OOP - Introduzione

Introduzione alla Programmazione Orientata agli Oggetti

Andrea Colleoni

Lezione 3 – Design pattern

Introduzione ai Design Pattern

Fondamenti e motivazioni...

- Il SW è un'entità complessa: esistono innumerevoli modi di descrivere un problema (modelli) e per ognuno di essi esistono innumerevoli soluzioni
- Quali sono i parametri per cui riteniamo una soluzione accettabile?
  - Risolve il problema per cui è stata implementata
  - Supporta il cambiamento
  - È economica
- Quale è l'oggettività della valutazione dell'accettabilità di una soluzione?
  - Come si può dire che una soluzione è oggettivamente “buona” o “cattiva”?

...Fondamenti e motivazioni

- L'analisi di soluzioni comunemente ritenute “buone” e di quelle comunemente ritenute “cattive” è alla base della riconoscibilità dei pattern: cosa hanno in comune le soluzioni “buone” che quelle “cattive” non hanno?
  - Il GoF (Gamma, Helm, Johnson, Vlissides) ha osservato e documentato tale situazione ed ha individuato 23 pattern di progettazione (i design pattern) e li ha raggruppati in 3 famiglie (creazionali, strutturali e comportamentali)

Patterns...

- Un pattern in generale (da definizioni di Alexander) è definito come una “soluzione ad un problema in un contesto”
- La sua descrizione è definita da quattro importanti indicazioni:
  - Il **nome** del pattern
  - La descrizione del **problema** che il pattern risolve
  - La **soluzione** che si ritiene soddisfacente per il problema descritto
  - Le **conseguenze**, i **vincoli**, gli **sforzi** che occorrono per realizzare la soluzione proposta dal pattern

...Patterns

- Il GoF ridefinisce i connotati di quanto osservato da Alexander per l'achitettura; un pattern è quindi definito da:
  - Nome e classificazione (AKA event.)
  - Intento
  - Motivazioni (il problema che risolvono)
  - Applicabilità e Struttura (la soluzione fornita)
  - Partecipanti e collaboratori
  - Effetti (Consequences)
  - Implementazione e codice di esempio
  - Usi concreti e riferimenti ad altri pattern correlati

Benefici dell'uso dei pattern

- Riuso
- Base terminologica comune per l'identificazione dei problemi
- Elevamento della prospettiva (visione architetturale della soluzione)
- Capacità di valutazione della bontà di un progetto
- Migliore apprendimento
- Minore resistenza alle change requests
- Facilità di adozione di alternative progettuali

Strategie per la progettazione

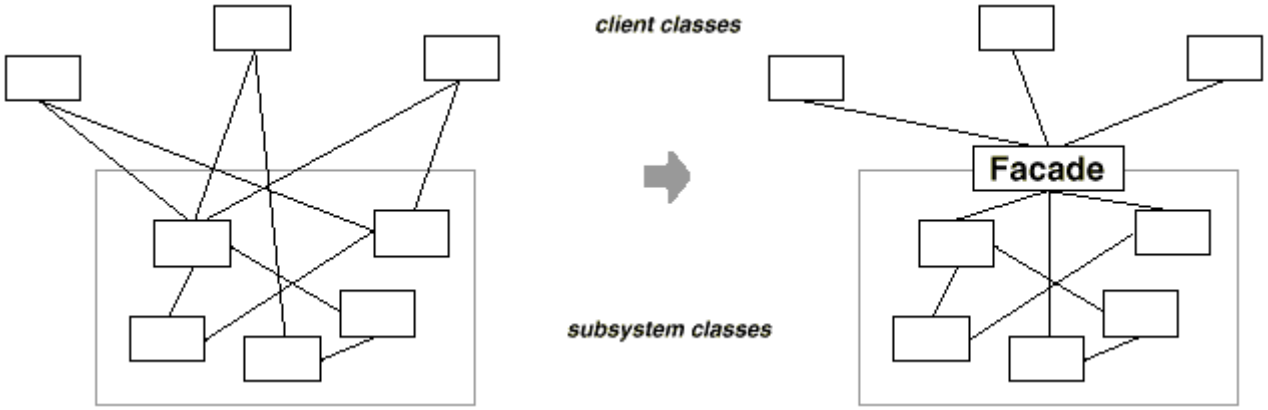
- Il GoF ha individuato delle strategie utili per uno sviluppo ad oggetti migliore:
  - Progettazione per interfacce
  - Preferire la composizione anziché l'ereditarietà
  - Individuare le parti variabili del problema e incapsularle
- Si è visto che i design pattern seguono (contengono) queste buone pratiche e aiutano a beneficiare dei vantaggi visti precedentemente

Famiglie di pattern

- Pattern creazionali
  - Astraggono il progesso di creazione di nuove istanze di oggetti; aiutano a rendere un sistema indipendente da come gli oggetti sono creati, composti e rappresentati
- Pattern strutturali
  - Guidano nella modalità di come classi e oggetti possono essere composti da strutture più grandi
- Pattern comportamentali
  - Guidano nella modalità di comunicazione tra gli oggetti e nell'assegnamento delle mutue responsabilità

Façade...

