# Il pattern Bridge

a cura di **Angelo Furfaro**
da “Design Patterns”, Gamma et al.

Dipartimento di
Ingegneria Informatica, Elettronica, Modellistica e Sistemistica
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@unical.it
Web: http://angelo.furfaro.dimes.unical.it

# Bridge

## Classificazione

- Scopo: strutturale
- Raggio d'azione: basato su oggetti

## Scopo

- Disaccoppia un'astrazione dalla sua implementazione in modo che le due possano variare indipendentemente l'una dall'altra.

## Altri nomi

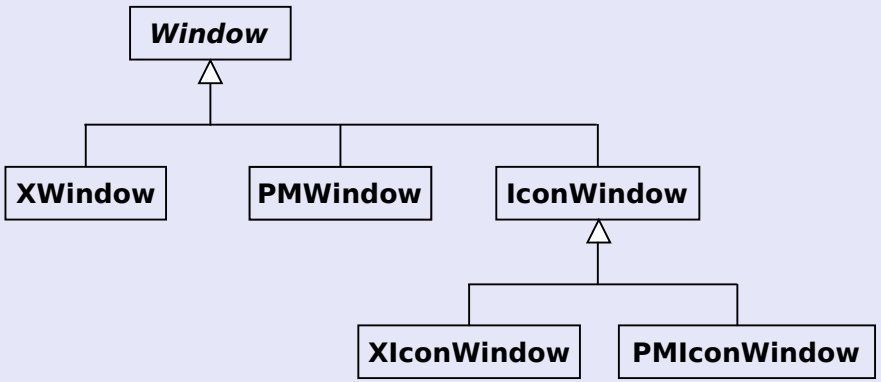
Handle/Body

# Motivazione

- Quando un'astrazione può avere una tra più implementazioni possibili, in genere si risolve il problema ricorrendo all'ereditarietà.
- L'astrazione viene definita da un'interfaccia o da una classe astratta e le sottoclassi concrete la implementano in modi differenti.
- Tale approccio non è flessibile poiché l'ereditarietà *lega* un'implementazione ad un'astrazione in modo *permanente*.
- Ciò rende difficile modificare, estendere e riusare astrazioni ed implementazioni in modo indipendente.

