Design patterns

Ingegneria del software

Vincenzo Bonnici
Corso di Laurea in Informatica
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università degli Studi di Parma

2025-2026

Viste architetturali

- L'architettura software di un sistema è un artefatto, frutto della attività di progettazione

- Una architettura software è descritta dai suoi componenti (e sottosistemi) e dalle relazioni tra essi
  - Componenti, costituiscono i 'building block' di un sistema (es: moduli, classi, funzioni)
  - Relazioni, denotano una connessione tra componenti: aggregazione, eredità, interazione

- Un sottosistema è un sistema le cui operazioni sono indipendenti dai servizi di altri sottosistemi

- Un componente fornisce servizi ad altri componenti e non è considerato come un sistema a sé stante

- Vari punti di vista possono produrre varie architetture che rappresentano prospettive diverse del sistema
  - Mostrare i componenti del sistema
  - Definire le interfacce tra sottosistemi

Stili architetturali

- L’architettura può conformare un **modello** o stile: si parla di architetture di riferimento, architectural pattern e stili architetturali

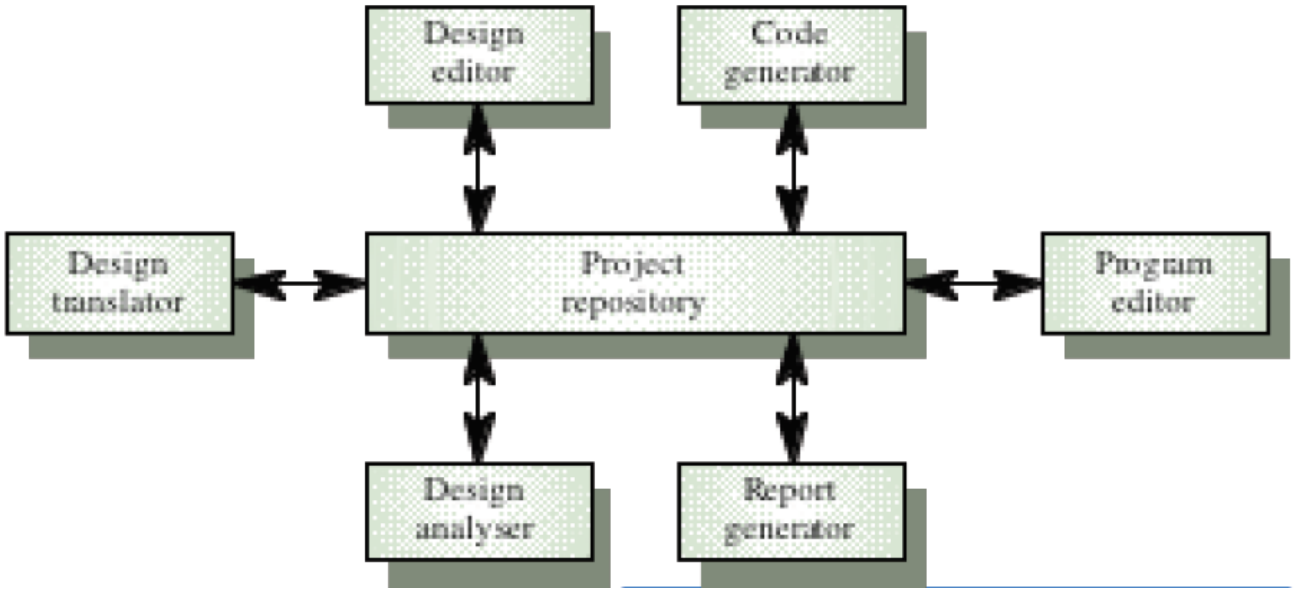
- La conoscenza degli stili può semplificare la definizione dell’architettura per un sistema

- I sistemi più grandi sono eterogenei e non seguono un singolo stile

- Caratteristiche importanti che una architettura dovrebbe esibire
  - Modularità
  - Indipendenza funzionale
  - Information hiding

Modello Repository (blackboard)