- È un pattern strutturale
- Nasconde la complessità di un'API complessa fornendo all'utilizzatore un'interfaccia amichevole



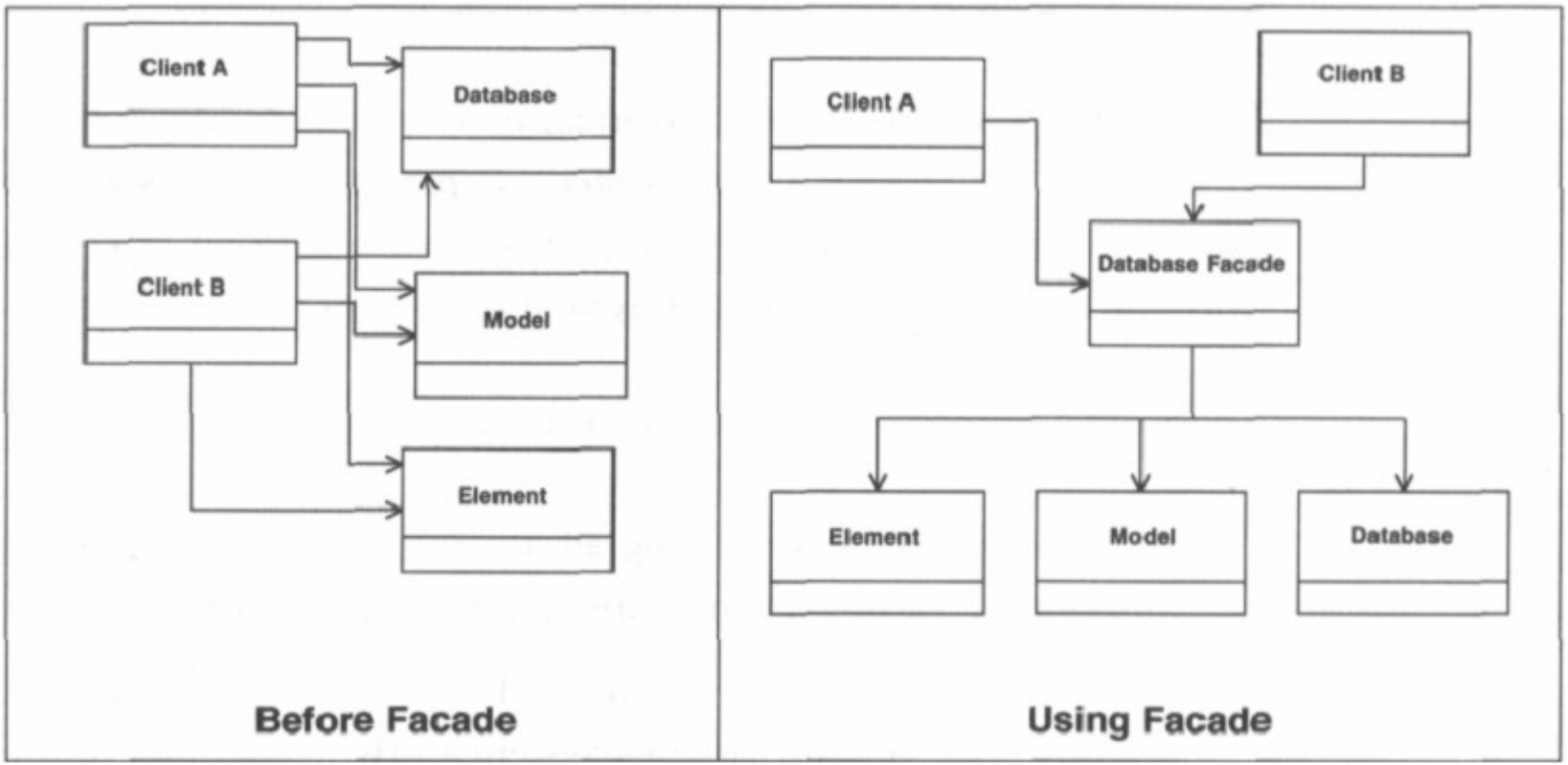
...Façade...

- Intent
  - You want to simplify how to use an existing system. You need to define your own interface.
- Problem
  - You need to use only a subset of a complex system. Or you need to inter-act with the system in a particular way.
- Solution
  - The Façade presents a new interface for the client of the existing system to use.
- Participants and Collaborators
  - It presents a specialized interface to the client that makes it easier to use.

...Façade

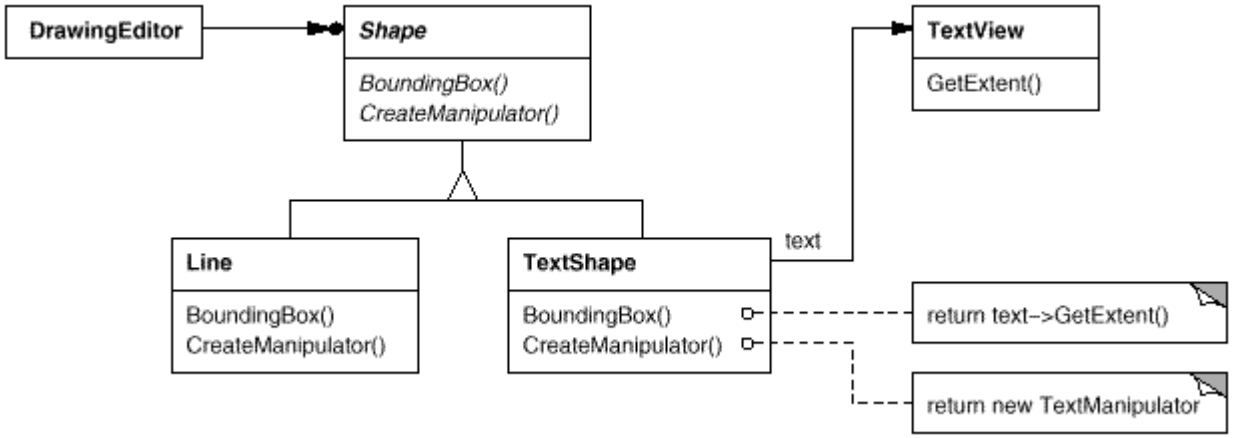
- Consequences
  - The Façade simplifies the use of the required subsystem. However, since the Façade is not complete, certain functionality may be unavailable to the client.
- Implementation
  - Define a new class (or classes) that has the required interface.
  - Have this new class use the existing system.

