Design Patterns

MSc in Computer Science

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Bridge

Structural Pattern

Bridge Summary

- To avoid a permanent binding between an abstraction and its implementation.
- Particularly when the implementation may be selected or switched at run-time.
- Both the abstractions and their implementations should be extensible independently
- Changes in the implementation of an abstraction should have no impact on clients

Bridge Intent

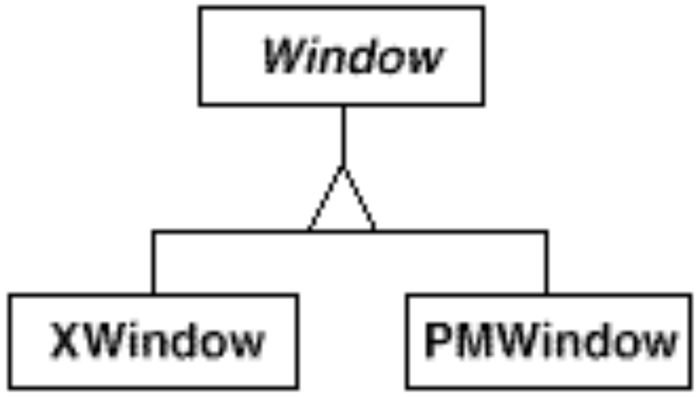
- Intent
    - Decouple an abstraction from its implementation so that the two can vary independently. Often used to achieve platform independence.
- Application-specific code on one side of the bridge (the “business logic”) uses platform-dependant code on the other side through a well defined interface.
    - Reimplement that interface and the “business” logic doesn’t know or care.
    - Reimplement the business logic and the platform-specific interface implementations don’t care.
- Examples of Bridge in Java are Swing, JFace, SWT, JDBC.

Motivation (1)

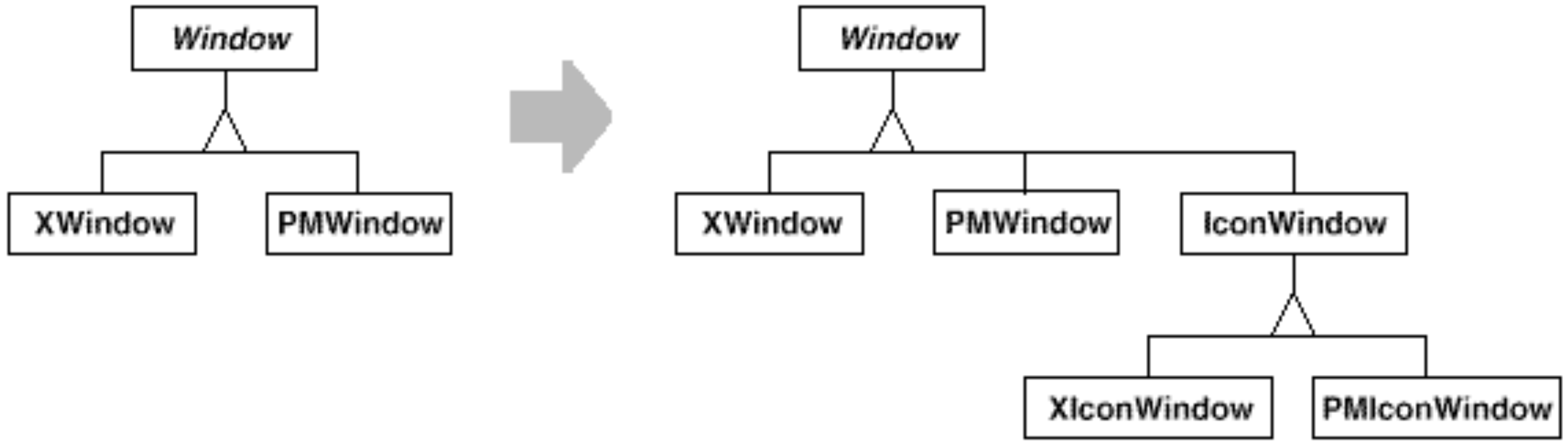
- When an abstraction can have one of several possible implementations, a common way to accommodate them is to use inheritance.
- An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.
- This approach isn't always flexible enough:
  - Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.

Example

- Consider the implementation of a portable Window abstraction in a user interface toolkit.
- This abstraction should enable us to write applications that work on different windowing systems (PM & X for example)
- Using inheritance, we could define an abstract class Window and subclasses XWindow and PMWindow that implement the Window interface for the different platforms.



