Journal of Advances in Developmental Research (IJAIDR)
E-ISSN: 0976-4844 ● Website: www.ijaidr.com ● Email: editor@ijaidr.com

An Examination of the Adapter Design Pattern and Its Use in Object-Oriented System Interoperability

Arun Neelan

Independent Researcher
PA, USA
arunneelan@yahoo.co.in

Abstract

The Adapter Design Pattern is a fundamental structural pattern in object-oriented design that enables the integration of incompatible interfaces without requiring changes to existing code. As software systems evolve and grow in complexity, the demand for interoperability between legacy modules, third-party components, and modern architectures becomes increasingly critical. This review paper provides a comprehensive exploration of the Adapter pattern from both theoretical and practical perspectives. It examines the two primary variants—Class Adapter and Object Adapter—as well as the Two-Way Adapter, which facilitates bidirectional communication. Through the use of class diagrams and real-world examples, the paper illustrates the practical applications of each variant and discusses their respective trade-offs. Additionally, it highlights the benefits of using the Adapter pattern, such as enhanced code reusability and system flexibility, alongside potential drawbacks like increased abstraction and the risk of overuse. Best practices for effectively implementing the Adapter pattern in modern software development are also presented. This review aims to deepen understanding of the Adapter pattern's role in building modular, maintainable, and extensible software systems.

Keywords: Adapter Design Pattern, Object Adapter, Class Adapter, Software Design Patterns, Structural Design Patterns, Object-Oriented Design, Software Engineering

I. INTRODUCTION

As modern software systems grow in complexity and scale, the need for seamless integration across diverse systems and components becomes a constant challenge. As applications scale and incorporate legacy code, third-party libraries, or modern APIs, developers frequently encounter incompatible interfaces that complicate interoperability. To address such recurring architectural issues, design patterns offer proven, reusable solutions.

Among structural design patterns, the Adapter Pattern plays a critical role in enabling communication between otherwise incompatible interfaces—without requiring changes to existing code. By acting as a translator between different modules, it allows independently developed components to interact through a unified interface. This facilitates not only interoperability but also enhances modularity, reusability, and long-term maintainability in software systems.

This review explores the Adapter pattern in depth, focusing on its main variants—Class Adapter, Object Adapter, and Two-Way Adapter. It highlights practical implementation strategies, real-world examples, and the trade-offs involved. By examining common use cases and best practices, the paper aims to provide a comprehensive understanding of the Adapter pattern's role in contemporary software development.

II. ADAPTER PATTERN

The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces [1, p. 139]. In essence, the adapter pattern facilitates the integration of disparate interfaces by acting as a bridge, converting one interface into another to allow communication between classes or systems that would otherwise not be able to interact.

This concept can be illustrated using the analogy of a plug adapter. Consider an electrical device with a specific plug type, such as a US plug, which must be connected to a socket of a different type, such as a European socket. In this case, an adapter is used to make the US plug compatible with the European socket. In software development, the adapter pattern operates similarly. It enables two distinct systems or components to interact without necessitating changes to their original code. Through this intermediary, incompatible interfaces or systems are integrated, allowing them to communicate effectively.

Below class diagram illustrates the (Object) Adapter pattern.

The Adapter Design Pattern is demonstrated in the example below, where key concepts like the adapter, target interface, and client are implemented. The adapter acts as a bridge, allowing the client to interact with the class seamlessly without modifying its original structure, thus enhancing flexibility and reusability in systems with diverse interfaces.

The adapter pattern can be classified into two main types: object adapter and class adapter.

A. Object Adapter

An object adapter relies on composition rather than inheritance. In this pattern, the adapter maintains a reference to an instance of the adaptee class and delegates method calls to it [2, p. 246]. This approach, as illustrated in earlier sections through class diagrams and examples, offers greater flexibility. By avoiding the constraints of inheritance, it accommodates a wider range of use cases. Object adapters are especially useful in programming languages that do not support multiple inheritance. They also allow for more advanced patterns, such as wrapping multiple adaptee types or implementing two-way adapters for bidirectional communication between incompatible systems. Below explains two-way adapters with an example.

