This segregation facilitates easier modifications, enhances testability, and supports better adherence to the Open/-Closed principle, where classes are open for extension but closed for modification.
3. State management: Focus particularly on components that handle state management, as these components are critical and often benefit significantly from architectural enhancements like the Puzzle Pattern. Refactor these components by isolating state-related methods into specific interfaces. These interfaces can then be implemented by concrete classes as needed. This isolation helps in encapsulating the state logic, making the system more robust against changes and easier to debug. It also enhances scalability by allowing different parts of the system to manage their state independently or even adopt different state management strategies if required.

In example, consider a set of legacy classes: User 2 in Appendix A; UserDatabaseService 3 in Appendix A; UserManager 2 in Appendix A. User is the simplistic data model that represents a technical user, UserDatabaseService represents the DAO layer, and UserManager represents the controller that manipulates users working together with UserDatabaseService. In order to introduce Puzzle Pattern here, we can proceed as follows: First of all, we introduce a UserDataProcessor interface 5 in Appendix A that implements the same methods as UserManager. It is crucial here to keep the same signature as that of the original methods getUserById and updater. UserDataProcessor also defines a getUserDatabaseService method that provides the right UserDatabaseService instance to work on. Then, we can start working on UserManager implementing the interface UserDataProcessor. Now, it is time to implement the getUserDatabaseService method defined in UserDataProcessor. Once completed, we can finally remove the two public methods getUserById and updateUser from the UserManager class. Since the original signature has not changed, we have properly refactored the class. UserDataProcessor now holds all the behavior, while UserManager holds only the state in the form of a UserDatabaseService instance (the refactored version of UserManager 6 in Appendix A). The adoption of the Puzzle Pattern in this context underscores its utility in transforming legacy systems into more modular, maintainable architectures. Through careful abstraction and incremental refactoring, systems become better structured for future enhancements and easier debugging. However, this approach requires meticulous planning and consideration of existing system dependencies to avoid introducing new complexities. The modular design facilitated by the Puzzle Pattern can significantly improve the agility and responsiveness of software development teams, particularly in environments where requirements evolve rapidly.

Procedural programming emphasizes a sequence of computational steps, fundamentally differing from the modular, interface-based approach of the Puzzle Pattern. Adopting this pattern necessitates a significant shift in thinking and coding practices. Existing procedural codebases are typically monolithic and tightly coupled, making the identification and extraction of modular components (puzzle pieces) challenging. Substantial refactoring is required to introduce interfaces and separate concerns. Developers accustomed to procedural programming must learn and adopt object-oriented principles, including the use of interfaces, abstraction, and encapsulation. The integration process begins with identifying cohesive functional units in the procedural code that can be refactored into separate interfaces. These units should represent distinct pieces of functionality that can stand alone.

The next step involves creating interfaces for these functional units, specifying the required methods and abstracting the behavior of existing functions into well-defined contracts. The procedural code is then gradually refactored to implement these interfaces, which may involve breaking down large functions into smaller methods and encapsulating state where necessary. An incremental refactoring approach is advisable to minimize disruption, starting with a few key components and progressively applying the pattern to the rest of the system.

Achieve Interoperability Using Puzzle Pattern, Facade Pattern, and Adapter Pattern

In complex software systems, achieving seamless interoperability between different components and architectural styles is crucial. The integration of the Adapter and Facade patterns [2] with the Puzzle Pattern addresses this challenge by providing mechanisms to bridge gaps between modern, modular components and legacy systems. These design patterns play complementary roles in ensuring that disparate parts of the system can communicate and function together effectively.

Integrating the Adapter and Facade patterns with the Puzzle Pattern provides a robust solution for achieving interoperability between disparate system components. The Adapter Pattern plays a crucial role in translating the interfaces of the Puzzle Pattern components to match those expected by legacy systems. This translation ensures seamless interaction and data flow between modern, modularized components and older, monolithic ones. In the Java code 7 in Appendix A (refer to User.java 4 in Appendix A in order to get an overview of User class), the UserServiceAdapter class exemplifies this by adapting the UserService interface, which follows the Puzzle Pattern, to work with the legacy LegacyUserManager class. The adapter ensures that calls to getUserById and updateUser are correctly routed to the legacy methods, thus bridging the gap between the new and old parts of the system. The Facade Pattern, on the other hand, simplifies the interaction with the system by providing a unified interface that abstracts the complexities of multiple underlying components. In the example, the UserFacade class consolidates operations from both the legacy and the new Puzzle Pattern-based systems. This facade hides the intricacies of whether the underlying implementation is from the legacy system or the new modular system. Clients interact with the UserFacade to perform user-related operations without needing to understand the underlying architecture. This abstraction is particularly beneficial in large systems where different parts may evolve at different rates or where different teams manage different components.