Example

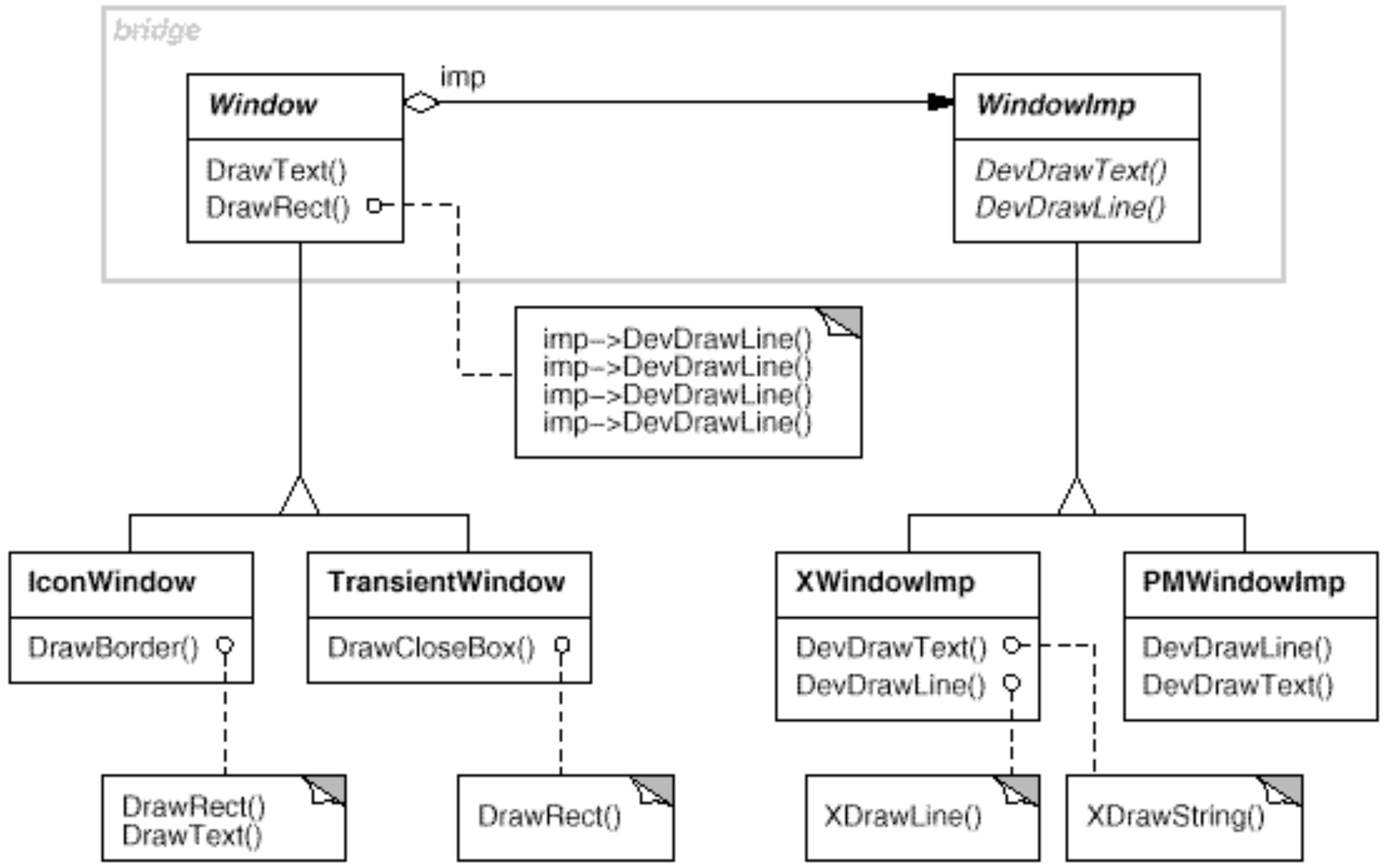


- Imagine an IconWindow subclass of Window that specializes the Window abstraction for icons.
- To support IconWindows for both platforms, we have to implement two new classes, XIconWindow and PMIconWindow.
- Worse, we'll have to define two classes for every kind of window. Supporting a third platform requires yet another new Window subclass for every kind of window.
- It is highly inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms.

Example

- It makes client code platform-dependent, because whenever a client creates a window, it instantiates a concrete class that has a specific implementation.
    - For example, creating an XWindow object binds the Window abstraction to the X Window implementation, which makes the client code dependent on the X Window implementation. This, in turn, makes it harder to port the client code to other platforms.
- Clients should be able to create a window without committing to a concrete implementation.
- Only the window implementation should depend on the platform on which the application runs. Therefore client code should instantiate windows without mentioning specific platforms.

