**Bridge（桥接）**

意图

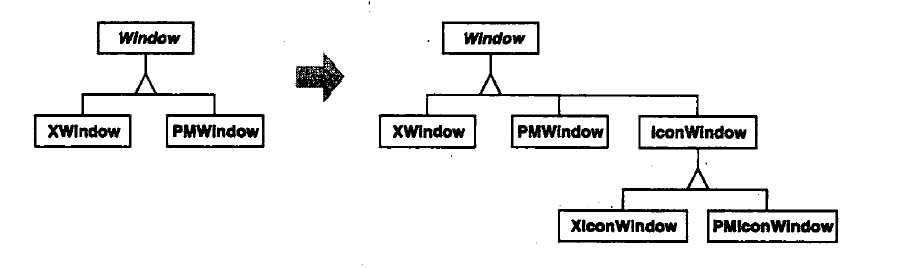
将抽象部分与它的实现部分分离，使它们都可以独立地变化。

动机

当一个抽象可能有多个实现时，通常用继承来协调它们。抽象类定义对该抽象的接口，而具体的子类则用不同方式加以实现。但是此方法有时不够灵活。继承机制将抽象部分与它的实现部分固定在一起，使得难以对抽象部分和实现部分独立地进行修改、扩充和重用。

让我们考虑在一个用户界面工具箱中，一个可移植的Window抽象部分的实现。例如，这一抽象部分应该允许用户开发一些在X Window System和IBM的Presentation Manger系统中都可以使用的应用程序。运用继承机制，我们可以定义Window抽象类和它的两个子类XWindow与PMWindow，由于它们分别实现不同系统平台上的Window界面，但是继承机制有两个不足之处：

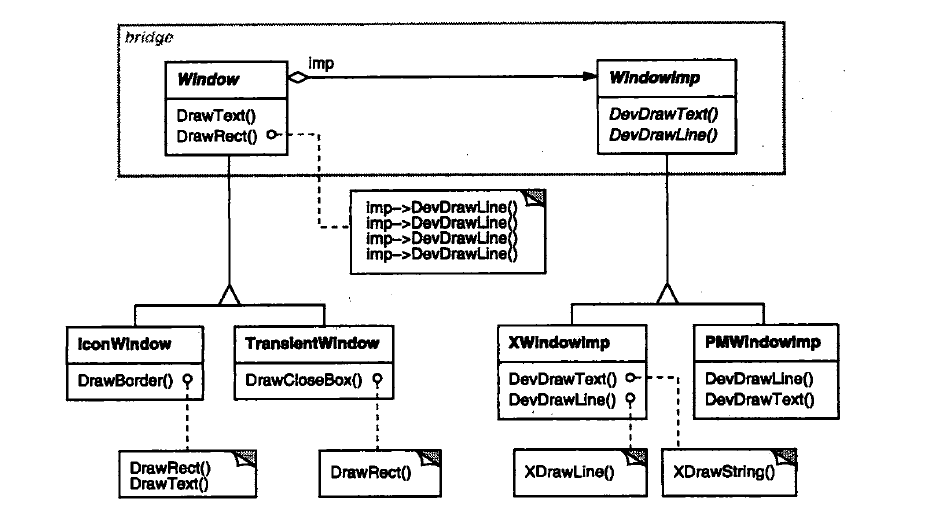
1，扩展Window抽象使之适用于不同种类的窗口或新的系统平台很不方便。假设有Window的一个子类IconWindow，它专门将Window抽象用于图标处理。为了使IconWindow支持两个平台，我们必须实现两个新类XIconWindow和PMIconWindow，更为糟糕的是，我们不得不为每一种类型的窗口都定义两个类。而为了支持第三个系统平台我们还必须为每一种窗口定义新的Window子类，如下图所示：



2，继承机制使得客户代码与平台无关。每当客户创建一个窗口时，必须要实例化一个具体的类，这个类有特定的实现部分。例如，创建XWindow对象会将Window抽象与XWindow的实现部分绑定起来，这使得客户程序依赖于X Window的实现部分。这将使得很难将客户代码移植到其他平台上去。

客户在创建窗口时应该不涉及到其具体实现部分。仅仅是窗口的实现部分依赖于应用运行的平台。这样客户代码在创建窗口时就不应涉及到特定的平台。

Bridge模式解决以上问题的方法是，将Window抽象和它的实现部分分别放在独立的类层次结构中。其中一个类层次结构针对窗口接口（Window，IconWindow，TransientWindow），另外一个独立类层次结构针对平台相关的窗口实现部分，这个类层次结构的根类为WindowImp。例如XWindowImp子类提供了一个基于X Window系统的实现，如下图所示。



对Window子类的所有操作都是用WIndowImp接口中的抽象操作实现的。这就将窗口的抽象与系统平台相关的实现部分分离开来。因此，我们将Window与WindowImp之间的关系称之为桥接，因为他在抽象类与它的实现之间起到了桥梁作用，使它们可以独立地变化。

结构图

