Article

Not peer-reviewed version

Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in OOP

Francesca Fallucchi * and Manuel Gozzi

Posted Date: 2 May 2024

doi: 10.20944/preprints202405.0104.v1

Keywords: software engineering; software pattern; object oriented programming

Article

Puzzle Pattern, a Systematic Approach to Multiple Behavioral Inheritance Implementation in OOP

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

1 Università degli Studi Guglielmo Marconi
2 Università degli Studi Guglielmo Marconi; m.gozzi@studenti.unimarconi.it
* 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.

Simple Summary: A software design pattern that aims to reduce code complexity enhancing ease of development.

Abstract: Object-Oriented Programming (OOP) has long been a dominant paradigm in software development, but it's 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 interfaces. 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

1. Introduction

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of ”objects”, which encapsulate both state and behaviors. It has become a fundamental approach in software development, offering a more intuitive way to model complex systems, but despite its widespread adoption, OOP can present challenges: tight coupling, complex inheritance chains, and difficulties in modifying or extending code are some of the issues commonly associated with this paradigm. Tight coupling in Object-Oriented Programming (OOP) presents both technical and non-technical challenges that impact the development, maintenance, and scalability of software systems. From a technical perspective, tight coupling leads to a lack of modularization within the codebase. When classes or modules are tightly coupled, they become highly dependent on each other's implementations, making it difficult to isolate and modify individual components without affecting the entire system. This increases the risk of introducing bugs or unintended side effects when making changes, as developers must navigate complex interdependencies that span across different parts of the codebase. As a result, the development process becomes more error-prone, time-consuming, and resource-intensive. Furthermore, tight coupling inhibits code reusability and extensibility. Since tightly coupled components are closely intertwined, they cannot be easily reused in other contexts or extended to accommodate new requirements without significant modifications. This limits the flexibility of the software system and hinders its ability to evolve over time. Developers may find themselves duplicating code or implementing workarounds to circumvent tight coupling, leading to code bloat, decreased maintainability, and reduced overall code quality. On a non-technical level, tight coupling can also have organizational and collaborative implications. Teams working on tightly coupled codebases may face challenges in coordinating their efforts and collaborating effectively. The complexity introduced by tight coupling can make it difficult for team members to understand and reason about each other's code, leading to communication barriers and decreased productivity. Moreover, as the codebase grows larger and more intertwined, onboarding new team members becomes a daunting task, as they must familiarize themselves with the intricate relationships between various components. Addressing these challenges often involves implementing design patterns and best practices that promote modularity, reduce coupling, and simplify maintenance.

In this paper, we introduce Puzzle Pattern: an innovative approach that complements existing software design patterns for addressing challenges in Object-Oriented Programming (OOP). While solutions like Dependency Injection pattern[1], Observer pattern[2], Strategy pattern[2], and Composition over Inheritance[3] provide effective means to mitigate issues such as tight coupling, inheritance complexity, and encapsulation, the Puzzle Pattern offers a unique perspective on extreme modularity and code organization. Whereas traditional OOP often relies on concrete classes to represent functionality, the Puzzle Pattern advocates for writing code exclusively in interfaces. By doing so, it promotes a level of abstraction that goes beyond what other patterns typically offer. Rather than implementing functionality directly in classes, developers define interfaces that represent individual puzzle pieces of functionality. These interfaces serve as blueprints for concrete classes, which are then assembled through composition or dependency injection. This approach significantly reduces coupling by decoupling the implementation details from the interface definitions. It fosters a modular design where each puzzle piece can be developed, tested, and maintained independently. Moreover, the Puzzle Pattern promotes conceptual clarity by encouraging developers to focus on defining clear and cohesive interfaces, thus facilitating better communication and understanding among team members. By embracing extreme modularity and favoring interfaces over concrete classes, the Puzzle Pattern introduces a fresh perspective on software construction. It aligns well with the principles of Dependency Injection and Composition over Inheritance by promoting loose coupling and flexibility. Additionally, it complements patterns like the Observer pattern and Strategy pattern by providing a framework for organizing and assembling disparate pieces of functionality in a cohesive manner. The highlighted pattern presents numerous benefits within software development. Its focal point on extreme modularity by exclusively composing code in interfaces fosters a clearer division of responsibilities and simplifies upkeep. Leveraging multiple inheritance enables a class's behavior to amalgamate from various interfaces, thereby augmenting adaptability in system architecture. Moreover, the pattern upholds the SOLID principles[4], championing a sturdy and expandable framework. Nevertheless, akin to any methodology, it brings forth its set of hurdles. Relying on multiple inheritance might introduce intricacy, necessitating vigilant management of shared states among interfaces. Furthermore, while striving for compatibility with conventional Object-Oriented methods, teams accustomed to more orthodox practices may encounter a learning curve.