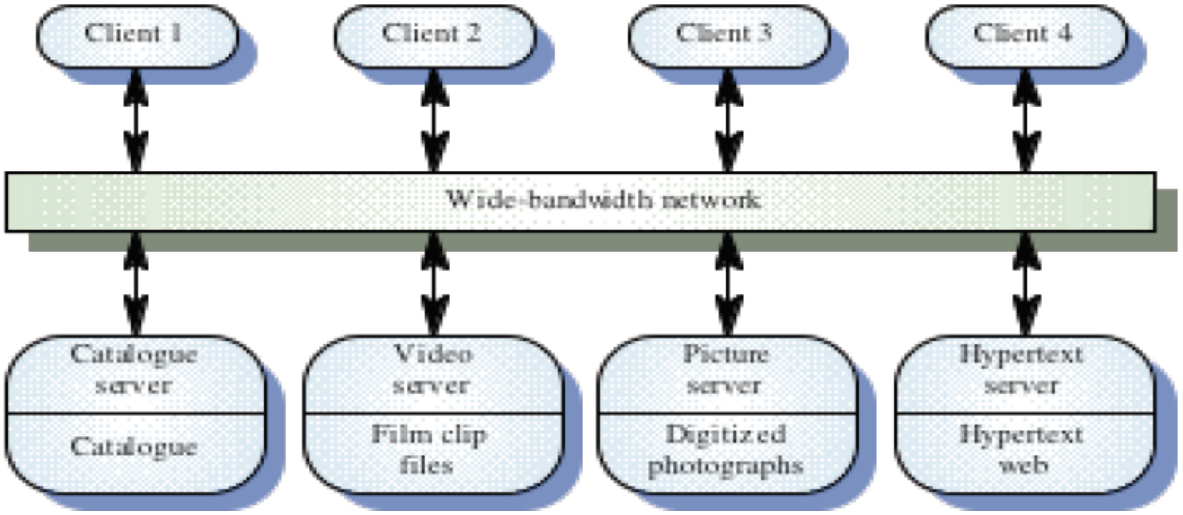
- I sottosistemi necessitano di scambiare dati
- I dati sono tenuti in un database centrale o repository accessibile da tutti i sottosistemi



- Vantaggi
  - Modo efficiente di condividere grandi quantità di dati
  - I sottosistemi non necessitano di sapere come i dati sono prodotti
  - La struttura del repository è pubblicata
- Svantaggi
  - I sottosistemi devono scegliere una struttura del repository (compromesso)
  - Cambiamenti sui dati possono essere difficili (coinvolgono tutti i sottosistemi)
  - Difficile distribuire i dati efficientemente

Modello Client-Server

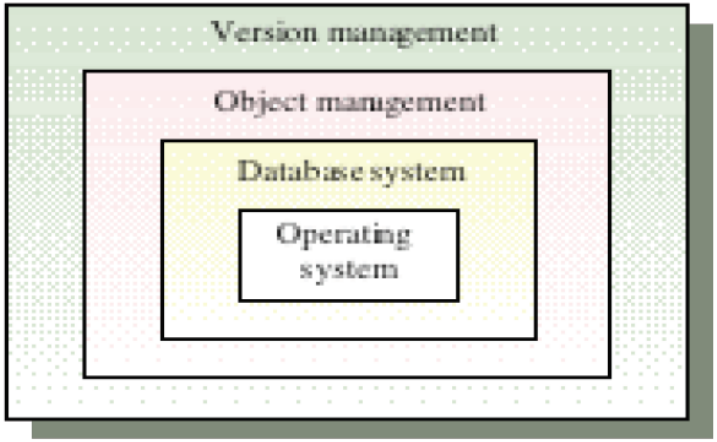
- Set di server che forniscono servizi specifici
    - Stampa, gestione dati, etc.
- Set di client che chiamano i server
- La rete permette ai client di accedere ai server



- Vantaggi
    - Usa efficacemente sistemi di rete
        - Può richiedere hardware meno costoso
    - Aggiungere nuovi server o aggiornare server è facile
- Svantaggi
    - Scambio di dati può essere inefficiente
    - Management ridondante su ciascun server
    - Non esiste un registro di nomi e servizi
        - Può essere difficile trovare i servizi disponibili