By combining these patterns with the Puzzle Pattern, we ensure that the system is both flexible and maintainable. The adapter handles the compatibility issues, allowing new components to be integrated without altering the legacy code. The facade simplifies the API exposed to clients, promoting ease of use and reducing the learning curve associated with understanding the system's internal structure. This integration strategy not only enhances modularity but also ensures that the system can evolve gracefully, accommodating new features and technologies without disrupting existing functionalities.

5. Conclusions

In this paper, we introduced the Puzzle Pattern, a systematic approach to implementing multiple behavioral inheritance in object-oriented programming (OOP). Throughout our analysis, we elucidated how this pattern offered an innovative solution to some of the perennial issues associated with traditional OOP, such as tight coupling and rigid inheritance hierarchies, which often complicate system modifications and scalability. The Puzzle Pattern represents a significant departure from traditional OOP by advocating for the use of interfaces to encapsulate behavior, thus promoting extreme modularity and flexibility in software design. This approach not only facilitates the dynamic assembly of applications but also enhances system maintainability and scalability by allowing individual components to be developed, tested, and maintained independently. By decomposing complex systems into interchangeable interface-based components, or "puzzle pieces", developers

can achieve a higher degree of abstraction and reduce the coupling between components. This modularity enables easier system upgrades or changes without extensive rework, aligning perfectly with the principles of agile and responsive software development. The Puzzle Pattern not only enhances modularity and flexibility but also aligns closely with the SOLID principles, which are pivotal to creating maintainable and scalable software. Each interface in the Puzzle Pattern embodies the Single Responsibility Principle by encapsulating a distinct behavior or functionality, ensuring that each component has one reason to change. The Open/Closed Principle is adhered to as systems can introduce new functionalities through new interfaces without altering existing code, thereby remaining open for extension but closed for modification. The Liskov Substitution Principle is maintained as interfaces ensure that replaceable components behave in predictable ways. Interface Segregation is naturally achieved as the pattern encourages the use of specific interfaces rather than broad, catch-all interfaces that may not be relevant to all consuming classes. Lastly, the Dependency Inversion Principle is central to the Puzzle Pattern's architecture, which depends on abstractions rather than concrete implementations, promoting a decoupled and flexible interaction between higher-level and lower-level software components. Furthermore, the Puzzle Pattern supports incremental refactoring, making it particularly valuable for updating and enhancing legacy systems. This approach allows developers to systematically refactor a large, monolithic codebase into independently manageable modules. By implementing the Puzzle Pattern, teams can tackle refactoring in manageable phases, each phase improving part of the system without a complete overhaul, thereby reducing risk and minimizing disruption to existing operations. This phased approach not only facilitates smoother transitions and easier integration but also allows for continuous validation of system functionality and performance, ensuring that each step contributes positively to the overall system architecture. This strategy of incremental improvement is essential in maintaining system integrity and operability throughout the refactoring process, providing a practical pathway for legacy system modernization.

The Puzzle Pattern presents a robust strategy for enhancing code reuse and modularity by using interfaces to ship multiple behavioral inheritance, thereby reducing boilerplate and promoting shared functionality. However, this approach comes with its own set of challenges. It is best applied to small, atomic, and idempotent functionalities to avoid overcomplicating the architecture. The cardinality of "puzzle pieces" should be managed prudently to maintain code readability and maintainability. Additionally, the assembly of these pieces into a concrete class requires careful consideration to ensure coherence and compatibility. The Puzzle Pattern is particularly effective when complex functionalities can be decomposed into reusable components, but it necessitates a meticulous management of dependencies and alignment with the overarching class structure. Ultimately, while the Puzzle Pattern offers significant benefits in promoting a modular and maintainable codebase, it must be employed with a strategic understanding of its constraints to fully realize its potential.

Author Contributions: Conceptualization, M.G.; methodology, F.F. and M.G.; software, F.F. and M.G.; validation, F.F. and M.G.; formal analysis, F.F. and M.G.; investigation, F.F. and M.G.; resources, F.F. and M.G.; data curation, F.F. and M.G.; writing—original draft preparation, F.F. and M.G.; writing—review and editing, F.F. and M.G.; visualization, F.F. and M.G.; supervision, F.F. and M.G.; project administration, F.F. and M.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in the article itself (Appendix A).