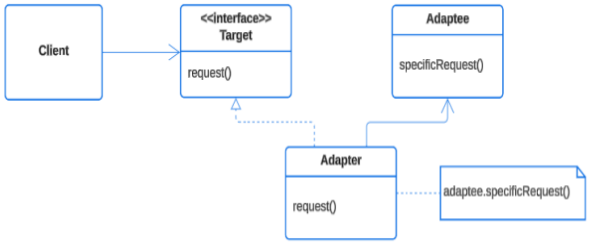

Fig. 1 – Object Adapter Pattern

```
//Target interface that defines the operations the
//client expects. The Adapter will implement this
//interface and translate calls to the Adaptee's methods.
public interface FileSystem {

    // Defines the saveFile operation that the client will
    // call to save a file. The Adapter will implement this
    // method and map it to the Adaptee's corresponding method.
    void saveFile(String fileName, String content);
}
```

Listing 1. Object Adapter Pattern - Target Interface

```
//Adaptee class with an existing implementation that may not match
//the Target interface. The Adapter will translate calls from the
//Target interface to this class's methods.
public class LocalFileSystem {
    // Existing method in the Adaptee that saves a file.
    // The Adapter will map this method to the saveFile() method of the Target
    // interface.
    public void writeToLocalFile(String fileName, String content) {
        // Simulate saving a file to the local system.
        System.out.println("Saving file to local system: " + fileName);
    }
}

//Adapter class that implements the FileSystem interface and adapts the
LocalFileSystem.
public class LocalFileSystemAdapter implements FileSystem {
    private LocalFileSystem localFileSystem;
    public LocalFileSystemAdapter(LocalFileSystem localFileSystem) {
        this.localFileSystem = localFileSystem;
    }

    // Implements the saveFile method from FileSystem interface.
    // This method can include additional logic like validation, logging etc.,
    // before/after calling the Adaptee.
    @Override
    public void saveFile(String fileName, String content) {
        // Additional logic (e.g., validation, logging) can be added here.
        // Call to the Adaptee's method to save the file.
        localFileSystem.writeToLocalFile(fileName, content);

        // Additional logic, as required can be added here.
    }
}
```

Listing 2. Object Adapter Pattern - Adaptee - 1 and Adapter - 1

```
//Adaptee class with an existing implementation that may not match the
Target interface.
//The Adapter will translate calls from the Target interface to this class's
methods.
public class AmazonS3 {
    // Existing method in the Adaptee that uploads a file to Amazon S3.
    // The Adapter will map this method to the saveFile() method of the
Target
    // interface.
    public void uploadToS3(String fileName, String content) {
        // Simulate uploading a file to Amazon S3.
        System.out.println("Uploading file to Amazon S3: " + fileName);
    }
}


//Adapter class that implements the FileSystem interface and adapts the
AmazonS3.
public class AmazonS3Adapter implements FileSystem {
    private AmazonS3 amazonS3;
    public AmazonS3Adapter(AmazonS3 amazonS3) {
        this.amazonS3 = amazonS3;
    }

    // Implements the saveFile method from FileSystem interface.
    // This method can include additional logic like validation, logging,
etc.,
    // before/after calling the Adaptee.
    @Override
    public void saveFile(String fileName, String content) {
    // Additional logic (e.g., validation, logging) can be added here.

    // Call to the Adaptee's method to upload the file to Amazon S3.
    amazonS3.uploadToS3(fileName, content);

    // Additional logic, as required can be added here.
    }
}
```

Listing 3. Object Adapter Pattern - Adaptee - 2 and Adapter - 2

```
//Client class that uses the FileSystem interface to save files.
//It demonstrates the use of the Adapter pattern to work with different file
systems.
public class FileManager {
   public static void main(String[] args) {
   // Create an instance of LocalFileSystemAdapter to adapt the
LocalFileSystem to the FileSystem interface.
   FileSystem localFileSystem = new LocalFileSystemAdapter(new
LocalFileSystem());

   // Create an instance of AmazonS3Adapter to adapt AmazonS3 to the
FileSystem interface.
   FileSystem amazonS3FileSystem = new AmazonS3Adapter(new
AmazonS3());

   // Call saveFile on localFileSystem, which is adapted from
LocalFileSystem to FileSystem interface.
   localFileSystem.saveFile("localfile.txt", "Content of local file.");

   // Call saveFile on amazonS3FileSystem, which is adapted from
AmazonS3 to FileSystem interface.
   amazonS3FileSystem.saveFile("cloudfile.txt", "Content of file on S3.");
   }
}
```

Output

```
Saving file to local system: localfile.txt
Uploading file to Amazon S3: cloudfile.txt
```

Listing 4. Object Adapter Pattern - Client & Output

```
// Interface for OrderProcessingSystem (expects JSON).
interface OrderProcessor {
   void sendOrderInJson(String orderDetails);
}


// Interface for InventorySystem (expects XML).
interface InventoryManager {
   void sendInventoryInXml(String inventoryDetails);
}
```

Listing 5. Two-Way Adapter Pattern - Interfaces

```
// Adaptee for OrderProcessingSystem (works with JSON).
class JsonOrderProcessor implements OrderProcessor {
   @Override
   public void sendOrderInJson(String orderDetails) {
      System.out.println("OrderProcessingSystem sending order in JSON: "
+ orderDetails);
   }
}


// Adaptee for InventorySystem (works with XML).
class XmlInventoryManager implements InventoryManager {
   @Override
   public void sendInventoryInXml(String inventoryDetails) {
      System.out.println("InventorySystem sending inventory in XML: " +
inventoryDetails);
   }
}
```