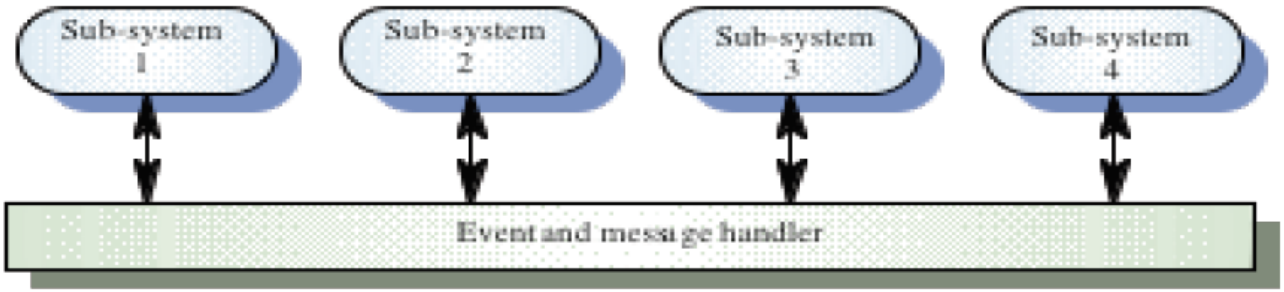
Modello a macchina astratta

- Organizza un sistema in un set di strati (o macchine astratte) ognuno dei quali fornisce un set di servizi
- Supporta lo sviluppo incrementale dei diversi livelli. Quando l'interfaccia di un livello cambia solo il livello adiacente è influenzato



Modello event-driven

- Eventi esterni pilotano i sottosistemi che processano gli eventi
- Broadcast
  - Un evento è inviato a tutti i sottosistemi
  - Ogni sottosistema che è in grado di gestire l'evento lo gestisce
  - I sottosistemi decidono se l'evento è di interesse
- Interrupt-driven
  - Usati in sistemi realtime
- Esempi: fogli elettronici, sistemi di produzione



Design pattern

- I **design pattern** sono schemi generali di soluzioni per problemi ricorrenti.
- Un pattern consiste nella **descrizione sintetica** di un **problema** e della **relativa soluzione.**
- Specifica gli elementi strutturali (classi, componenti, interfacce. . .), le relazioni reciproche e il loro modo di interagire.
- La descrizione viene integrata con una discussione dei vantaggi e svantaggi della soluzione, delle condizioni di applicabilità e delle possibili tecniche di implementazione.
- Normalmente vengono presentati anche dei casi di studio.
- I design pattern si usano principalmente nella fase intermedia fra progetto di sistema e progetto dettagliato, ma anche nel progetto di sistema (architetture standard).

Design pattern

- I design pattern sono organizzati sul catalogo in base allo scopo

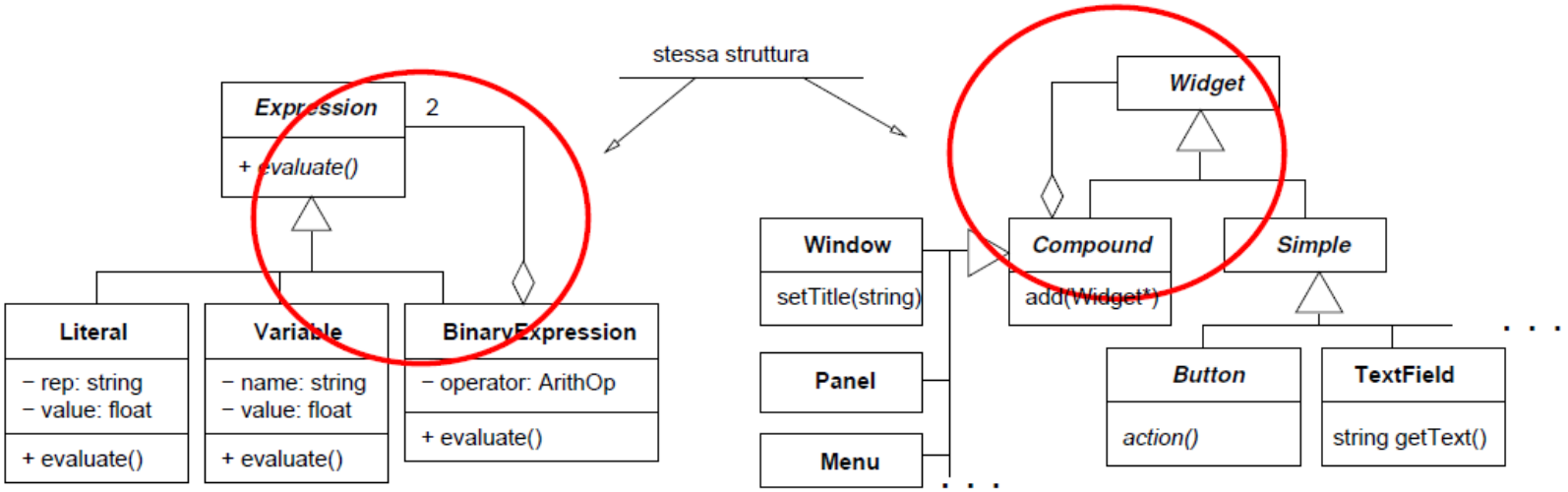
- **Creazionali**: riguardano la creazione di istanze
  - Permettono di astrarre il processo di creazione oggetti: rendono un sistema indipendente da come i suoi oggetti sono creati, composti, e rappresentati
  - Singleton, Factory Method, Abstract Factory, Builder, Prototype

