1. The **Adapter Design Pattern** is a **structural design pattern** that helps two incompatible interfaces work together. Think it like a translator or bridge between two systems.  
   For example:   
   Imagine we have a **mobile phone charger** from the US (two flat pins), but your power socket is **Indian standard** (three round pins).

* Instead of replacing our charger, we buy a **plug adapter**.
* The adapter converts the socket interface to the one our charger can understand.

1. Participants in Adapter Pattern

* **Adaptee** → The existing class with a different interface.
* **Adapter** → The "bridge" class that makes Adaptee compatible with Target.
* **Client** → The code that needs to use Target.
* **Target** → The interface the client expects.

1. When to Use Adapter Pattern

* We want to make legacy code work with new systems.
* Integrating with third-party libraries.
* You want to use an existing class but its interface doesn’t match what you need.

1. Adaptor is a wrapper (middle layer) that **takes the Adaptee as input** (via composition or inheritance) and **produces the Target’s interface as output** so the client can use it without knowing about the Adaptee.  
   So, adaptor is a kind of wrapper class but not all wrapper class is an adaptor.
2. Adaptor Code Example:  
   Suppose we have an **old library** that works with **XML**, but your **new client expects JSON**.  
   Here we have an **interface mismatch**A screenshot of a computer

   AI-generated content may be incorrect.A screen shot of a computer program

   AI-generated content may be incorrect.  
     
     
   A computer screen with white text

   AI-generated content may be incorrect.
3. Side Effects / Drawbacks of Adapter Pattern  
   A. Limited Access to Adaptee Features:

* The Adapter only exposes what Target defines.
* Some advanced Adaptee features may be lost (not mapped).
* If the client later needs those → you may end up expanding the Target, which defeats the simplicity goal.

B. Maintenance / Coupling Risk (**Most Important**)

* If the **Adaptee changes its API**, the Adapter must be updated → you become responsible for keeping adapters in sync.
* If the **Target evolves**, adapters must also evolve to keep compatibility.

C. Potential for Misuse

* Developers sometimes use Adapter as a “quick fix” for mismatched interfaces instead of properly refactoring or designing a consistent abstraction.
* This can create a **patchwork of adapters** that hide technical debt.

1. Adaptor pattern is best when we have a **legacy system** or a **third-party library** that is *relatively stable*. Or just need to translate interfaces so your client code stays clean.