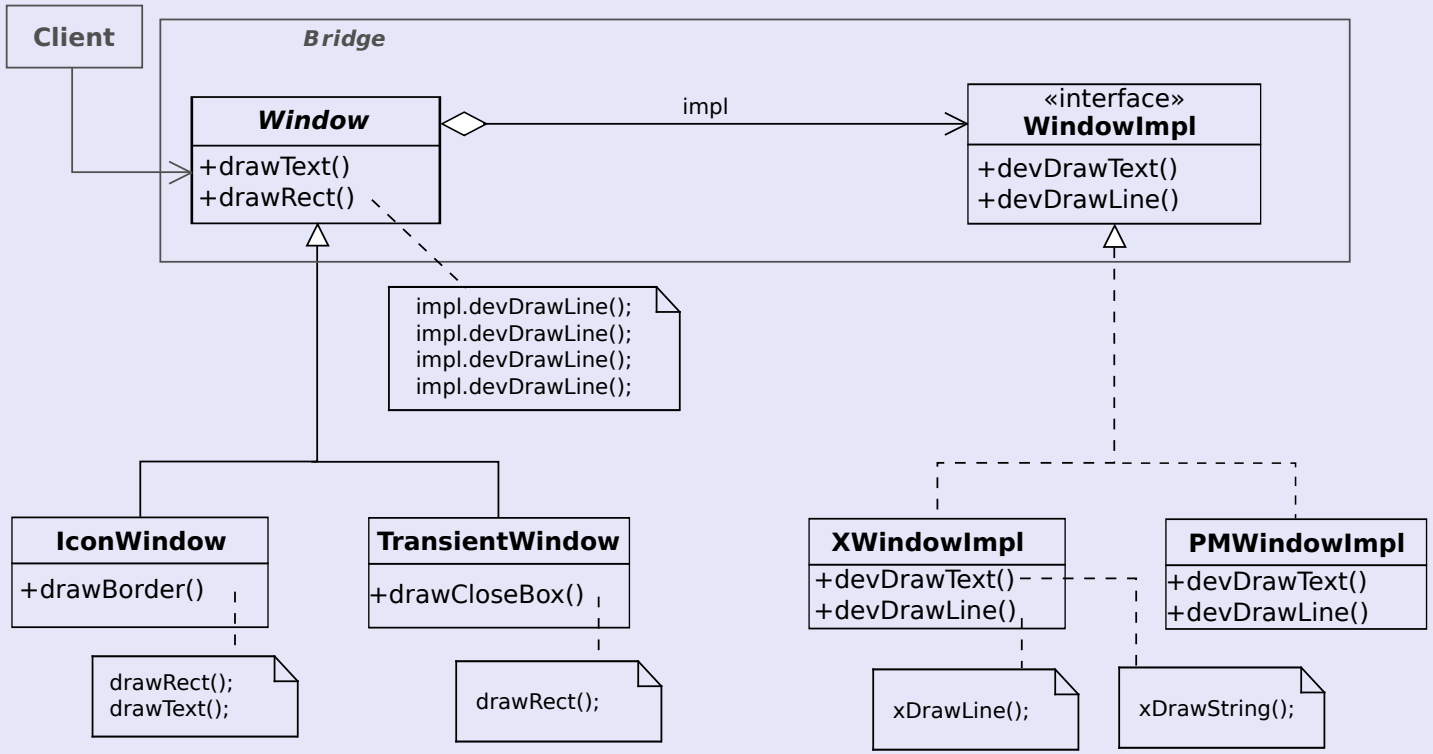
# Esempio

- Si supponga di volere scrivere un toolkit per la realizzazione di interfacce grafiche. Sicuramente ci sarebbe il bisogno di un'astrazione `Window` per rappresentare una finestra. Si vuole far in modo che il toolkit funzioni con diversi gestori grafici come ad esempio X Windows o IBM Presentation Manager.
- Si può fare ricorso all'ereditarietà rendendo `Window` una classe astratta (o un'interfaccia) ed introducendo due sottoclassi `XWindow` e `PMWindow` per fornire due implementazioni dell'astrazione. Tale approccio ha due difetti principali:
  1) È scomodo estendere l'astrazione `Window` per supportare tipologie diverse di finestre o nuove piattaforme. Se ad esempio si volesse introdurre una specializzazione `IconWindow` per le icone, occorrerebbe anche introdurre due sottoclassi `XIconWindow` e `PMIconWindow` per le due piattaforme.




  2) Il codice del client diventa dipendente dalla piattaforma utilizzata. Ogni volta che occorre creare una finestra occorre istanziare una specifica classe concreta.