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 play a crucial role in addressing issues such as tight coupling in Object-Oriented Programming (OOP) by providing proven solutions to common problems encountered in software development. These patterns encapsulate best practices, design principles, and architectural guidelines that promote modularity, flexibility, and maintainability in software systems. One of the primary ways design patterns mitigate tight coupling is by promoting loose coupling between components. Furthermore, design patterns facilitate the creation of flexible and extensible software architectures by promoting principles such as encapsulation, abstraction, and separation of concerns, providing a common language and set of solutions that facilitate communication and collaboration among developers. By following established patterns, developers can convey design decisions, communicate intentions, and share knowledge more effectively. This fosters a culture of best practices and design consistency within development teams, leading to higher code quality, improved maintainability, and enhanced productivity. The biggest impact in this field came from the Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) software patterns[1] in the 1990s. The GoF introduced 23 design patterns categorized into creational, structural, and behavioral patterns. For example, the Dependency Injection pattern decouples classes by externalizing their dependencies, allowing them to be provided from external sources rather than being instantiated internally. This reduces the direct dependencies between classes, making them more modular and easier to test, maintain, and extend, as dependencies can be replaced or mocked as needed. This pattern promotes the principle of ”programming to interfaces, not implementations,” thereby reducing coupling and enhancing flexibility. Similarly, the Observer pattern decouples subjects and observers, enabling them to communicate without being directly dependent on each other’s implementations. In this pattern, objects (observers) subscribe to changes in another object (subject) and are notified whenever there is a state change. By decoupling the observing objects from the subject, changes in one component do not directly impact others, promoting a more loosely coupled architecture. The Strategy pattern provides a way to encapsulate interchangeable algorithms within their own classes, allowing clients to switch between different algorithms dynamically without modifying client code. This promotes code reuse, modularity, and extensibility, as individual algorithms can be developed, tested, and maintained independently.

Beyond tight coupling, another challenge in OOP is managing inheritance hierarchies. While inheritance can facilitate code reuse and promote a hierarchical structure, it can also lead to overly complex and fragile designs, known as the ”fragile base class problem”. As systems evolve, changes in base classes can inadvertently affect derived classes, potentially introducing bugs and making maintenance difficult. To address this issue, the Composition over Inheritance principle advocates for favoring object composition over class inheritance, enabling developers to compose objects dynamically at runtime rather than relying on static inheritance relationships. This promotes a more modular and composable design, mitigating the risks associated with inheritance hierarchies and tight coupling. By composing objects dynamically at runtime rather than relying on static inheritance relationships, developers can achieve greater flexibility and resilience to change. This approach promotes a more modular and composable design, mitigating the risks associated with inheritance hierarchies. In terms of encapsulation, while OOP provides mechanisms such as access modifiers (e.g., public, private, protected), achieving true encapsulation can be challenging. Objects often have access to each other’s internal state, leading to tight coupling and potential unintended side effects. To mitigate this, the Principle of Least Privilege suggests restricting access to the bare minimum necessary for functionality, thereby reducing the risk of unintended interactions and enhancing encapsulation.

3. Puzzle Pattern Conceptualization

A puzzle consists of two main elements: a board and a set of pieces. The pieces are assembled on the board and together form a complex design. However, if we just look at the individual pieces, we will notice that each of them encloses only a small part of the picture. Considering this analogy, to which the pattern is named, we can associate the puzzle pieces with the interfaces, and the puzzle board with the concrete class. A concrete class, a puzzle board, takes on a definite meaning only when all the pieces are in place, but each piece is in its own way independent of the puzzle board itself. The idea behind the Puzzle Pattern is to develop code within interfaces (the puzzle pieces), and to assemble concrete classes (the puzzle boards) by implementing interfaces. Each interface is intended to implement a behavior, and to declare abstract all the requirements that need to be met in order to realize the developed behavior. The requirements are intended to be provided by the concrete class. In other words, each abstract method need only return the instance that can provide the functionality required in the concrete method. The behavior is given by the interface itself, which exploits the state implemented by the class that implements the interface itself. In the UML class diagram below, the class C is composed by the union of I and J: $C = I \cup J$. The behavior of class C is determined by a combination of interfaces I and J facilitated through the use of behavioral multiple inheritance. The only task required of class C is to ensure that the appropriate state is made available to both interfaces I and J.