Bridge Implementation



Bridge Pattern

- The Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies.
- There is one class hierarchy for window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific window implementations, with WindowImp as its root.
- The XWindowImp subclass, for example, provides an implementation based on the X Window System.
- All operations on Window subclasses are implemented in terms of abstract operations from the WindowImp interface.
- This decouples the window abstractions from the various platform-specific implementations. We refer to the relationship between Window and WindowImp as a bridge, because it bridges the abstraction and its implementation, letting them vary independently.

Applicability

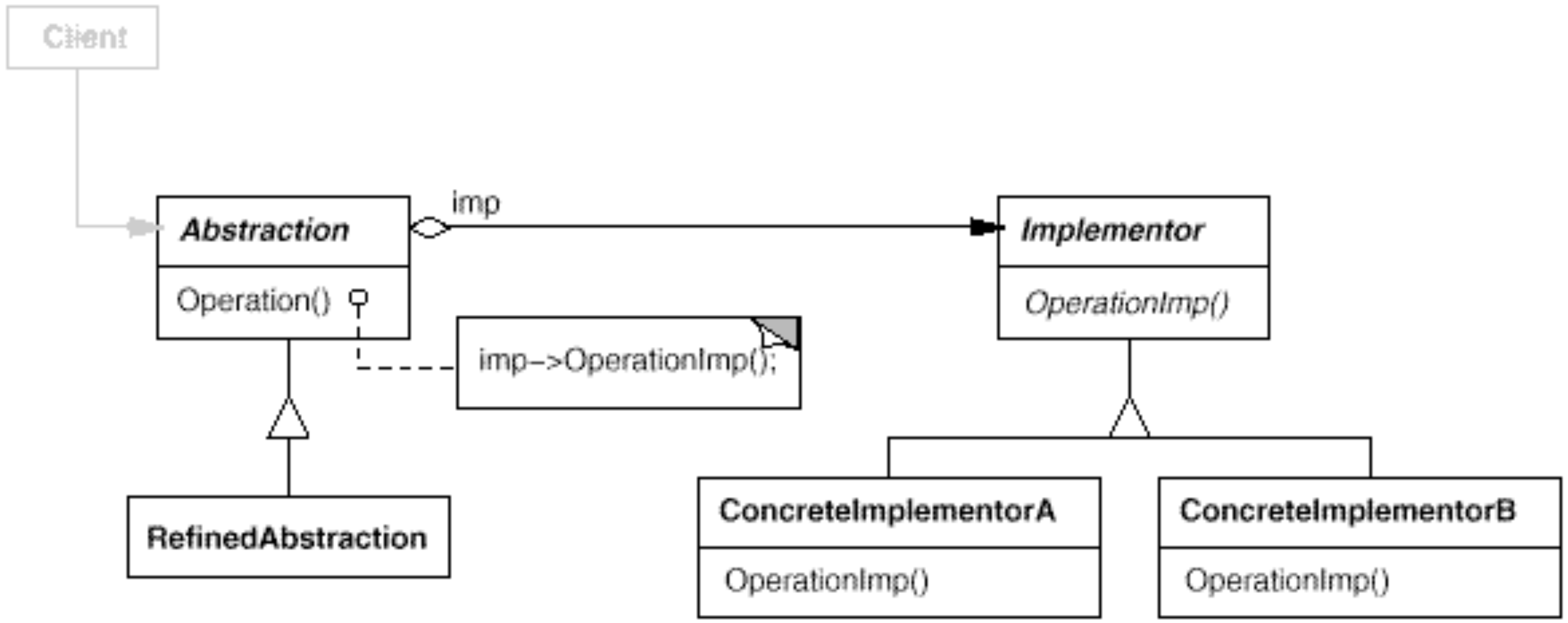
- To avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.

- Both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently

- Changes in the implementation of an abstraction should have no impact on clients

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

Structure


Client
Abstraction
Operation()
imp
imp->OperationImp();
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

Participants

- Abstraction : defines the abstraction's interface and maintains a reference to an object of type Implementor.
- RefinedAbstraction : Extends the interface defined by Abstraction.
- Implementor: defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based
- ConcreteImplementor: implements the Implementor interface and defines its concrete implementation.

Collaborations

- Abstraction (Window)
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- RefinedAbstraction (IconWindow)
  - Extends the interface defined by Abstraction.
- Implementor (WindowImp)
  - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor (XWindowImp, PMWindowImp)
  - Implements the Implementor interface and defines its concrete implementation

Consequences

- Decoupling interface and implementation. An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.
- This decoupling encourages layering that can lead to a better-structured system. The high-level part of a system only has to know about Abstraction and Implementor.
- Improved extensibility. You can extend the Abstraction and Implementor hierarchies independently.
- Hiding implementation details from clients. You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit