Bridge

An object structural pattern

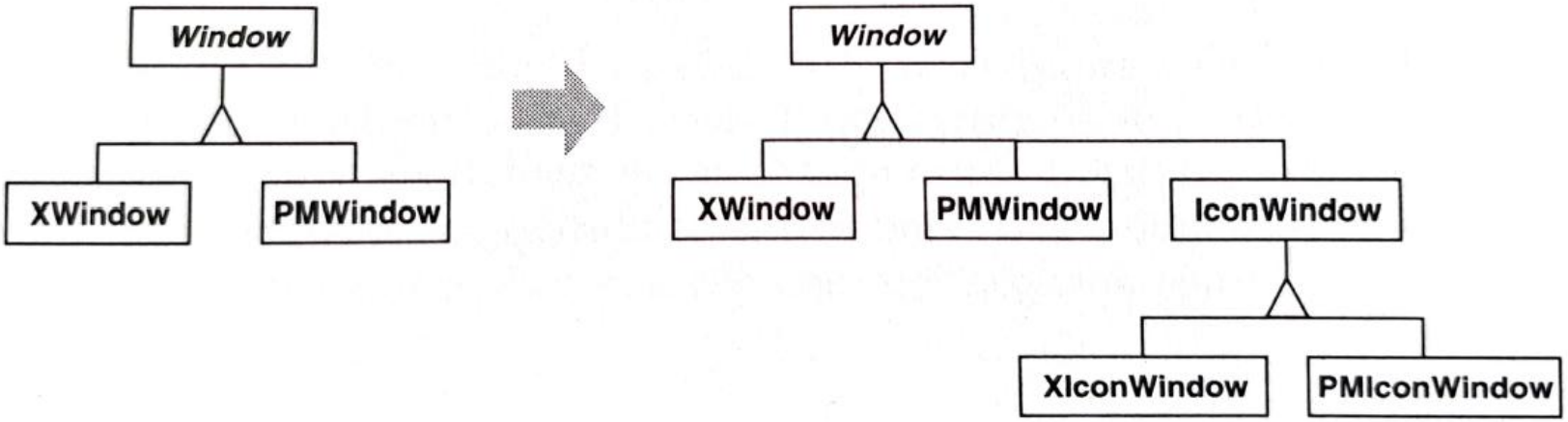
Learning goals

1. Learn the idea, structure, and Java implementation of the Bridge design pattern.
2. Learn to apply the Bridge DP in your own programming.

Idea of Bridge

- The Bridge design pattern separates an **abstraction** from its **implementation**.
  - Example of an abstraction: a Window
  - Example of an implementation: a JavaFX (or Swing) Window
- Promotes flexibility and extensibility in a software's architecture.
- It helps in avoiding monolithic designs that are difficult to modify and extend.
- This enables changing implementation details without altering the abstraction.
  - Clients do not see the details of underlying implementations.
  - Abstractions can evolve at their own pace based on high-level requirements.

Example: Windowing environments



- There are two windowing platforms: XWindow and PMWindow.
- This is a monolithic approach, where the platform-specific code is intertwined in the object structure.
- Bad expandability: what happens if a third platform is introduced?