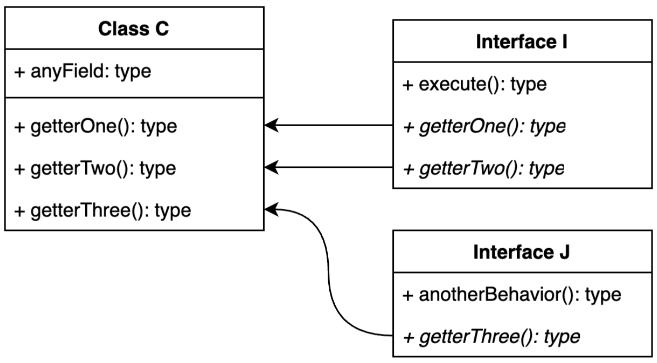

Figure 1. UML class diagram that formalize the foundamental pattern concept.

The fundamental objectives of Puzzle Pattern are clearly outlined in order to guide the developer in achieving specific results and improvements over conventional approaches. Key objectives include modularity, flexibility, loose coupling, conceptual clarity, gradual project growth, and maintaining compatibility with the traditional OO approach. The primary goal is to promote extreme modularity in the software development process. By focusing on writing code exclusively in interfaces, the aim is to reduce the degree of dependency between concrete classes, thus making it easier to manage and maintain the system. Allowing flexible assembly of concrete classes by implementing interfaces provides a greatly enhanced level of adaptability. Developers can compose classes according to specific project requirements, allowing precise customization without having to modify the source code of existing classes. Reducing the degree of coupling between concrete classes is one of the central goals. Minimizing dependencies between system components improves maintainability, testability, and ease of software evolution over time. Improve conceptual clarity in software design. Focusing on writing code in interfaces attempts to more accurately reflect the conceptual structure of the system, facilitating understanding and collaboration among development team members. Fostering the scalability of the system is a key objective. The modular structure of the pattern should allow for smooth and gradual growth of the system, enabling smooth management of even continuously evolving projects. Ensuring compatibility with the traditional Object-Oriented approach is an important aspect. The proposed pattern should integrate smoothly with existing code written according to OO conventions, allowing a seamless transition. These goals have been outlined with the aim of addressing specific challenges encountered in software projects and introducing a paradigm that can contribute significantly to the improvement of software design and maintenance.

3.1. SOLID Principle Compliance

The SOLID principles —Single Responsibility[5], Open-Closed[5], Liskov Substitution [6], Interface Segregation[2] and Dependency Inversion[7] —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 rather both depend 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.

4. Use Case: The Puzzle Pattern Applied in the CRUD Operations

In the following section, as a result the application of the above Puzzle Pattern in the common case of CRUD[8] scenario implementation is demonstrated. The CRUD operations are a perfect example case for the application of the Puzzle Pattern. With ”CRUD” operations here we intend the basic Create, Read, Update and Delete operations that are executed on the database side in order to update a logical entity state. The type of the database does not impact the pattern adoption. Let’s visualize an analogy between the CRUD scenario and the Puzzle Pattern main concepts: boards and pieces. In order to fulfill the implementation, we need a board and four pieces (create, read, update and delete). The board is the concrete class that connects all the pieces together, forming a complex entity, while the pieces are atomic functionalities. In order to describe the pattern application, we base our exposition on top of a simplistic Person Java class (appendix A1).