Applicare Façade: class diagram



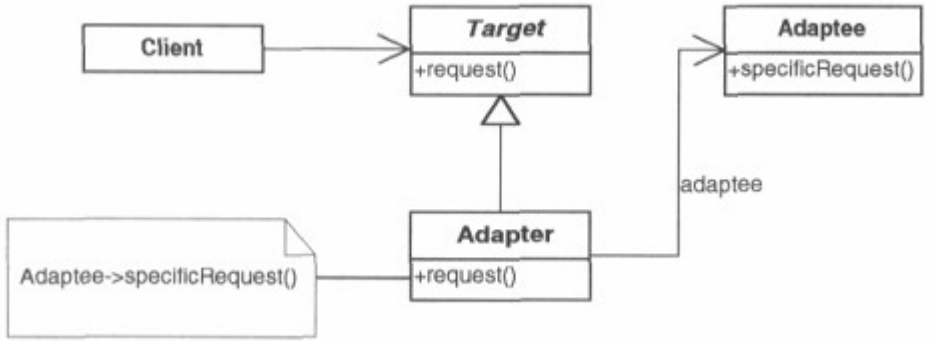
Adapter...

- AKA Wrapper
- È un pattern strutturale
- Converte l'interfaccia di una classe in una interfaccia diversa che il client si aspetta
- Può servire per far parlare due librerie diverse
- Supponiamo di dover progettare DrawingEditor e di disporre già di TextView...



...Adapter...

- TextShape è un'estensione di Shape e contiene una proprietà private di tipo TextView; con l'interfaccia richiesta da Shape TextShape chiama i metodi di TextView
- Questo consente al nostro DrawingEditor di trattare TextView (che ci è dato e non vogliamo cambiare) come uno Shape



...Adapter...

- Intent
  - Match an existing object beyond your control to a particular interface.
- Problem
  - A system has the right data and behavior but the wrong interface. Typically used when you have to make something a derivative of an abstract class we are defining or already have.
- Solution
  - The Adapter provides a wrapper with the desired interface.

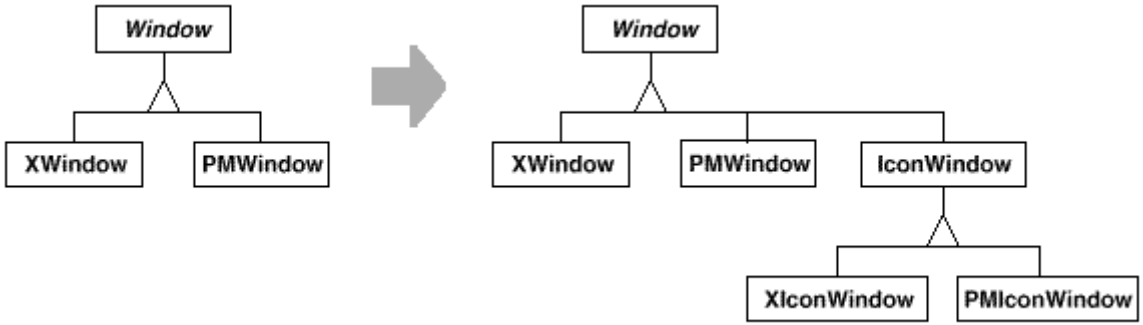
...Adapter

- Participants and Collaborators
  - The Adapter adapts the interface of an Adaptee to match that of the Adapter's Target (the class it derives from). This allows the Client to use the Adaptee as if it were a type of Target.
- Consequences
  - The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interfaces.
- Implementation
  - Contain the existing class in another class. Have the containing class match the required interface and call the methods of the contained class.