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 151

Example: Windowing environments

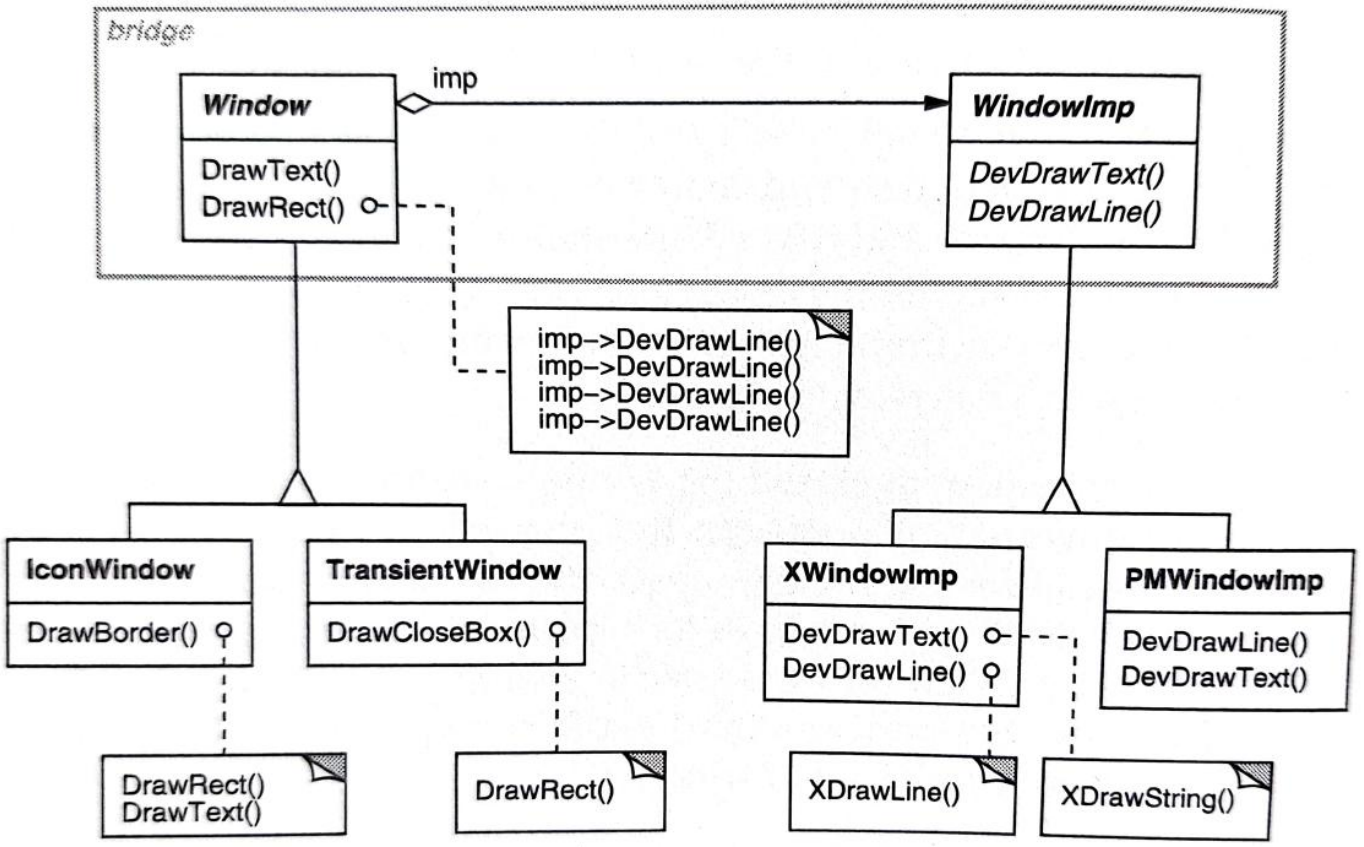


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 152

Example: Windowing environments

- Note the two **independent hierarchies**:
  - The abstraction hierarchy on the left
  - The implementation hierarchy on the right
- The Client communicates with the abstraction, i.e. the Window interface and its abstract operations.
  - The implementation is not visible to the client.
- Both the abstraction and implementation hierarchies can be **developed independently**.
- The relationship between the abstraction and implementation interfaces is called the **bridge**.