We start defining the CRUD puzzle pieces, whose aim is to handle the data-layer operations of the generic type ”T”. The first piece where we start from is C. C is responsible for the creation of a new entity on an underlying data-layer. Let’s assume that the "R" repository includes an "insert()" method. This method takes an instance of "T" as an argument and, upon the creation activity, returns the updated instance of "T" (appendix A.2). We define the "create()" method in "C" by calling "insert()" on the data-layer. Then, let’s define "R", that is responsible for the data-reading operations. Basically, we use "R" piece to fetch data from the data-layer. Assume that the generic type "T" possesses a method "getId()" that can return an object of type "ID". Let us also assume that the repository "R" possesses a method "find()" that receives as an argument an instance of "ID", returning "T" if present (appendix A.3). We define the "read" method in "R" by calling "insert()" on the data-layer. Let’s move along with U. Let’s also presume that the "R" repository features an "update()" method, wherein it takes an instance of "T" as a parameter. After the update operation, it returns the modified instance of "T" (appendix A.4), updating the "T" instance state on the data-layer. We define the "update" method in "U" by calling its homonym method on the data-layer. Finally, let’s assume that the "R" repository possesses a method "deleteById()" that receives as an argument an instance of "ID", softly deleting the related entity from the data layer (appendix A.5). We define the "delete" method in "D" by calling "deleteById()" on the data-layer. The Puzzle Pattern can now be applied by creating the CRUD class - the puzzle board. The action needed here is to assemble the board by implementing all the CRUD pieces mentioned before (appendix A.6). To cover the class with unit tests, it is sufficient to add a unit test for each interface functionality and add a unique one at the CRUD class level to verify that the returned repository instance conforms to expectations. After assembling the concrete CRUD class for the Person class with the "ID" of type Long, a similar process could be applied for another class C2 by implementing the same interfaces, provided that C2 also includes the "getId()" method returning a result of type Long. Moreover, it would be easily possible to make hybrid classes that implement only some of the CRUD functions (maybe just the create and delete).

4.1. Discussion

The Puzzle Pattern was explained with the help of a simplistic, but nevertheless factual and real-world example. The same pattern can be employed in literally any context, but it is obvious to agree that the greatest benefit is obtained by exploiting the multiple behavioral inheritance paradigm. In cases where multiple inheritance is not applicable, application of this pattern could cause over-engineering problems. As an example, the Java programming language was used, but the same concepts can of course be applied with other technologies (Python, Scala, C++ or C# just to name a few). Java is an excellent candidate, as it has a very robust compiler that, thanks to the use of generics, makes it possible to resolve any conflicts when implementing interfaces in a completely safe and automatic manner. It is also important to assert that hierarchies could be established between interfaces. In the example case of CRUD operations all pieces of the puzzle have the same requirement: having access to a repository R that satisfies the signature of the generics. In this case, a further optimization might be to declare an additional abstraction as represented below by the RepositoryBasedPuzzlePiece interface. With this in mind, the interfaces CreatePuzzlePiece, ReadPuzzlePiece, UpdatePuzzlePiece, and DeletePuzzlePiece could extend RepositoryBasedPuzzlePiece (appendix A.7). This further testifies to the degree of flexibility and modularity that can be achieved through the implementation of such a pattern. By continuing to code at the interface level, moreover, it is possible to push the principle of Dependency Inversion as far as possible, forcing the software engineer to develop on interfaces rather than concrete classes. More atomic functionality also incentives code reuse and limits the risk of writing duplicate code.

Finally, it should be mentioned that in the example cases shown, the business logic inherent in the interfaces was found to be minimal in order to be able to facilitate explanation. However, there may be more complex application cases (e.g., the recording of user audits in conjunction with the execution of specific operations) that require the interaction of multiple services. The Puzzle Pattern makes it possible to do this; all that is required is to declare in the puzzle piece of interest an abstract method, the implementation of which must conform to a puzzle board-side ”getter” designed to provide the instance that implements the required functionality. When writing the concrete methods, sites in the interfaces, normal best practices should be adhered to; no veto is imposed in this regard.

5. Conclusions

The presented pattern offers several advantages in software development. Its emphasis on extreme modularity through writing code exclusively in interfaces promotes a clearer separation of concerns and facilitates maintenance. The use of multiple inheritance allows the behavior of a class to be defined by a combination of interfaces, enhancing flexibility in system design. Additionally, the pattern adheres to the SOLID principles, promoting a robust and scalable architecture. However, like any approach, it comes with its challenges. The reliance on multiple inheritance may introduce complexity, and developers need to carefully manage the state shared among interfaces. Furthermore, while the pattern aims for compatibility with the traditional Object-Oriented approach, there may be a learning curve for teams accustomed to more conventional practices. In summary, the pattern provides a promising avenue for modular and scalable software design, but its successful implementation requires careful consideration of its complexities and potential learning curve.

