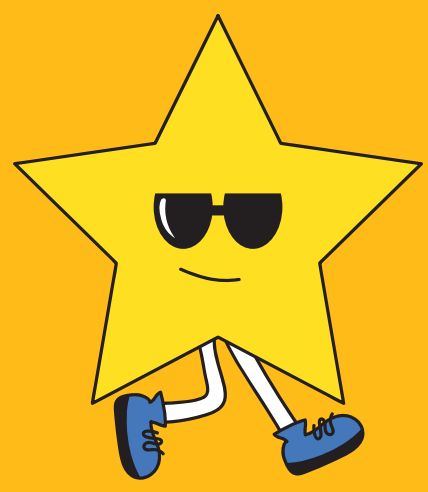
Bridge Pattern

All things you should know about Bridge.

What?

It belongs to the structural category.
Decouples an abstraction from its implementation so that the two can vary independently.

Shirin Monzavi



Real-World Scenario

When an abstraction can have multiple implementations, the typical approach is inheritance. But this leads to some issues.

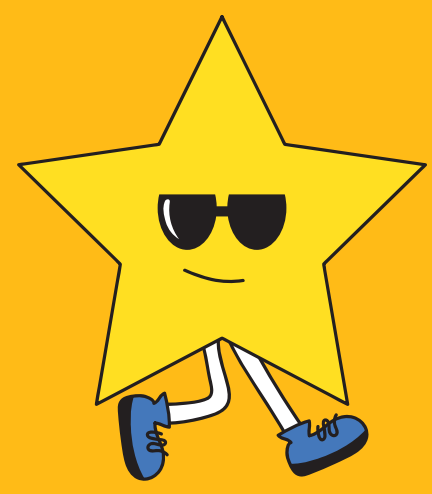
Let’s say we have an abstract class Window, with two implementations: XWindow and PMWindow. If we use inheritance and want to add a new type like IconWindow, we’ll need to create XIconWindow and PMIconWindow.

This leads to:

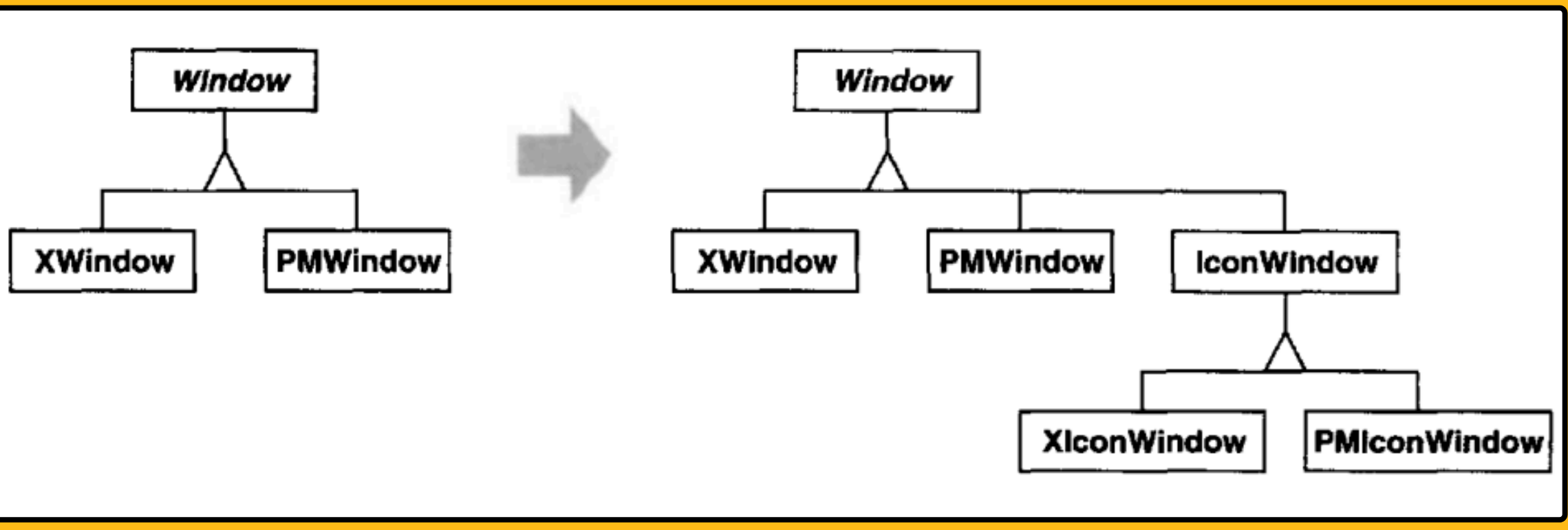
1️⃣ Class explosion – For every new window type, we must implement it for each platform.

2️⃣ Tight coupling – The client is forced to choose a concrete platform (XWindow or PMWindow), tying the code to a specific implementation.