Listing 6. Two-Way Adapter Pattern - Adaptees

```
// Two-Way Adapter that adapts communication in both directions.
class CommunicationAdapter implements OrderProcessor,
InventoryManager {
    private OrderProcessor orderProcessor;
    private InventoryManager inventoryManager;

    public CommunicationAdapter(OrderProcessor orderProcessor,
InventoryManager inventoryManager) {
        this.orderProcessor = orderProcessor;
        this.inventoryManager = inventoryManager;
    }

    // Converts OrderProcessingSystem's JSON message to
InventorySystem's XML format.
    @Override
    public void sendOrderInJson(String orderDetails) {
        String xmlOrderDetails = "<order>" + orderDetails + "</order>";
        inventoryManager.sendInventoryInXml(xmlOrderDetails);
    }

    // Converts InventorySystem's XML message to
OrderProcessingSystem's JSON format.
    @Override
    public void sendInventoryInXml(String inventoryDetails) {
        String jsonInventoryDetails = "{ \"inventory\": \"" + inventoryDetails
+ "\" }";
        orderProcessor.sendOrderInJson(jsonInventoryDetails);
    }
}
```

Listing 7. Two-Way Adapter Pattern – Two-Way Adapter class

```
// Client Code.
public class CommunicationSystem {
    public static void main(String[] args) {
        OrderProcessor orderProcessor = new JsonOrderProcessor();
        InventoryManager inventoryManager = new XmlInventoryManager();

        // Create the Two-Way Adapter to facilitate communication in both
directions.
        CommunicationAdapter adapter = new
CommunicationAdapter(orderProcessor, inventoryManager);

        // OrderProcessingSystem sends order in JSON format.
        adapter.sendOrderInJson("Order #12345, Item: Laptop, Quantity: 1");

        // InventorySystem sends inventory data in XML format.
        adapter.sendInventoryInXml("<inventory>Item: Laptop, Quantity:
100</inventory>");
    }
}
```

Listing 8. Two-Way Adapter Pattern – Client Usage

B. Class Adapter

A class adapter employs inheritance, where the adapter class inherits from both the target interface and the adaptee class [2, p. 246]. This inheritance allows the adapter to translate calls between the two interfaces. This approach is commonly used in languages that support multiple inheritance.

The following section explains the class adapter pattern using a simple example, demonstrating how inheritance is used to adapt one interface to another.

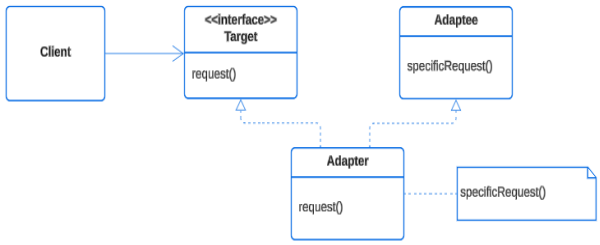

Fig. 2 – Class Adapter Pattern

```
// Target Interface.
// Defines the method expected by the client (EuropeanSocket).
interface EuropeanSocket {
        void connect();
}


// Adaptee (Existing Class).
// Represents the legacy system (US plug), with its own method.
class USPlug {
        public void connectUS() {
                System.out.println("Connected using US plug.");
        }
}
```

Listing 9. Class Adapter Pattern – Target

```
// Adapter Class.
// Inherits from USPlug (adaptee) and implements EuropeanSocket
(target).
// Adapts the incompatible US plug to work with the European socket.
class PlugAdapter extends USPlug implements EuropeanSocket {

    // Adapts the connect method to call the US plug's method.
    @Override
    public void connect() {
        System.out.println("Using adapter for European socket...");
        connectUS();
    }
}


// Client Code.
// Uses EuropeanSocket interface, unaware of the US plug
implementation.
public class Travel {
    public static void main(String[] args) {
        EuropeanSocket europeanSocket = new PlugAdapter();
        europeanSocket.connect(); // Adapter handles the compatibility
    }
}
```

Listing 10. Class Adapter Pattern – Adapter and Client Usage.

C. *Adapter Pattern in Java Libraries*

Below given are a few examples of adapter pattern in the Java standard library:

- *java.util.Arrays#asList():*
  String[] array = {"A", "B", "C"};
  List<String> list = Arrays.asList(array);

  Arrays.asList(array) acts as an adapter that converts a String[] array into a List<String>. This allows the array to be used with APIs or methods that expect a List interface, promoting compatibility without modifying the original array structure. For more information on the Arrays class and its methods, refer to [3].