General structure

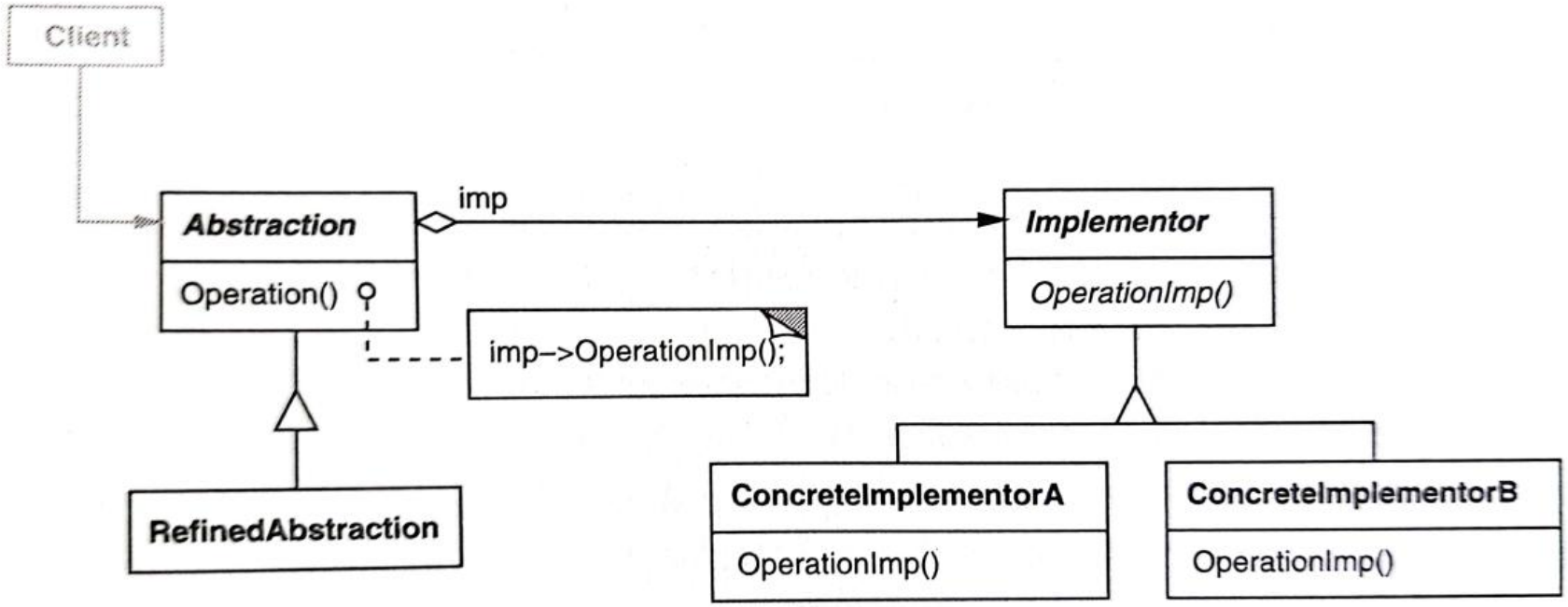


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 153

Roles

- **Abstraction**: Defines the abstraction's interface. Maintains a reference to an object of the Implementor type.
- **Refined Abstraction**: Extends the interface defined by Abstraction. Implements the abstracted part of the code which can vary independently from the implementor.
- **Implementor**: Defines the interface for implementation classes.
  - No need to be an exact replica of the Abstraction interface.
  - Often provides simple primitives whereas Abstraction provides higher-level operations.
- **Concrete Implementor**: Implements the Implementor interface and defines its concrete implementation.

Bridge vs. Adapter

- Both Bridge and Adapter promote interoperability between two classes.

	Bridge	Adapter
Point of view	Proactive: anticipate variation in implementation	Reactive: find a way to coexist with incompatible code
Abstraction	Abstraction acts as a middle layer	No middle layer: Target interface methods are directly mapped to Adaptee
Development	Allows independent development of both sides	One side of the adapter must match an existing interface

Practical issues

- The Bridge pattern is about extracting independent dimensions into their own hierarchies.
  - The dimension can also be something else than a platform.
- Use inheritance to extend parts of the abstraction when necessary.
  - Each extended abstraction can correspond to different kinds of high-level operations.
- Consider the possibility of switching implementors dynamically.
- The Abstraction can be implemented as an interface or an abstract class.
- Bridge can add an additional layer of indirection which may slightly impact performance.