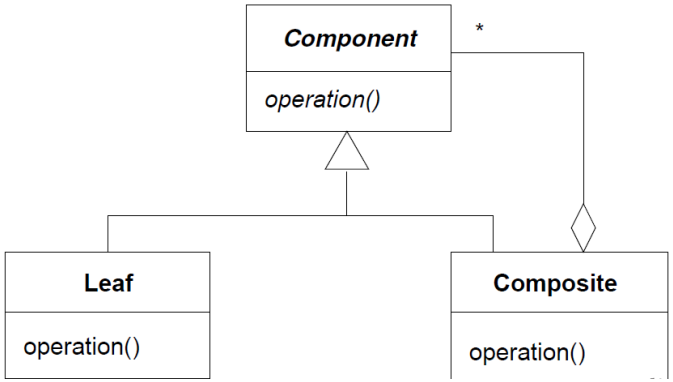
- **Strutturali**: riguardano la scelta della struttura
  - Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy

- **Comportamentali**: riguardano la scelta dell'incapsulamento di algoritmi
  - Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visito

Il pattern **Composite**

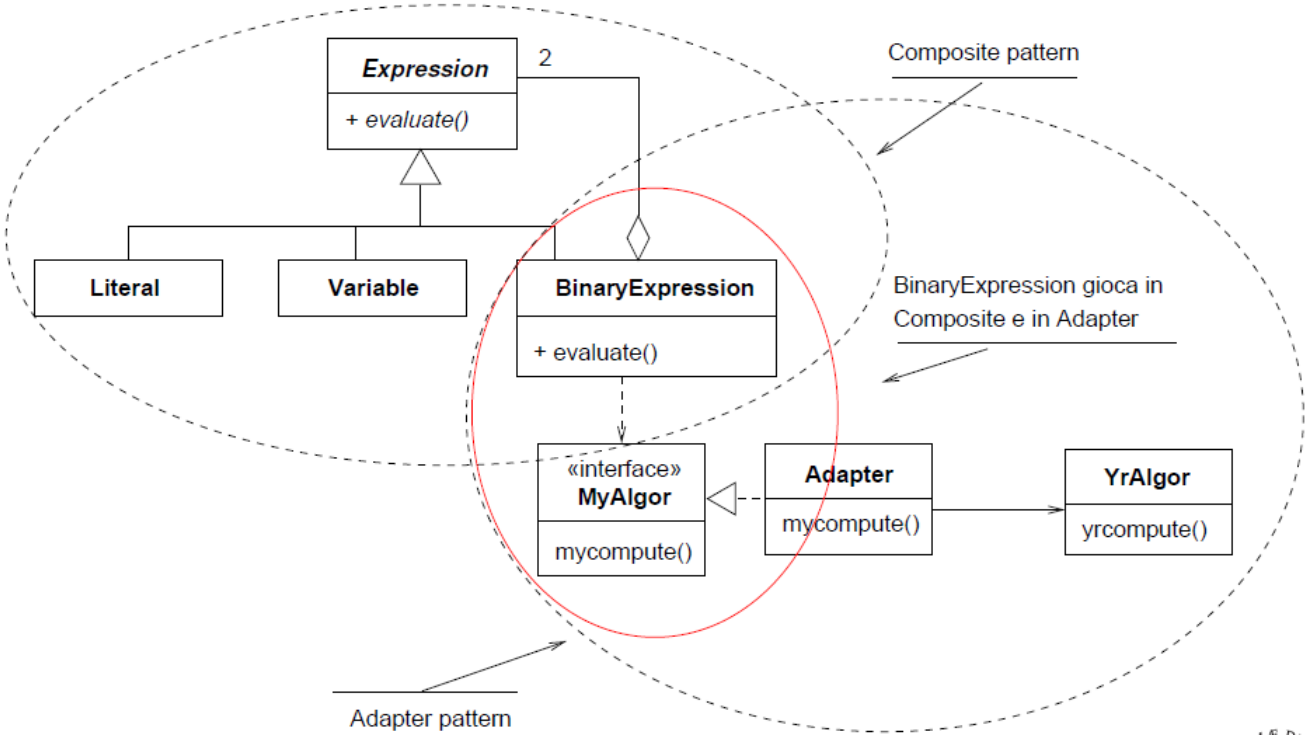
- **scopo**: offrire un'interfaccia comune a classi le cui istanze possono far parte di strutture gerarchiche.