Shirin Monzavi



Problem With UML



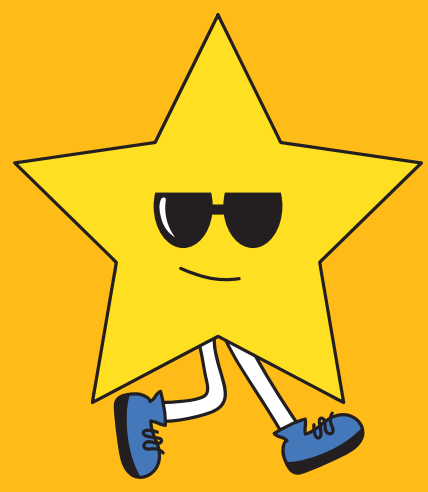
Shirin Monzavi

Solution

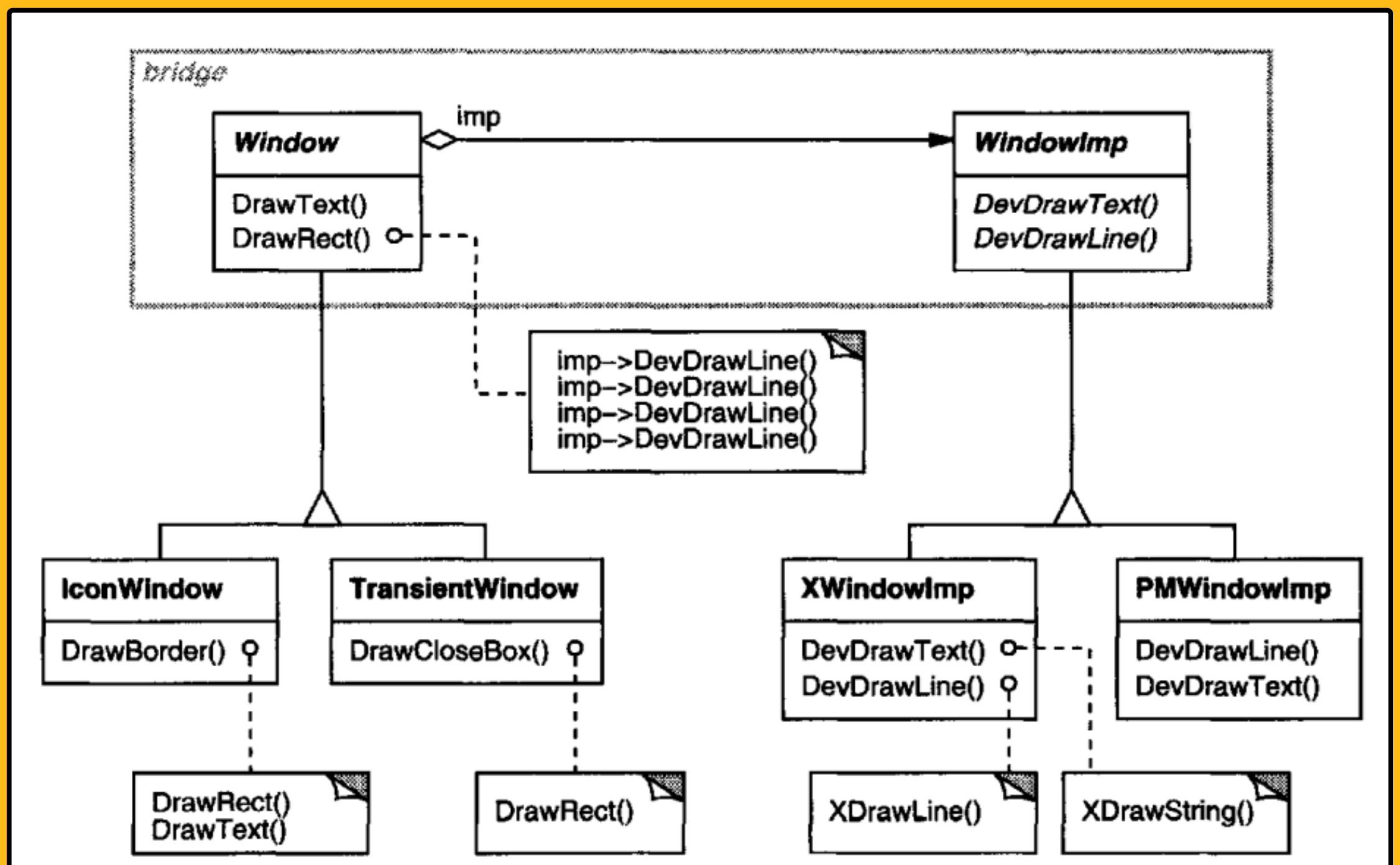
The Bridge pattern solves this by splitting the abstraction (Window, IconWindow, TransientWindow, etc.) from the implementation (WindowImp, XWindowImp, PMWindowImp, etc.).

💡 The abstraction holds a reference to the implementation, and they communicate through a defined interface. This bridges the two hierarchies, enabling independent evolution and better flexibility.

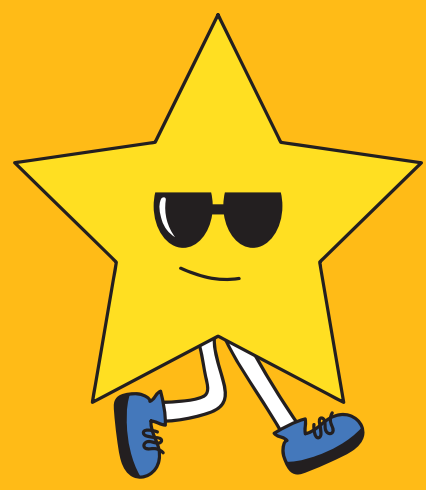
Shirin Monzavi



Structure



Shirin Monzavi



Usability

1- When the implementation must be selected or switched at run-time.

2- Both abstractions and their implementation should be extended by subclassing.

3- Changes in the implementation should not affect the client code.

4- You have a large number of classes as in the real-world scenario. Such a class hierarchy indicates the need for separating an object into two parts.

5- Want to share an implementation among multiple objects(perhaps using reference counting), and this fact should be hidden from the client.