# Esempio

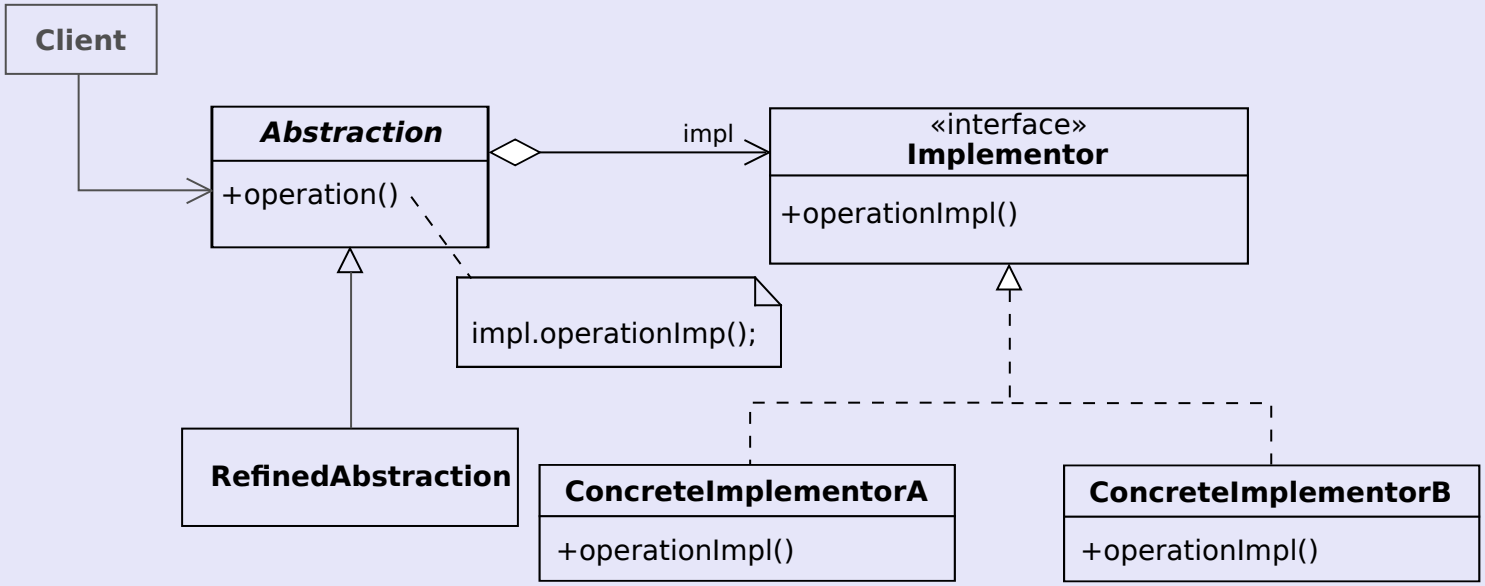



- Il pattern Bridge risolve questi problemi introducendo due gerarchie separate: una per le astrazioni (`Window`,`IconWindow`, `TransientWindow`) ed una (con radice (`WindowImpl`) per le diverse implementazioni dipendenti dalla piattaforma.
- I metodi di `Window` sono tutti implementati in termini dei metodi di `WindowImpl`.
- La relazione tra `Window` e `WindowImpl` è detta *bridge* in quanto funge da ponte tra un'astrazione ed una implementazione consentendo ad entrambe di variare indipendentemente.

# Applicabilità

- Si vuole evitare un legame permanente tra un'astrazione e la sua implementazione, come nel caso in cui l'implementazione deve poter essere selezionata e/o modificata durante l'esecuzione.
- Si vuole avere la possibilità di estendere sia le astrazioni che le implementazioni per mezzo dell'ereditarietà. Tramite Bridge è possibile combinare le astrazioni e le implementazioni in vario modo e di estendere le une indipendentemente dalle altre.
- I cambiamenti nell'implementazione di un'astrazione non devono avere impatto sui client.
- Si vuole condividere una stessa implementazione fra più oggetti nascondendo questa condivisione ai client.

## Struttura




## Partecipanti

- **Abstraction**: specifica l'interfaccia dell'astrazione. Mantiene un riferimento ad un oggetto di tipo `Implementor`.
- **RefinedAbstraction**: Estende l'interfaccia definita da `Abstraction`
- **Implementor**: definisce l'interfaccia per le classi che implementano l'astrazione. Non deve corrispondere esattamente all'interfaccia di `Abstraction`: `Implementor` fornisce le operazioni base, mentre `Abstraction` definisce operazioni di più alto livello implementate sfruttando quelle di base.
- **ConcreteImplementor**: definisce un'implementazione concreta dell'interfaccia `Implementor`.

# Conseguenze

- ☺ *Disaccoppiamento tra interfaccia ed implementazione.*
  Un'implementazione non è più legata in modo permanente a un'interfaccia. L'implementazione di un'astrazione può essere configurata durante l'esecuzione. Il disaccoppiamento tra `Abstraction` ed `Implementor` elimina le dipendenza dall'implementazione a tempo di compilazione. Ciò aiuta ad ottenere una struttura stratificata del sistema.
- ☺ *Maggiore estensibilità.*
  Le gerarchie di classi `Abstract` e `Implementor` possono essere estese indipendentemente.
- ☺ *Mascheramento dei dettagli dell'implementazione ai client.*
  I client non devono preoccuparsi dei dettagli implementativi, quali per esempio la condivisione di oggetti `Implementor` e il relativo meccanismo di conteggio dei riferimenti.

# Pattern Correlati

## Abstract Factory

Abstract Factory può essere utilizzato per creare e configurare un particolare Bridge

## Adapter

- Il pattern Adapter ha lo scopo di far cooperare tra di loro classi non correlate dopo che sono state progettate.
- Il pattern Bridge è utilizzato all'inizio di un progetto per consentire ad astrazioni ed implementazioni di variare in modo indipendente.