Bridge

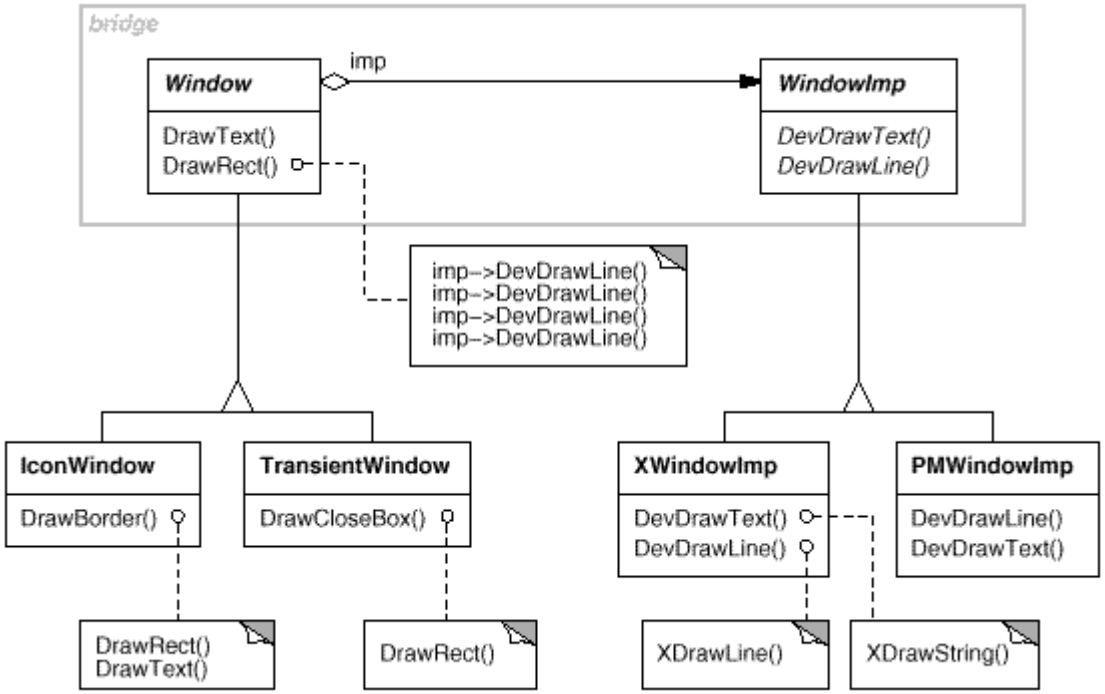
- È un pattern strutturale
- Ha il compito di disaccoppiare un concetto rispetto alla sua implementazione, in modo da consentire ad entrambi di variare
- Segue i due principi:
  - Incapsulamento delle variabilità
  - Favorire la composizione rispetto all'ereditarietà
- È utile quando sono richieste varie implementazioni di un concetto, ma poi quel concetto può anche essere esteso

...Bridge...



- Finestre
  - Vogliamo diverse implementazioni
  - Vogliamo essere liberi di estendere il concetto di finestra
- Le implementazioni non dovrebbero essere estensioni di un concetto, ma dovrebbero essere in un altro albero di ereditarietà
- Per usare una Window devo istanziare un oggetto foglia > Il codice diventa dipendente dall'implementazione

...Bridge...



- La soluzione è quella di adottare alberi diversi per le implementazioni e per le estensioni dei concetti
  - In WindowImp i metodi sono astratti e vengono implementati dalle implementazioni

