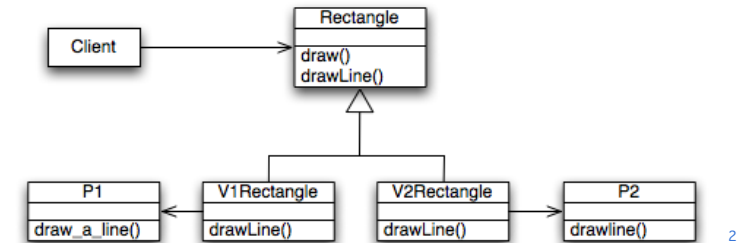


# Bridge

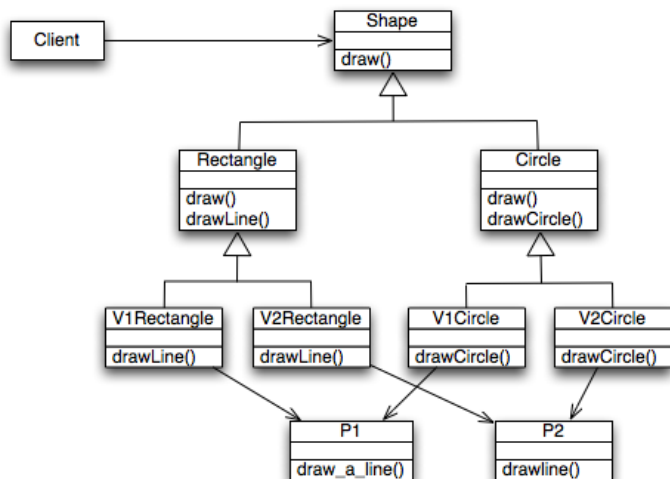
- **Intento**
  - Disaccoppiare una *astrazione* dalla sua *implementazione* così che le due possano variare indipendentemente
- **Motivazione**
  - Quando una astrazione può avere varie implementazioni, di solito si usa l'ereditarietà
    - Una classe astratta definisce l'interfaccia dell'astrazione, le sottoclassi concrete la implementano in modi diversi
    - Questo approccio non è flessibile poiché collega l'astrazione all'implementazione permanentemente

# Bridge

- **Esempio**
  - In un sistema, occorre avere Rettangoli e Cerchi (astrazioni)
  - Occorre che Rettangoli e Cerchi siano disponibili per due tipi di piattaforme P1 e P2 (implementazioni)
  - Per P1 devo usare draw\_a\_line() e draw\_a\_circle()
  - Per P2 devo usare drawline() e drawcircle()
- **Struttura classi iniziale (prima di usare Bridge)**



## Esempio, prima di usare Bridge

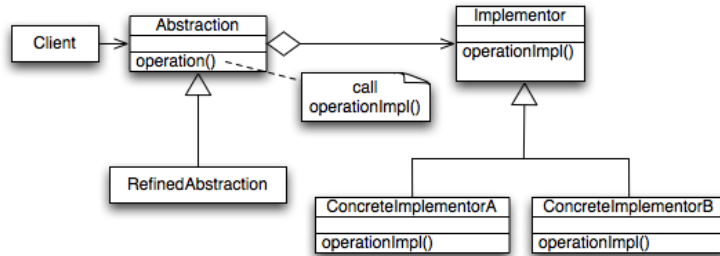


## Considerazioni

- La soluzione appena vista porterebbe ad una proliferazione di classi
  - Se introducessi un'altra piattaforma (implementazione) avrei bisogno di 6 classi concrete (2 shape x 3 piattaforme)
  - Se introducessi un'altra shape (astrazione), avrei bisogno di 9 classi concrete (3 x 3)
- Per ogni variazione da introdurre vorrei invece un incremento lineare del numero di classi
- Inoltre, la classe V1Rectangle è legata alla piattaforma P1 in modo permanente
  - Una istanza di V1Rectangle non può usare una piattaforma diversa da P1

## Soluzione tramite Bridge

- Struttura



- Partecipanti

- Abstraction definisce l'interfaccia per i client e mantiene un riferimento ad un oggetto di tipo Implementor
- RefinedAbstraction estende l'interfaccia definita da Abstraction
- Implementor definisce l'interfaccia per le classi dell'implementazione. Questa interfaccia non deve corrispondere ad Abstraction, di solito Implementor fornisce operazioni primitive, mentre Abstraction definisce operazioni di più alto livello
- ConcreteImplementor implementa l'interfaccia di Implementor e fornisce le operazioni concrete