In closing, the proposed pattern represents an innovative response to software design challenges by synergistically integrating SOLID principles with a modular approach in the context of CRUD operations. Practical application of this pattern in the example of CRUD operations demonstrated its effectiveness in promoting modularity, flexibility, and conceptual clarity. Adherence to SOLID principles was evidenced through the clear separation of responsibilities in interfaces, enabling efficient management of CRUD operations. Extreme modularity was manifested in the way concrete classes are assembled through the implementation of atomic interfaces, ensuring conceptual clarity and gradual evolution of the system. The application of the pattern to CRUD operations has shown that greater flexibility in software design can be achieved, allowing specific, customized services to be built without affecting pre-existing code. This modular approach can be particularly advantageous in scenarios where system requirements can change rapidly, allowing rapid adaptability to new demands. The presented pattern emerges as a valuable resource for developers seeking to improve maintainability, flexibility, and conceptual clarity in their software projects. Its adherence to SOLID principles, combined with clear practical application in CRUD operations, positions it as a significant contribution to the field of modular and sustainable software development.

Author Contributions: Conceptualization, Gozzi Manuel; methodology, Fallucchi Francesca and Gozzi Manuel; software, Fallucchi Francesca and Gozzi Manuel; validation, Fallucchi Francesca and Gozzi Manuel; formal analysis, Fallucchi Francesca and Gozzi Manuel; investigation, Fallucchi Francesca and Gozzi Manuel; resources, Fallucchi Francesca and Gozzi Manuel; data curation, Fallucchi Francesca and Gozzi Manuel; writing—original draft preparation Fallucchi Francesca and Gozzi Manuel; writing—review and editing, Fallucchi Francesca and Gozzi Manuel; visualization, Fallucchi Francesca and Gozzi Manuel; supervision, Fallucchi Francesca and Gozzi Manuel; project administration, Fallucchi Francesca and Gozzi Manuel; funding acquisition, no funds. 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.

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

Appendix A

Appendix A.1

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

Source code 1: Person.java class.

```java
public class Person {

  private Long id;
  private String fullName;
  private java.util.Date dateOfBirth;

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

Source code 2: CreatePuzzlePiece.java class.

```java
public interface CreatePuzzlePiece<T, ID, R extends Repository<T, ID>> {

  R repository();

  default T create(T newInstance) {
    // Saving instance...
    return this.repository().insert(newInstance);
  }
}
```

Source code 3: ReadPuzzlePiece.java class.

```java
public interface ReadPuzzlePiece<T, ID, R extends Repository<T, ID>> {

  R repository();

  default T read(ID id) {
    // Finding instance...
    return this.repository().find(id);
  }
}
```

Source code 4: UpdatePuzzlePiece.java class.

```
public interface UpdatePuzzlePiece<T, ID, R extends Repository<T, ID>> {

  R repository();

  default T update(T tInstance) {
    // Saving instance...
    return this.repository().update(tInstance);
  }
}
```

Source code 5: DeletePuzzlePiece.java class.

```
public interface DeletePuzzlePiece<T, ID, R extends Repository<T, ID>> {

  R repository();

  default void delete(ID id) {
    // Deleting instance...
    this.repository().deleteById(id);
  }
}
```

Source code 6: CRUDPuzzleBoard.java class.

```
public class CRUDPuzzleBoard implements CreatePuzzlePiece<Person, Long,
PersonRepository>,
    ReadPuzzlePiece<Person, Long, PersonRepository>,
    UpdatePuzzlePiece<Person, Long, PersonRepository>,
    DeletePuzzlePiece<Person, Long, PersonRepository> {

  private final PersonRepository repository;

  public CRUD(PersonRepository repository) {
    this.repository = repository;
  }

  @Override
  public PersonRepository repository() {
    return this.repository;
  }
}
```

Source code 7: RepositoryBasedPuzzlePiece.java class.

```
public interface RepositoryBasedPuzzlePiece<T, ID, R extends
Repository<T, ID>> {

    R repository();
}
```

References

1. Fowler, M. *Patterns of enterprise application architecture*; Addison-Wesley, 2013.
2. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. *Design patterns: Elements of reusable object-oriented software*; Addison-Wesley, 1995.
3. Freeman, S., & Bates, D. *Head first design patterns*; O'Reilly Media, 2009.
4. Martin, R. C.; *Design principles and patterns*; Object Technology International, 2000.
5. Meyer, B.; *Object-oriented software construction*; Prentice Hall, 1988.
6. Liskov, B. H.; Keynote address - Data abstraction and hierarchy. *ACM SIGPLAN Notices* 1987, *22(10)*.
7. Martin, R. C.; *Agile software development: Principles, patterns, and practices.*; Prentice Hall, 2003.
8. Martin, J.; *Managing the data-driven organization.*; Prentice-Hall, 1983.