Bridge Pattern

Jim Fawcett

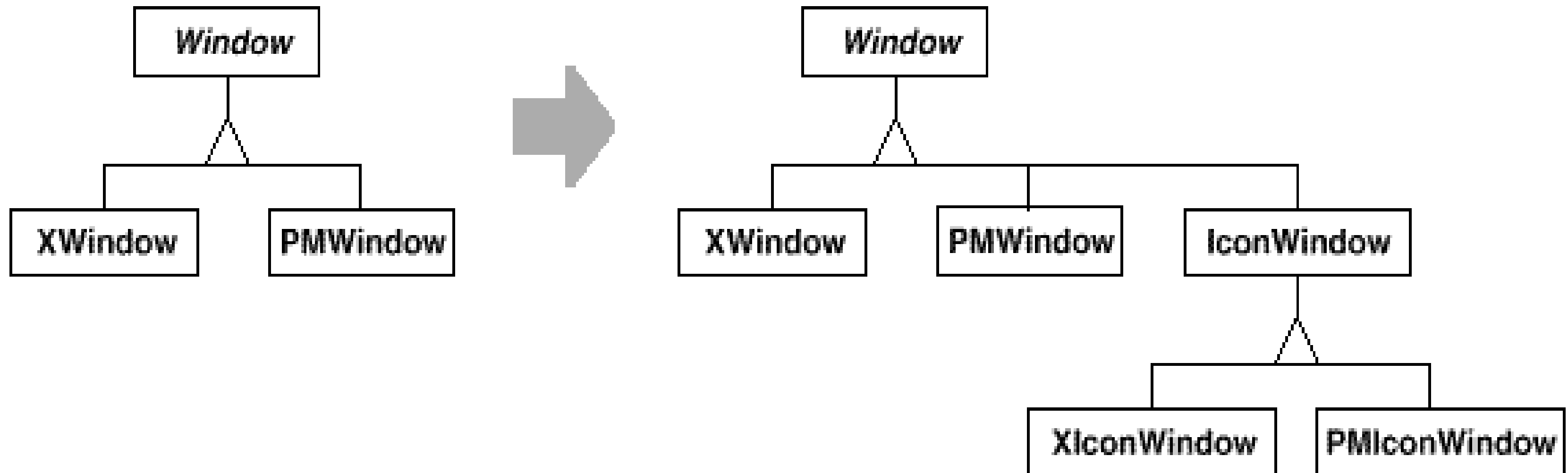
CSE776 – Design Patterns

Fall 2011

Intent

- "Decouple an abstraction from its implementation so that the two can vary independently."
  - Multiple Dependent Implementations.
  - Single Independent Interface.

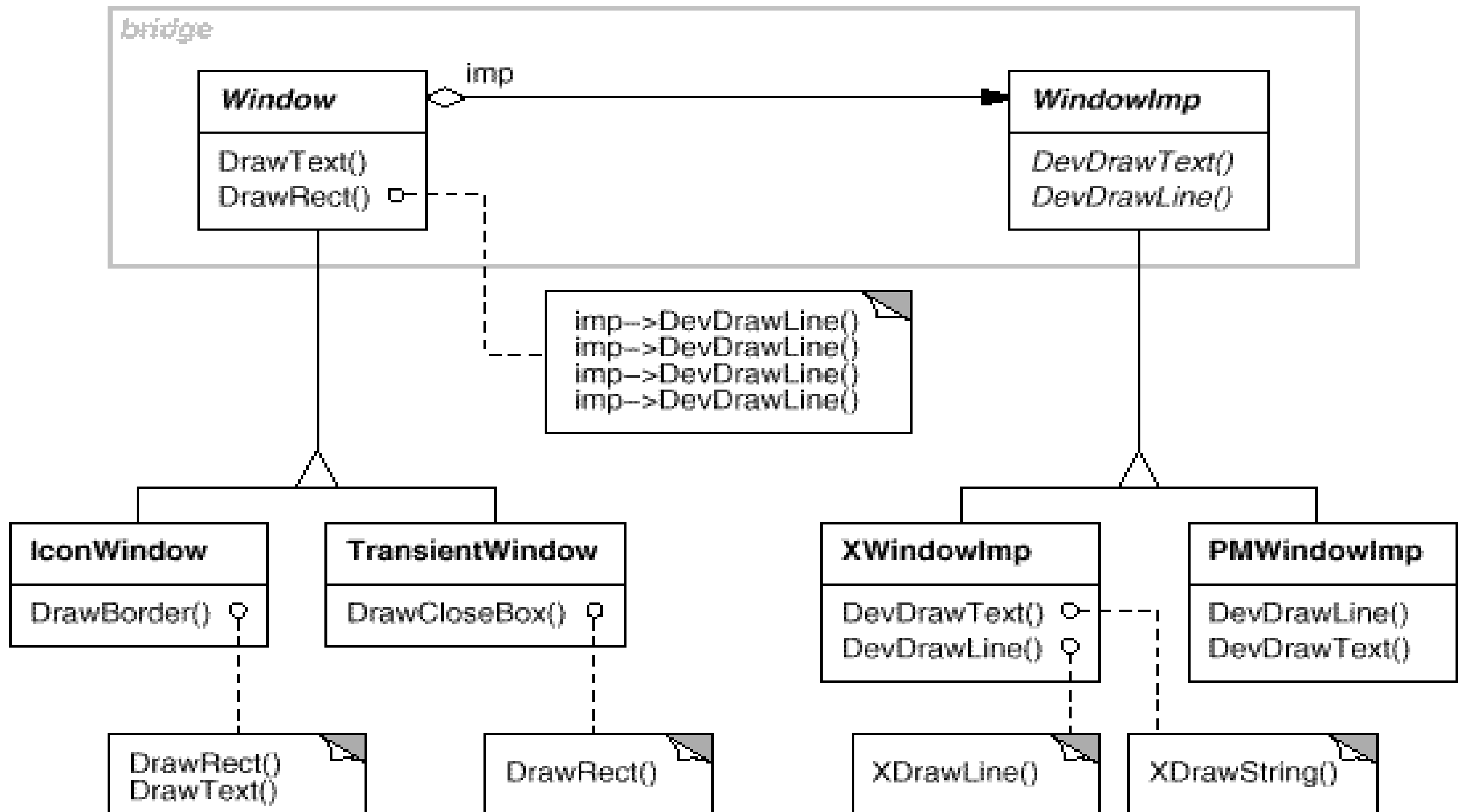
Motivation



Motivation

- When an abstraction can have several implementations inheritance is used to accommodate them.
- But inheritance binds an implementation to the abstraction permanently, hence its difficult to modify, extend and reuse abstraction and implementations ***independently***.
- It's inconvenient to extend the abstraction to cover different kinds of windows or new platforms (window abstraction example in the text).
- Inheritance without a **Bridge** makes client code platform dependent.

Motivation



Motivation

- Bridge Pattern addresses these problems:
  - Puts the Window Abstraction and its implementation in separate class hierarchies.
  - One class hierarchy for window interfaces and a separate hierarchy for platform specific window implementation with WindowImp as its root.
  - All operations on Window subclasses are implemented in terms of abstract operations from WindowImp interface. Decouples the window abstraction from the various platform specific implementations.
  - We refer to the relationship between Window and WindowImp as a bridge.

Forces

- We want to avoid binding clients to an implementation
- Separating abstraction from implementation adds complexity
- Well suited to cross-platform development
- Easy to provide stubs for early development without breaking clients when real code is inserted

Applicability

Use the bridge Pattern when:

- You want to avoid a permanent binding between an abstraction and its implementation. Implementation may be selected or switched at run time.
- Both the abstraction and their implementation should be extensible by subclassing.
- Changes in the implementation of an abstraction should have no impact on the clients (that is their code should not be recompiled).

Structure

Client

Abstraction

Operation()

imp

Implementor

OperationImp()

imp->OperationImp()

RefinedAbstraction

ConcreteImplementorA

OperationImp()

ConcreteImplementorB

OperationImp()

Participants

- **Abstraction (Window)**
  - defines the abstraction interface
  - maintains a reference to an object of type implementor.
- **Refined Abstraction (Icon Window)**
  - Extends the interface defined by Abstraction (optional) .
- **Implementor (WindowImp)**
  - defines the interface for the implementation classes.
- **ConcreteImplementor (XWindowImp)**
  - implements the implementor interface and defines its concrete implementation.

Collaborators

- " Abstraction forwards client requests to its Implementor object."
  - Client interface with the abstraction class.
  - Abstraction class uses the implementor class interface to make use of the specific concrete class interface.

Consequences

- Decoupling interface and implementation.
  - An implementation is not bound permanently to the interface. The implementation of an abstraction can be configured at run time.
- Improved Extensibility
  - You can extend the Abstraction and Implementation hierarchies independently.
- Hiding Implementation details from the client.
  - You can shield clients from implementation details like the sharing of implementor objects.

Implementation

- Only One Implementor (Authors' advice)
  - Start with single abstraction and implementation, but allow for Additional Implementations.
- Creating the right Implementor object.
  - How, when and where to chose which implementor ?
  - Can be instantiated by parameter passed to constructor.
  - Chose default implementation when constructed and change later, based on usage.
  - Delegate the decision to another object (Abstract factory).

Implementation (continued)

- Sharing Implementors
  - How to share implementations among several objects?
  - Can use the Handle/Body Idiom. Clients share a reference counted implementation.

Unique Point-of-View

- Bridge allows you to decouple an implementation so that it is not bound to an abstraction
  - A party guest can wear several masks
  - Abstraction is changed at run-time
  - Different user interfaces for normal operation and critical operations.
- Abstraction is not bound to a specific implementation
  - One mask can be worn by several party guests
  - Implementation is changed at run-time
  - Fault-tolerant system reconfigures, but preserves the same user interface, under partial failure

Windows is a Bridge

- The Bridge Pattern allows a designer to provide a simple interface in the abstraction, while providing a powerful, but complex interface for the implementation.
- That is essentially what windows does:
  - Win32API is the abstraction's interface
  - Kernel language is the implementation's interface

The .Net Run-time is a Bridge

- C#, Visual Basic, Managed C++ are all abstractions
- MSIL is the implementation
- Mono and dotGnu are other implementations

Known Uses

- "Design Patterns" authors cite the example:
  - Windows example (from ET++).
    - WindowImp is called WindowPort and has subclasses such as XWindowPort and SunWindowPort.
    - Window Object creates its corresponding Implementor by requesting it from an abstract factory called Window System.
    - Window/WindowPort design extends the Bridge Pattern in that WindowPort also keeps a reference back to the window.

Related Patterns

- Abstract Factory
  - Can create and configure a particular bridge
- Adapter Pattern
  - geared towards making unrelated classes work together.