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

This appendix contains the Java source code related to the project.

Source code 1: Java classes that describe the example scenario of implementing the Puzzle Pattern described in Section 3.

```
import java.util.Date;

// Interface for document acquisition functionality
interface DocumentAcquisitionPuzzlePiece {
    DatabaseService getDatabaseService();

    default boolean acquireDocument(Document document) {
        // Other business logic with pre-processing purposes
        return getDatabaseService().acquireDocument(document);
    }
}

// Interface for document deletion functionality
interface DocumentDeletionPuzzlePiece {
    DatabaseService getDatabaseService();

    default boolean deleteDocument(Document document) {
        // Other business logic with pre-processing purposes
        return getDatabaseService().deleteDocument(document);
    }
}

// Class that implements both document acquisition and deletion
    interfaces
class DocumentService implements DocumentAcquisitionPuzzlePiece,
    DocumentDeletionPuzzlePiece {
    private DatabaseService dbService;

    public DocumentService(DatabaseService dbService) {
        this.dbService = dbService;
    }

    @Override
    public DatabaseService getDatabaseService() {
        return this.dbService;
    }
}

// Service class for database operations
class DatabaseService {
    public boolean saveDocument(Document document) {
        // Placeholder for saving document logic
        return true;
    }

    public boolean deleteDocument(Document document) {
        // Placeholder for deleting document logic
        return true;
    }
```

```
}

// Data class for documents
class Document {
    private String content;
    private Date timestamp;
    private String type;

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(Date timestamp) {
        this.timestamp = timestamp;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Source code 2: UserManager.java class before applying the Puzzle Pattern.

```
public class UserManager {
    private DatabaseService dbService;

    public UserManager(DatabaseService dbService) {
        this.dbService = dbService;
    }

    public User getUserById(int userId) {
        return dbService.findUserById(userId);
    }

    public void updateUser(User user) {
        dbService.saveUser(user);
    }
}
```

Source code 3: UserDatabaseService.java class.

```
public class UserDatabaseService {
    public User findUserById(int userId) {
        // Simulated database access
```

```
        return new User(userId, "John Doe");
    }

    public void saveUser(User user) {
        // Simulated save operation
    }
}
```

Source code 4: User.java class.

```
public class User {
    private int id;
    private String name;

    public User(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // Getters and setters...
}
```

Source code 5: UserDataProcessor.java class.

```
public interface UserDataProcessor {
    DatabaseService getDatabaseService();

    default User getUserById(int userId) {
        return getDatabaseService().findUserById(userId);
    }

    default void updateUser(User user) {
        getDatabaseService().saveUser(user);
    }
}
```

Source code 6: UserManager.java class after applying the Puzzle Pattern.

```
public class UserManager implements UserDataProcessor {
    private DatabaseService dbService;

    public UserManager(DatabaseService dbService) {
        this.dbService = dbService;
    }

    @Override
    public DatabaseService getDatabaseService() {
        return dbService;
    }
}
```

Source code 7: Java classes that demonstrate an example of interoperability with Facade and Adapter patterns.

```
public class LegacyUserManager {
    public User getUserById(int userId) {
        // Simulated database access
```

```
        return new User(userId, "John Doe");
    }

    public void updateUser(User user) {
        // Simulated save operation
        System.out.println("Legacy system updating user: " + user
            .getName());
    }
}

public interface UserPersistencePuzzlePiece {
    DatabaseService getDatabaseService();

    default User findUserById(int userId) {
        return getDatabaseService().findUserById(userId);
    }

    default void saveUser(User user) {
        getDatabaseService().saveUser(user);
    }
}

public interface UserService {
    User getUserById(int userId);
    void updateUser(User user);
}

public class DatabaseService {
    public User findUserById(int userId) {
        // Placeholder for database access logic
        return new User(userId, "Jane Smith");
    }

    public void saveUser(User user) {
        // Placeholder for save operation logic
        System.out.println("New system saving user: " + user.
            getName());
    }
}

public class UserServiceImpl implements UserService,
    UserPersistencePuzzlePiece {
    private final DatabaseService dbService;

    public UserServiceImpl(DatabaseService dbService) {
        this.dbService = dbService;
    }

    @Override
    public DatabaseService getDatabaseService() {
        return dbService;
    }

    @Override
```

```
    public User getUserById(int userId) {
        return findUserById(userId);
    }

    @Override
    public void updateUser(User user) {
        saveUser(user);
    }
}

public class UserServiceAdapter implements UserService {
    private final LegacyUserManager legacyUserManager;

    public UserServiceAdapter(LegacyUserManager legacyUserManager
        ) {
        this.legacyUserManager = legacyUserManager;
    }

    @Override
    public User getUserById(int userId) {
        return legacyUserManager.getUserById(userId);
    }

    @Override
    public void updateUser(User user) {
        legacyUserManager.updateUser(user);
    }
}

public class UserFacade {
    private final UserService userService;

    public UserFacade(UserService userService) {
        this.userService = userService;
    }

    public User getUser(int userId) {
        return userService.getUserById(userId);
    }

    public void updateUser(User user) {
        userService.updateUser(user);
    }
}
```

Source code 8: Java main that demonstrate an example of interoperability with Facade and Adapter patterns.

```
public class Main {
    public static void main(String[] args) {
        // Using legacy system
        LegacyUserManager legacyUserManager = new
            LegacyUserManager();
        UserService legacyAdapter = new UserServiceAdapter(
            legacyUserManager);
```

```
            UserFacade legacyFacade = new UserFacade(legacyAdapter);

            User legacyUser = legacyFacade.getUser(1);
            System.out.println("Legacy User: " + legacyUser.getName()
                );
            legacyUser.setName("Updated Legacy User");
            legacyFacade.updateUser(legacyUser);

            // Using new system
            DatabaseService databaseService = new DatabaseService();
            UserService newUserService = new UserServiceImpl(
                databaseService);
            UserFacade newSystemFacade = new UserFacade(
                newUserService);

            User newUser = newSystemFacade.getUser(2);
            System.out.println("New System User: " + newUser.getName
                ());
            newUser.setName("Updated New System User");
            newSystemFacade.updateUser(newUser);
        }
    }
```

References

1. omg.org. OMG Meta Object Facility (MOF) Core Specification. Available online: https://www.omg.org/spec/MOF/2.4.1/PDF (accessed on 21 May 2024).
2. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison-Wesley: Boston, MA, USA, 1995.
3. MartinFowler.com. Inversion of Control Containers and the Dependency Injection Pattern. Available online: https://martinfowler.com/articles/injection.html (accessed on 21 May 2024).
4. Kerievsky, J. *Refactoring to Patterns*; Addison-Wesley Professional: Boston, MA, USA, 2005.
5. testing.googlegblog.com. How to Think about the "New" Operator with Respect to Unit Testing. Available online: https://testing.googleblog.com/2008/07/how-to-think-about-new-operator-with.html (accessed on 12 May 2024).
6. Martin, R.C. *Agile Software Development: Principles, Patterns, and Practices*; Prentice Hall: Upper Saddle River, NJ, USA, 2003.
7. Suryanarayana, G.; Samarthyam, G.; Sharma, T. *Refactoring for Software Design Smells: Managing Technical Debt*; Morgan Kaufmann: San Francisco, CA, USA, 2014.
8. Saltzer, J.H.; Schroeder, M.D. The protection of information in computer systems. *Proc. IEEE* **1975**, *63*, 1278–1308.
9. Bracha, G.; Cook, W. Mixin-based inheritance. *ACM Sigplan Not.* **1990**, *25*, 303–311.
10. Schärli, N.; Ducasse, S.; Nierstrasz, O.; Black, P.A. Traits: Composable Units of Behaviour. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2003; Volume 2743, pp. 248–274.
11. Ducasse, S.; Nierstrasz, O.; Schärli, N.; Wuyts, R.; Black, A.P. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.* **2006**, *28*, 331–388.
12. Richardson, C. *Microservices Patterns: With Examples in Java*; Manning Publications: New York, NY, USA, 2018.
13. Vernon, V. *Implementing Domain-Driven Design*; Addison-Wesley Professional: Boston, MA, USA, 2013.
14. Newman, S. *Building Microservices: Designing Fine-Grained Systems*; O'Reilly Media: Sebastopol, CA, USA, 2015.
15. Flores, A.P.; Moore, R. *GoF Structural Patterns: A Formal Specification*; The United Nations University: Tokyo, Japan, 2000.
16. Meyer, B. *Object-Oriented Software Construction*; Prentice Hall: Upper Saddle River, NJ, USA, 1988.
17. Liskov, B.H. Keynote address—Data abstraction and hierarchy. *ACM SIGPLAN Not.* **1987**, https://doi.org/10.1145/62139.62141.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.