Design pattern

- Un design pattern non è un componente software, ma solo uno schema di soluzione per un particolare aspetto del funzionamento di un sistema.
- Gli elementi strutturali di un pattern rappresentano dei ruoli che saranno interpretati dalle entità effettivamente realizzate.
- Ciascuna di queste entità può interpretare un ruolo diverso in diversi pattern, poiché in un singolo sistema (o sottosistema) si devono risolvere diversi problemi con diversi pattern.
- Un pattern non è una ricetta rigida da applicare meccanicamente, ma uno schema che deve essere adattato alle diverse situazioni, anche con un po' d'inventiva.



Il pattern **Adapter**

- **scopo**: adattare un'interfaccia offerta ad un'interfaccia richiesta.
  Adapter permette ad alcune classi di interagire, eliminando il problema di interfacce incompatibili

- Problema:
  - Alcune volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia che si aspetta l'applicazione. Ovvero nome metodo, parametri, tipo parametri di chiamate all'interno dell'applicazione non sono corrispondenti a quelli offerti da una classe di libreria

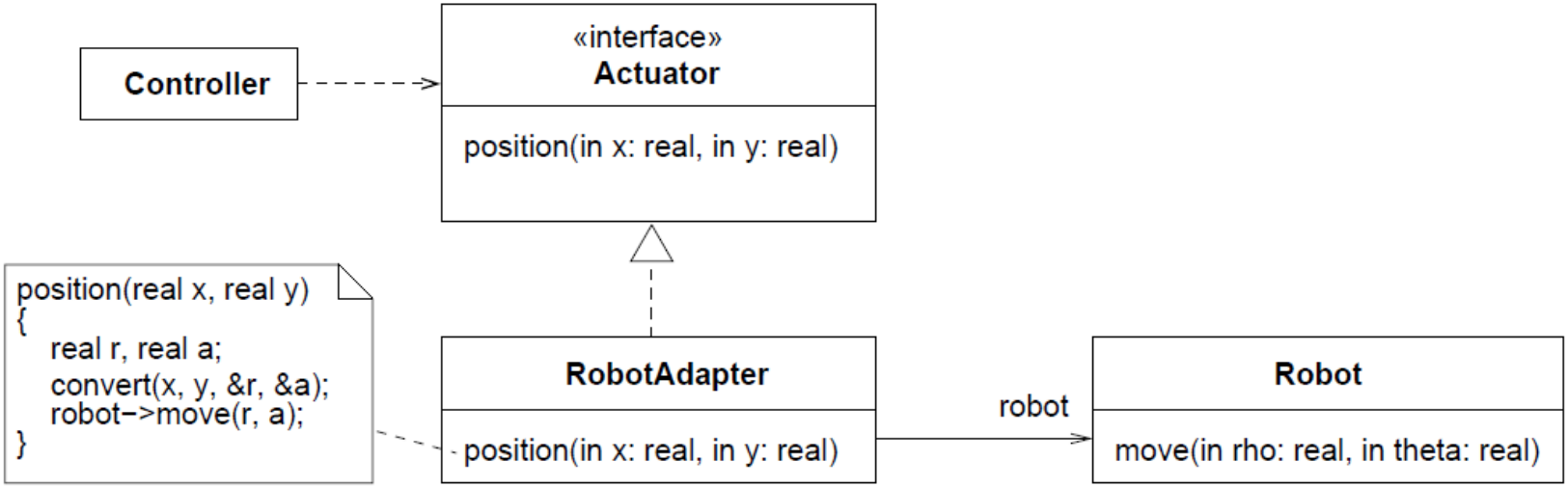
  - Non è possibile cambiare l'interfaccia della libreria, poiché non si ha il sorgente (comunque non conviene cambiarla)