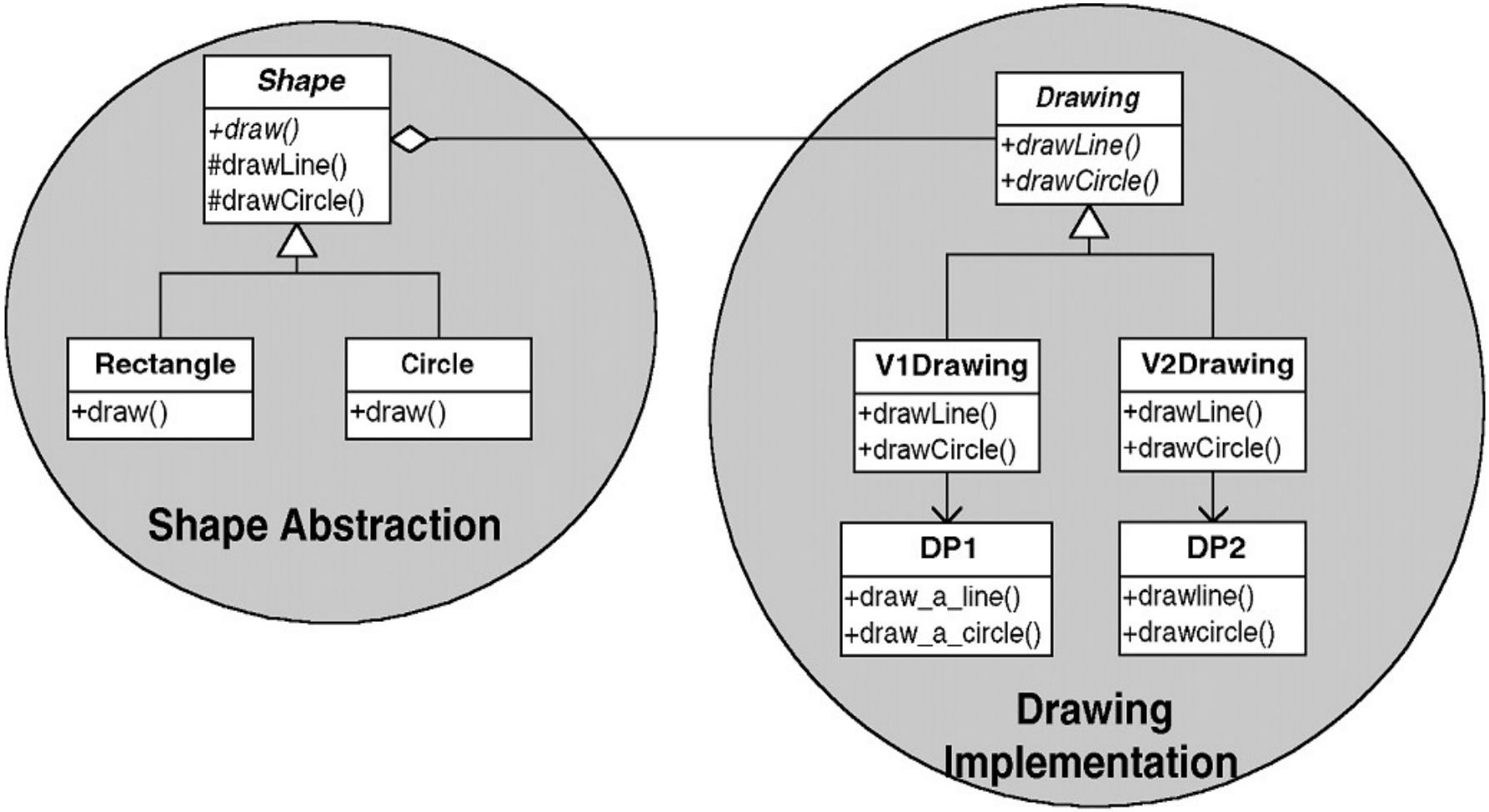
Bridge: esempio...

```
class Client {
  public static void main
    (String argv[]) {
      Shape r1, r2;
      Drawing dp;
      dp= new V1Drawing();
      r1= new Rectangle(dp,1,1,2,2);
      dp= new V2Drawing ();
      r2= new Circle(dp,2,2,3);
      r1.draw();
      r2.draw();
   }
}
abstract class Shape {
  abstract public draw() ;
  private Drawing _dp;
  Shape (Drawing dp) {
    _dp= dp;
  }
  public void drawLine (
    double x1,double y1,
    double x2,double y2) {
      _dp.drawLine(x1,y1,x2,y2);
  }
  public void drawCircle (
    double x,double y,double r) {
      _dp.drawCircle(x,y,r);
  }
}
abstract class Drawing {
  abstract public void drawLine (
    double x1, double y1,
    double x2, double y2);
abstract public void drawCircle (
    double x,double y,double r);
}
```

```
class V1Drawing extends Drawing {
  public void drawLine (
    double x1,double y1,
    double x2,double y2) {
    DP1.draw_a_line(x1,y1,x2,y2);
  }
  public void drawCircle (
    double x,double y,double r) {
    DP1.draw_a_circle(x,y,r);
  }
}
class V2Drawing extends Drawing {
  public void drawLine (
    double x1,double y1,
    double x2,double y2) {
    // arguments are different in DP2
    // and must be rearranged
    DP2.drawline(x1,x2,y1,y2);
  }
  public void drawCircle (
    double x, double y,double r) {
    DP2.drawcircle(x,y,r);
  }
}
class Rectangle extends Shape {
  public Rectangle (
    Drawing dp,
    double x1,double y1,
    double x2,double y2) {
      super( dp) ;
      _x1= x1; _x2= x2 ;
      _y1= y1; _y2= y2;
  }
  public void draw () {
   drawLine(_x1,_y1,_x2,_y1);
   drawLine(_x2,_y1,_x2,_y2);
   drawLine(_x2,_y2,_x1,_y2);
   drawLine(_x1,_y2,_x1,_y1);
  }
}
```

```
class Circle extends Shape {
  public Circle (
    Drawing dp,
    double x,double y,double r) {
      super( dp) ;
      _x= x; _y= y; _r= r ;
  }
  public void draw () {
   drawCircle(_x,_y,_r);
  }
}
// We've been given the implementations for DP1 and DP2
class DP1 {
  static public void draw_a_line (
    double x1,double y1,
    double x2,double y2) {
      // implementation
  }
  static public void draw_a_circle(
    double x,double y,double r) {
      // implementation
  }
}
class DP2 {
  static public void drawline (
    double x1,double x2,
    double y1,double y2) {
      // implementation
  }
  static public void drawcircle (
    double x,double y,double r) {
    // implementation
   }
}
```

...Bridge: esempio



Bridge: definizione...

- Intent
  - Decouple a set of implementations from the set of objects using them.
- Problem
  - The derivations of an abstract class must use multiple implementations without causing an explosion in the number of classes.
- Solution
  - Define an interface for all implementations to use and have the derivations of the abstract class use that.

...Bridge: definizione...

