**Adapter Design Pattern:** The Adapter design pattern is a [structural pattern](https://www.geeksforgeeks.org/structural-design-patterns/) that allows the interface of an existing class to be used as another interface. It acts as a bridge between two incompatible interfaces, making them work together. This pattern involves a single class, known as the adapter, which is responsible for joining functionalities of independent or incompatible interfaces.

**Example:**

* *Let’s say you have two friends, one who speaks only English and another who speaks only French. You want them to communicate, but there’s a language barrier.*
* *You act as an adapter, translating messages between them. Your role allows the English speaker to convey messages to you, and you convert those messages into French for the other person.*
* *In this way, despite the language difference, your adaptation enables smooth communication between your friends.*
* *This role you play is similar to the Adapter design pattern, bridging the gap between incompatible interfaces.*
* *Power Adapter & socket*
* *Xml to Json Conversion*
* *Convert to pounds to weight in a weighing machine.*

**Components of Adapter Design Pattern**

**1. Target Interface**

* **Description:** Defines the interface expected by the client. It represents the set of operations that the client code can use.
* **Role:** It’s the common interface that the client code interacts with.

**2. Adaptee**

* **Description:** The existing class or system with an incompatible interface that needs to be integrated into the new system.
* **Role:** It’s the class or system that the client code cannot directly use due to interface mismatches.

**3. Adapter**

* **Description:** A class that implements the target interface and internally uses an instance of the adaptee to make it compatible with the target interface.
* **Role:** It acts as a bridge, adapting the interface of the adaptee to match the target interface.

**4. Client**

* **Description:** The code that uses the target interface to interact with objects. It remains unaware of the specific implementation details of the adaptee and the adapter.
* **Role:** It’s the code that benefits from the integration of the adaptee into the system through the adapter.

**Why do we need Adapter Design Pattern?**

1. **Integration of Existing Code:**
   * **Scenario:** When you have existing code or components with interfaces that are incompatible with the interfaces expected by new code or systems.
   * **Need:** The Adapter pattern allows you to integrate existing components seamlessly into new systems without modifying their original code.
2. **Reuse of Existing Functionality:**
   * **Scenario:** When you want to reuse classes or components that provide valuable functionality but don’t conform to the desired interface.
   * **Need:** The Adapter pattern enables you to reuse existing code by creating an adapter that makes it compatible with the interfaces expected by new code.
3. **Interoperability:**
   * **Scenario:** When you need to make different systems or components work together, especially when they have different interfaces.
   * **Need:** The Adapter pattern acts as a bridge, allowing systems with incompatible interfaces to collaborate effectively.
4. **Client-Server Communication:**
   * **Scenario:** When building client-server applications, and the client expects a specific interface while the server provides a different one.
   * **Need:** Adapters help in translating requests and responses between client and server, ensuring smooth communication despite interface differences.
5. **Third-Party Library Integration:**
   * **Scenario:** When incorporating third-party libraries or APIs into a project, and their interfaces do not match the rest of the system.
   * **Need:** Adapters make it possible to use external components by providing a compatible interface for the rest of the application.

**When not to use Adapter Design Pattern?**

1. **When Interfaces Are Stable:**
   * **Scenario:** If the interfaces of the existing system and the new system are stable and not expected to change frequently.
   * **Reason:** Adapters are most beneficial when dealing with evolving or incompatible interfaces. If the interfaces are stable, the overhead of maintaining adapters might outweigh the benefits.
2. **When Direct Modification Is Feasible:**
   * **Scenario:** If you have control over the source code of the existing system, and it’s feasible to directly modify its interface to match the target interface.
   * **Reason:** If you can modify the existing code, direct adaptation of interfaces might be a simpler and more straightforward solution than introducing adapters.
3. **When Performance is Critical:**
   * **Scenario:** In performance-critical applications where the overhead introduced by the Adapter pattern is not acceptable.
   * **Reason:** Adapters may introduce a level of indirection and abstraction, which could have a minor impact on performance. In situations where every bit of performance matters, the Adapter pattern might not be the best choice.
4. **When Multiple Adapters Are Required:**
   * **Scenario:** If a system requires numerous adapters for various components, and the complexity of managing these adapters becomes overwhelming.
   * **Reason:** Managing a large number of adapters might lead to increased complexity and maintenance challenges. In such cases, reconsider the overall design or explore alternatives.
5. **When Adapters Introduce Ambiguity:**
   * **Scenario:** When introducing adapters leads to ambiguity or confusion in the overall system architecture.
   * **Reason:** If the presence of adapters makes the system design less clear or harder to understand, it may be worthwhile to explore alternative solutions that offer a clearer design.



