1. **Bridge Pattern** = separating **abstraction (high-level logic)** from **implementation (low-level details)** so both can change **independently**.  
      
   Here ***Abstraction*** is a **high-level control layer** for some entity. This layer isn’t supposed to do any real work on its own. It should delegate the work to the ***implementation*** layer (also called ***platform***). *It* ***defines the logic* or *interface visible to the client.***  
     
   Note that by the term Abstraction here is not mean to say the interfaces or abstract classes in the programming language.  
     
   When talking about **real applications**, the **abstraction layer** can be represented by a **graphical user interface (GUI)**, and the **implementation** could be the underlying **operating system code (API)** which the GUI layer calls in response to user interactions.  
     
   Generally speaking, we can extend such an app in two independent directions:

* Have several different GUIs (for instance, tailored for regular customers or admins).
* Support several different APIs (for example, to be able to launch the app under Windows, Linux, and macOS).

The **abstraction** object controls the appearance of the app, **delegating** the **actual** **work** to the linked **implementation** object. 

1. If DIP already helps me decouple things, why do I need the **Bridge Pattern**?  
    **A. Dependency Inversion — the *principle***The **Dependency Inversion Principle (DIP)** is one of the **SOLID** principles — a **guideline** or **philosophy** of how your code *should be designed*. It says “what to do”  
     
   **B**. **Bridge Pattern**The **Bridge Pattern** is a **specific design pattern** (a structured implementation technique) that *uses* dependency inversion to solve a **particular type of problem** that is when we have **two parallel hierarchies.** It says “how to do it”.
2. **The Bridge pattern is what prevents subclass explosion**  
   by letting two *independent hierarchies* (For ex: Remotes and Devices) evolve separately, and **connect dynamically via composition**.
3. **For Example:** Remote and Device (TV, Radio, Projector etc.)  
     
   We separate:
   * The **Abstraction** → Remote (what the client uses)
   * The **Implementation** → Device (what does the actual work)

So, instead of inheritance between every combination,  
we use **composition** to “bridge” them dynamically.  
  
A screenshot of a computer program

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

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

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

AI-generated content may be incorrect.  
  
Here we decoupled both Remote and Device so that **both hierarchies can evolve independently**.

1. **Without Bridge Pattern (Using only inheritance approach)  
     
   A white background with a black and white flag

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

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

   AI-generated content may be incorrect.**This leads to the **subclass explosion** and results into the cartesian product scenario that is for 2 remote and 5 devices, we should need to have 2 \* 5 = 10 classes and in future if mote remote or devices added, then there is an exponential increase in subclasses which leads to the **subclass explosion problem.**  
     
   Why this happens because **we combined two dimensions of change (Remote Type + Device Type) in a single inheritance hierarchy**. That is Each class now mixes *Remote logic* **and** *Device logic* — they’re no longer independent.

We can’t change one without touching the other. Which leads **to the *combining two dimensions of change into one hierarchy scenario****.*A screenshot of a computer

AI-generated content may be incorrect.

1. The only trade off of the bridge pattern is the ” **Tight Coupling via Wrong Abstraction Level”  
     
   A white background with black text

   AI-generated content may be incorrect.**Now, what happens when a Radio (which doesn’t have channels) implements it?  
   We’re forced to provide dummy methods or throw exceptions.

We created our bridge abstraction at the wrong granularity.  
  
**Solution:** Use **Interface segregation Principle (ISP)** or use **capability composition.**