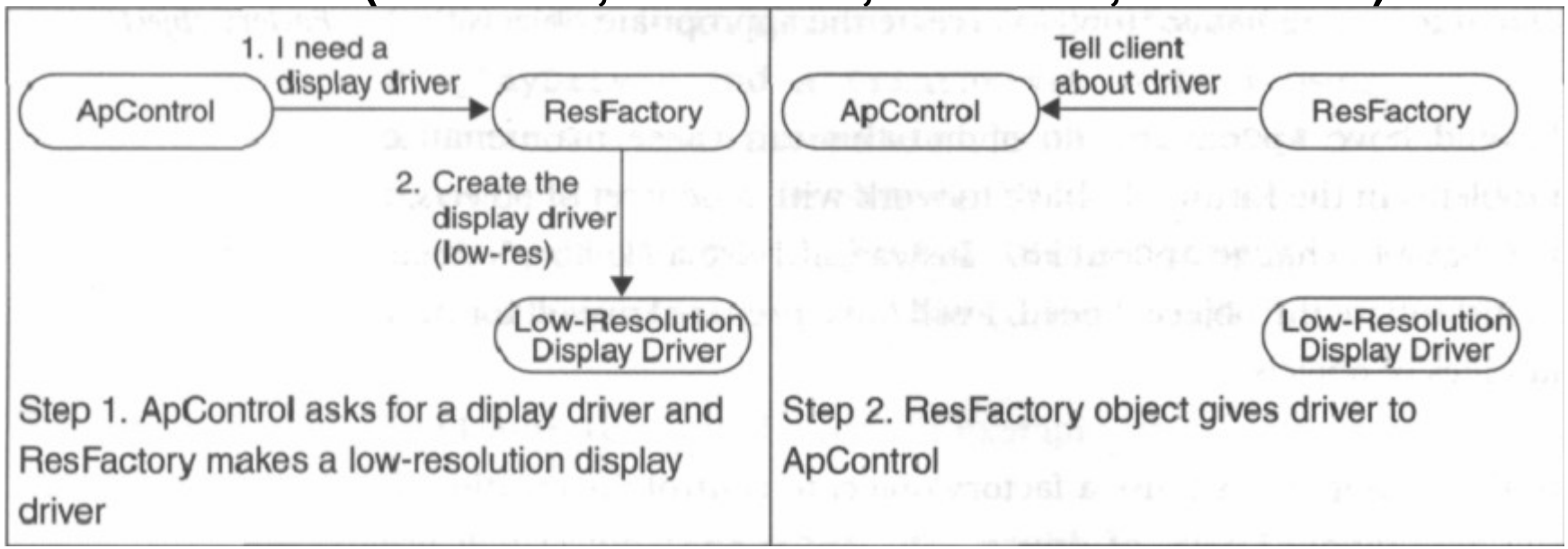
- Participants and Collaborators
  - The **Abstraction** defines the interface for the objects being implemented
  - The **Implementor** defines the interface for the specific implementation classes
  - Classes derived from the Abstraction use classes derived from the Implementor without knowing which particular ConcreteImplementor is in use
- Consequences
  - The decoupling of the implementations from the objects that use them increases extensibility. Client objects are not aware of implementation issues.

...Bridge: definizione

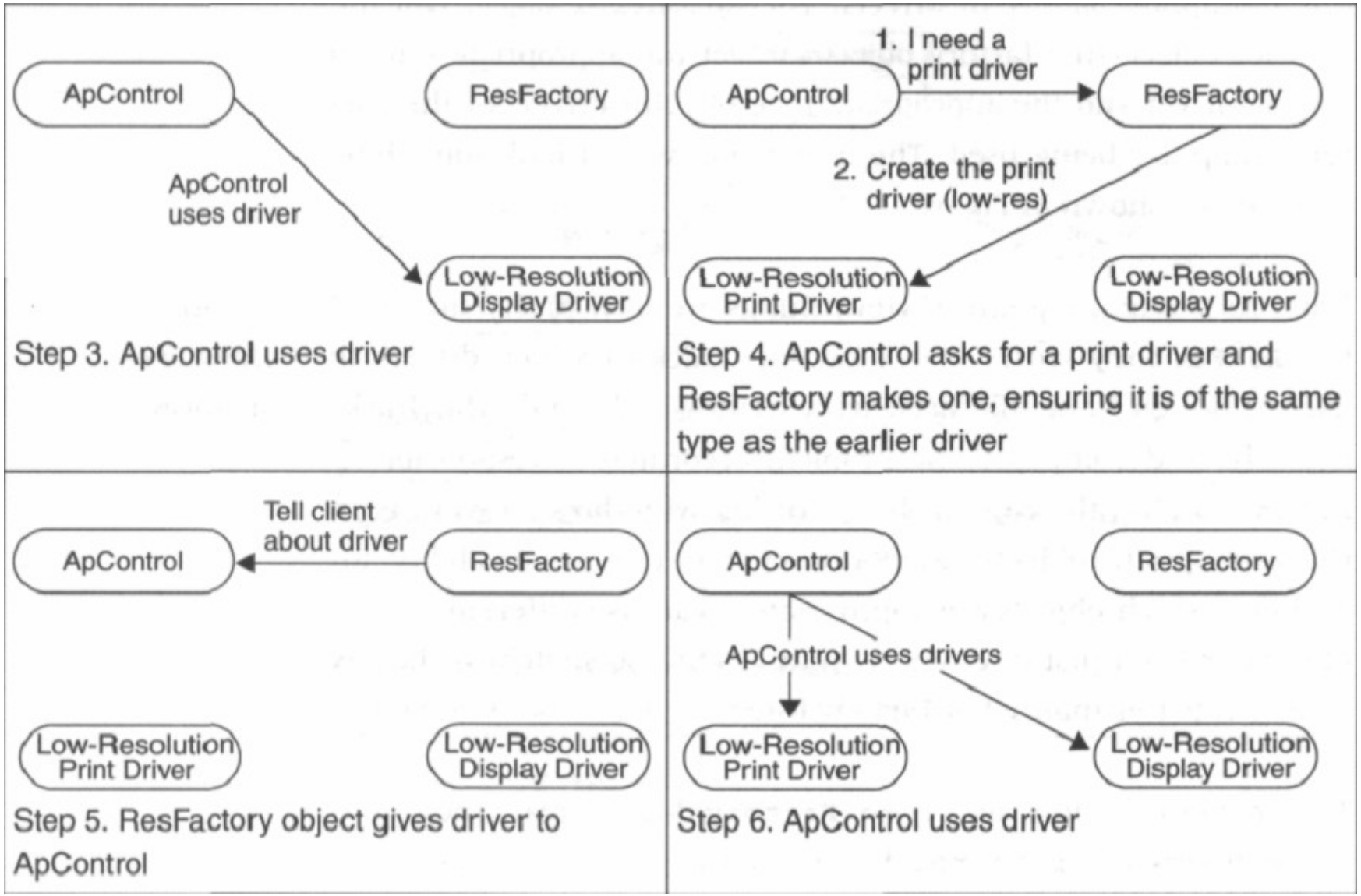
- Implementation
  - Encapsulate the implementations in an abstract class.
  - Contain a handle to it in the base class of the abstraction being implemented.
- Note: In Java, you can use interfaces instead of an abstract class for the implementation.