E. Tramontana - Bridge & Chain - 7 June 10 5

## Bridge (conseguenze)

- Conseguenze

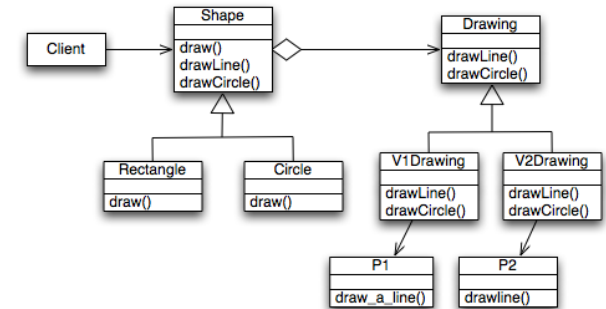
- Bridge permette ad una implementazione di non essere connessa permanentemente ad una interfaccia, l'implementazione può essere configurata ed anche cambiata a runtime
- Il disaccoppiamento permette di cambiare l'implementazione senza dover ricompilare Abstraction ed i Client
- Solo certi strati del software devono conoscere Abstraction e Implementor
- I Client non devono conoscere Implementor
- Le gerarchie di Abstraction e Implementor possono evolvere in modo indipendente

E. Tramontana - Bridge & Chain - 7 June 10 7

## Bridge

- Secondo la soluzione indicata dal design pattern Bridge, per il suddetto esempio si avrà il diagramma delle classi disegnato sotto

- Rectangle.draw() invoca drawLine() della superclasse Shape, quest'ultima invoca drawLine() dell'istanza di una sottoclasse di Drawing
  - Analogamente per Circle
- Una nuova classe, Rombo, sarà implementata come sottoclasse di Shape
  - La classe Rombo invocherà drawLine() presente in Shape



## Chain of Responsibility

- Intento

- Evitare di accoppiare il mandante di una richiesta con il ricevente, dando così la possibilità a più di un oggetto di gestire la richiesta. Concatena gli oggetti riceventi e passa la richiesta tra questi fino a che un oggetto la gestisce

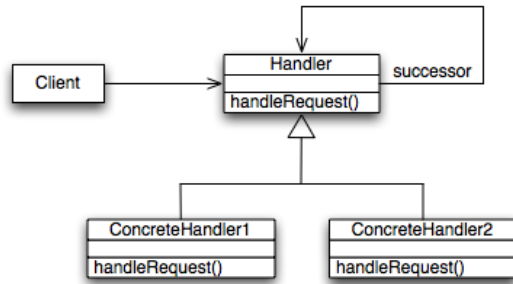
- Motivazione

- Consideriamo una interfaccia utente dove l'utente può richiedere aiuto su parti dell'interfaccia. Ad es. un bottone può fornire informazioni di aiuto. Se non esiste una informazione specifica allora il sistema dovrebbe fornire il messaggio d'aiuto del contesto più vicino (ad es. la finestra)
- Il problema: l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto (es. Button) che inizia la richiesta
- Disaccoppiare mandante e ricevente

E. Tramontana - Bridge & Chain - 7 June 10 8

## Soluzione

- Struttura



- Partecipanti

- Handler definisce l'interfaccia per gestire le richieste
- ConcreteHandler gestisce la richiesta per cui è responsabile, può accedere al suo successore, inoltra la richiesta se non può gestirla
- Client inizia la richiesta ad un oggetto ConcreteHandler
- La richiesta si propaga lungo la catena finché un oggetto ConcreteHandler la può gestire

## Chain of Responsibility

- Conseguenze

- Riduce l'accoppiamento: chi fa una richiesta non conosce il ricevente e viceversa
- Un oggetto della catena non deve conoscere la struttura della catena
- Gli oggetti handler anziché mantenere i riferimenti a tutti i candidati riceventi mantengono solo il riferimento al successore
- Si aggiunge flessibilità nella distribuzione di responsabilità agli oggetti. Si può cambiare o aggiungere responsabilità nella gestione di una richiesta cambiando la catena a runtime
- Non c'è garanzia che una richiesta venga gestita, poiché non c'è un ricevente esplicito, la richiesta potrebbe arrivare alla fine della catena senza essere gestita

## Chain of Responsibility

- Implementazione

- Handler o ConcreteHandler possono definire i link per avere il successore della catena
- Una gerarchia già esistente può essere usata, anziché ridefinire i link (vedi Composite ed il riferimento al parent), se la gerarchia riflette la catena, altrimenti i link vanno definiti
- Tipi di richieste
  - Ciascun tipo di richiesta può corrispondere ad una operazione. In questo caso il set di richieste è definito dall'Handler
  - In alternativa, a ciascun tipo di richiesta corrisponde un codice e solo una operazione è definita. Il set di richieste non è fissato, richiedente e riceventi prendono accordi sulla codifica delle richieste. Più controlli a runtime