- *java.io.InputStreamReader and OutputStreamWriter:*
  InputStream inputStream = new FileInputStream("data.txt");
  Reader reader = new InputStreamReader(inputStream);

  InputStreamReader adapts a byte-based InputStream to a character-based Reader, decoding bytes into characters using a specified charset [4] [5]. Similarly, OutputStreamWriter adapts a character-based Writer to a byte-based OutputStream, encoding characters into bytes [6] [7]. This enables smooth communication between byte-oriented and character-oriented I/O APIs.

- *java.util.Collections#list(Enumeration):*
  Enumeration<String> e = ...;
  List<String> list = Collections.list(e);

  Collections.list(e) acts as an adapter that converts a legacy Enumeration<String> into a modern List<String>. This allows the enumeration to be used with newer collection-based APIs, bridging the gap between old and new Java interfaces [8] [9].

D. *Benefits of the Adapter Pattern*

The Adapter Pattern offers several key benefits that simplify the interaction between incompatible interfaces in object-oriented systems. Below are the primary benefits:

Topic	***Detail***
Interface Compatibility & Code Reusability	Allows incompatible interfaces to work together by providing a wrapper that converts one interface into another, enabling seamless integration of components without altering their internal code [1, p. 140].
Encapsulation of Complexity	Simplifies client code by abstracting away the complexities of converting between incompatible interfaces, so the client doesn't have to manage the details.
Decoupling & System Flexibility	Decouples client code from external system specifics, enhancing flexibility, maintainability, and making it easier to adapt or extend the system without affecting the client.

System Extensibility & Legacy Integration	Facilitates the addition of new systems or integration with legacy systems without altering existing code, ensuring smooth system evolution and backward compatibility.

E. Challenges and Best Practices

While the Adapter Pattern offers several benefits, it also introduces challenges that can impact the flexibility and maintainability of a system. Below are the key challenges along with strategies to mitigate them:

Topic	***Challenge***	***Best Practice***
Increased Complexity & Overhead	The Adapter Pattern can introduce additional complexity and performance overhead, particularly when there are many adapters or when the system is performance-sensitive.	Minimize the logic within adapters, avoid unnecessary adapters, and use them only in critical areas where integration is required.
Excessive Use of Adapters	Overusing adapters can lead to a fragmented design, making the system harder to manage and extend.	Use adapters only where necessary, and keep the design minimal to prevent excessive abstraction layers.
Reduced Code Clarity	Adapters can obscure the flow of control and make system interactions less clear to developers.	Provide thorough documentation, and use clear naming conventions to ensure the purpose and role of adapters are easily understood.

III. CONCLUSION

The Adapter Design Pattern is a fundamental tool in software development, enabling systems with incompatible interfaces to communicate without the need to modify existing code. As software systems grow more complex, often incorporating legacy systems, third-party libraries, or modern APIs, the Adapter pattern becomes essential for ensuring smooth integration between diverse components.

This review has examined the core variants of the Adapter pattern—Class Adapter, Object Adapter, and Two-Way Adapter—demonstrating how each can be applied to address different integration challenges. Through real-world examples and class diagrams, the review highlights how these patterns enhance code reusability, system flexibility, and maintainability. The inclusion of examples from the Java standard library further underscores the pattern's relevance, showcasing its application in widely used frameworks and real-world development environments.

However, as with any design pattern, the Adapter pattern has its challenges. While it introduces abstraction that simplifies integration, overuse can lead to unnecessary complexity. It’s crucial to weigh the trade-offs carefully and apply the pattern thoughtfully, ensuring that its benefits outweigh potential drawbacks.

In summary, the Adapter pattern remains a vital tool in modern software architecture, especially for projects that require the integration of diverse systems and components. Its ability to promote flexibility, modularity, and long-term maintainability continues to make it an invaluable asset for developers.

REFERENCES

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

[2] E. Freeman and E. Robson, Head first design patterns: Building Extensible and Maintainable Object-Oriented Software. 2021.

[3] Oracle, "Class Arrays (Java SE 11 & JDK 11)," Oracle. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html

[4] Oracle, "Class FileInputStream (Java Platform SE 8)," Oracle. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/io/FileInputStream.html

[5] Oracle, "Class InputStreamReader (Java Platform SE 8)," Oracle. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html

[6] Oracle, "Class FileOutputStream (Java Platform SE 8)," Oracle. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/io/FileOutputStream.html

[7] Oracle, "Class OutputStreamWriter (Java SE 11 & JDK 11)," Oracle. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStreamWriter.html

[8] "Java Platform SE 8." Available: https://docs.oracle.com/javase/8/docs/api/?java/util/Collections.html

[9] Oracle, "Enumeration (Java SE 8)," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html.