Abstract Factory...

- È un pattern creazionale
- È detto anche Kit
- Fornisce un'interfaccia per creare famiglie di oggetti senza conoscerne la classe concreta
- Esempio: driver per display e stampa in bassa e alta ris. (LRDD, HRDD, LRPD, HRPD)



...Abstract Factory...



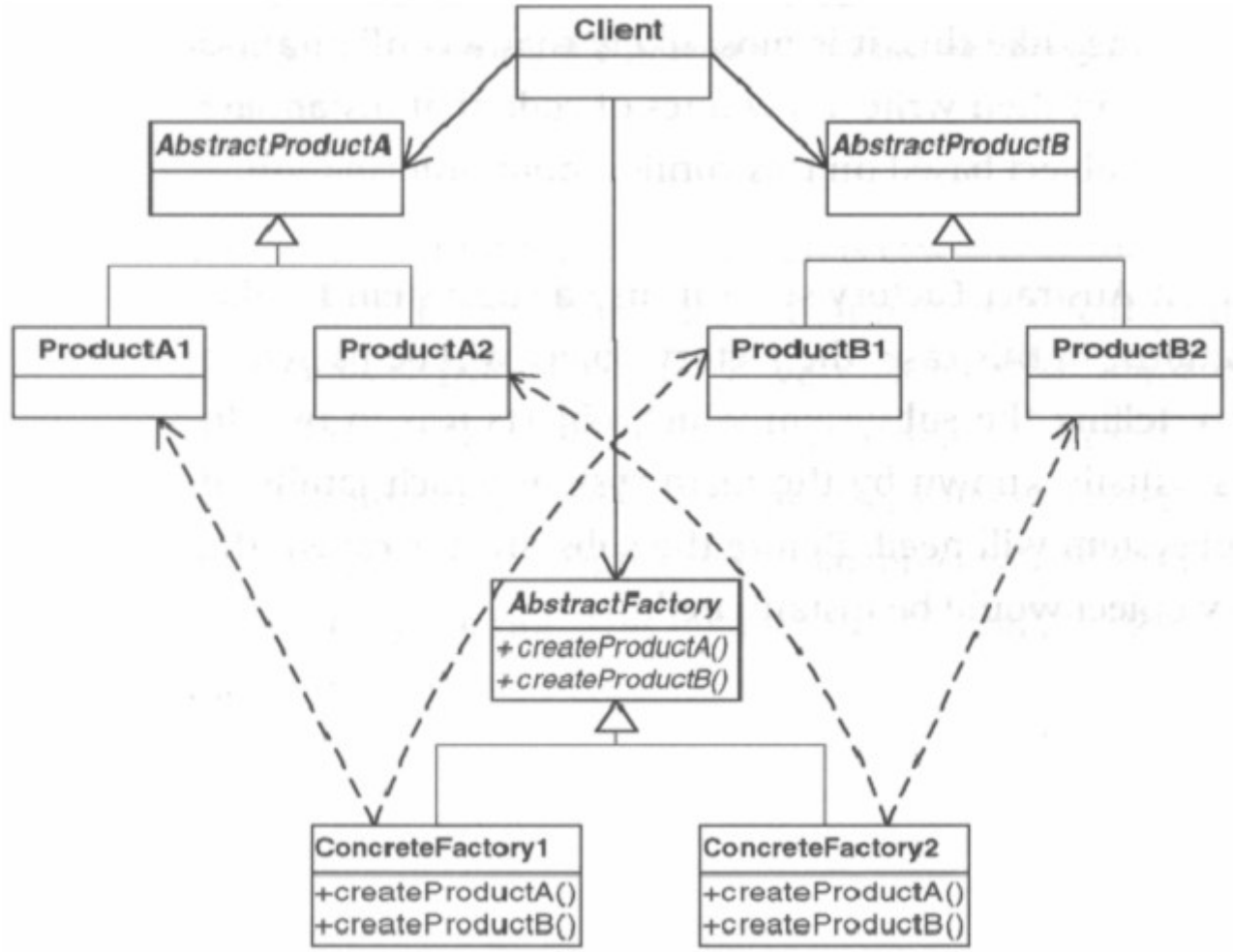
...Abstract Factory...

- Intent
  - You want to have families or sets of objects for particular clients (or cases).
- Problem
  - Families of related objects need to be instantiated.
- Solution
  - Coordinates the creation of families of objects. Gives a way to take the rules of how to perform the instantiation out of the client object that is using these created objects.

...Abstract Factory...

- Participants and Collaborators
  - The AbstractFactory defines the interface for how to create each member of the family of objects required. Typically, each family is created by having its own unique ConcreteFactory.
- Consequences
  - The pattern isolates the rules of which objects to use from the logic of how to use these objects.
- Implementation
  - Define an abstract class that specifies which objects are to be made. Then implement one concrete class for each family. Tables or files can also be used to accomplish the same thing.

...Abstract Factory



Il principio Aperto-Chiuso

- I moduli, i metodi e le classi dovrebbero essere aperti per l'estensione e chiusi per la modifica (Bertrand Meyer)
- Il Bridge Pattern ne è un esempio
  - Si possono aggiungere nuove implementazioni
  - Non cìè bisogno di toccare le classi esistenti

Strategy Pattern...

- È un pattern comportamentale
- Si basa su alcuni principi
  - Gli oggetti hanno responsabilità
  - Le diverse implementazioni di queste responsabilità vengono gestite con il polimorfismo
  - C'è la necessità di gestire diverse implementazioni come fossero una sola (concettualmente lo sono)
-

...Strategy Pattern...

- Intent
  - Allows you to use different business rules or algorithms depending upon the context in which they occur.
- Problem
  - The selection of an algorithm that needs to be applied depends upon the client making the request or the data being acted upon. If you simply have a rule in place that does not change, you do not need a Strategy pattern.
- Solution
  - Separates the selection of algorithm from the implementation of the algorithm. Allows for the selection to be made based upon context.

...Strategy Pattern...

- Participants and Collaborators
  - The strategy specifies how the different algorithms are used.
  - The concreteStrategies implement these different algorithms.
  - The Context uses the specific ConcreteStrategy with a reference of type Strategy. The strategy and Context interact to implement the chosen algorithm (sometimes the strategy must query the Context). The Context forwards requests from its Client to the Strategy.

...Strategy Pattern...

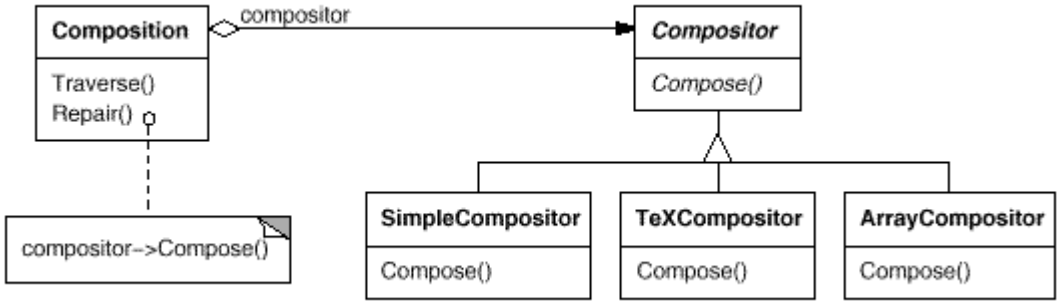
- Consequences
  - The Strategy pattern defines a family of algorithms.
  - Switches and/or conditionals can be eliminated.
  - You must invoke all algorithms in the same way (they must all have the same interface). The interaction between the ConcreteStrategies and the Context may require the addition of getstate type methods to the Context.
- Implementation
  - Have the class that uses the algorithm (the Context) contain an abstract class (the stragegy) that has an abstract method specifying how to call the algorithm. Each derived class implements the algorithm as needed.

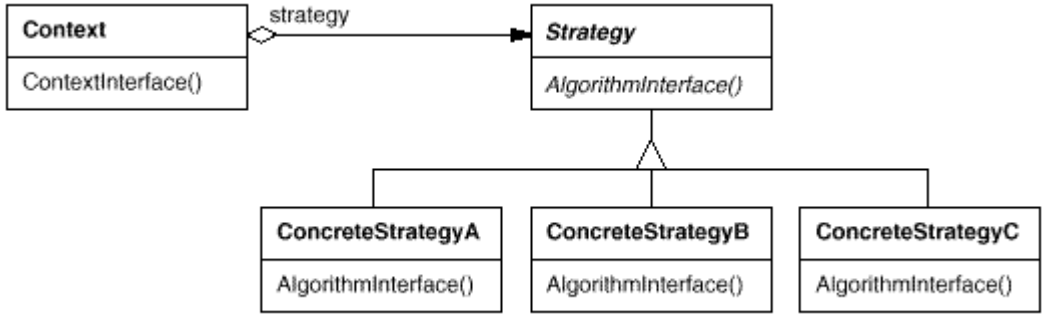
...Strategy Pattern...

- Note:
  - this method wouldn't be abstract if you wanted to have some default behavior.
- Note:
  - In the prototypical Strategy pattern, the responsibility for selecting the particular implementation to use is done by the Client object and is given to the context of the Strategy pattern.

...Strategy Pattern

- Esempio e generalizzazione: una composizione (un testo) e una famiglia di algoritmi per gestire le interruzioni di riga





Strategy: esempio C++

- Senza Strategy

```
void Composition::Repair () {
        switch (_breakingStrategy) {
                case SimpleStrategy:
                        ComposeWithSimpleCompositor();
                        break;
                case TeXStrategy:
                        ComposeWithTeXCompositor();
                        break; // ...
                }
        // merge results with existing composition, if necessary
}
```

- Con Strategy

```
void Composition::Repair () {
        _compositor->Compose();
        // merge results with existing composition, if necessary
}
```