  - Non è possibile cambiare l'applicazione, e si può voler cambiare quale metodo invocare, senza renderlo noto al chiamante

Il pattern **Adapter**

- **esempio**: la classe Controller richiede l'interfaccia Actuator. La semantica dell'operazione astratta position() è "posizionare un utensile sul punto di coordinate cartesiane (x, y)". L'interfaccia offerta dalla classe Robot ha l'operazione move() che posiziona l'utensile sul punto di coordinate polari (p,O).

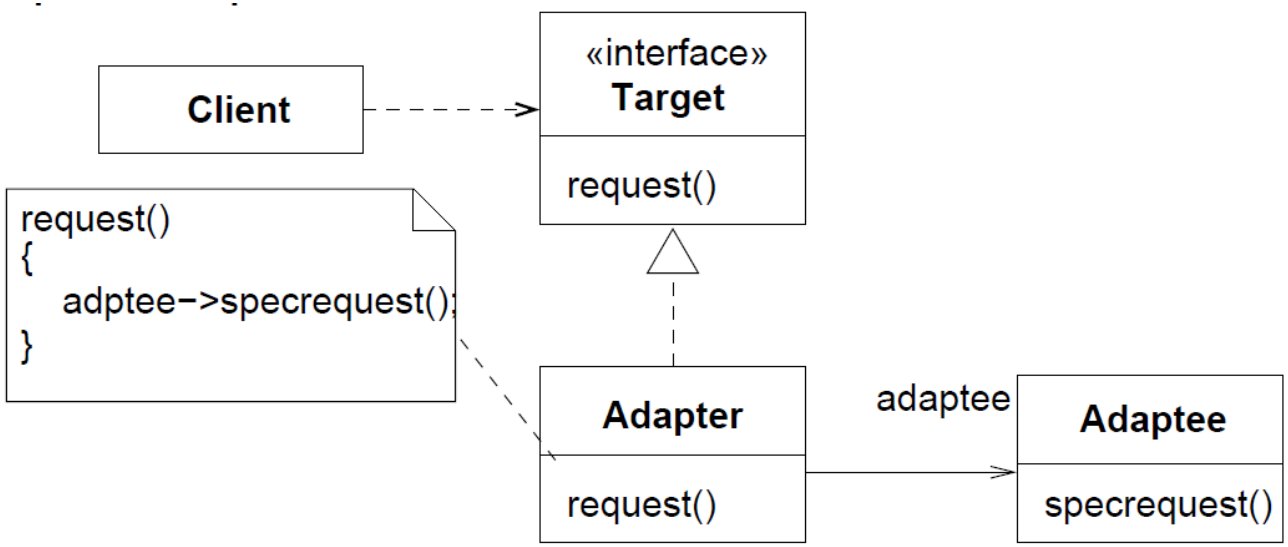
- **soluzione**: inserire la classe RobotAdapter che realizza Actuator trasformando le coordinate cartesiane in polari, e chiamando l'operazione move().



Il pattern **Adapter**

- **esempio**: la classe Controller richiede l'interfaccia Actuator. La semantica dell'operazione astratta position() è "posizionare un utensile sul punto di coordinate cartesiane (x, y)". L'interfaccia offerta dalla classe Robot ha l'operazione move() che posiziona l'utensile sul punto di coordinate polari (p,O).

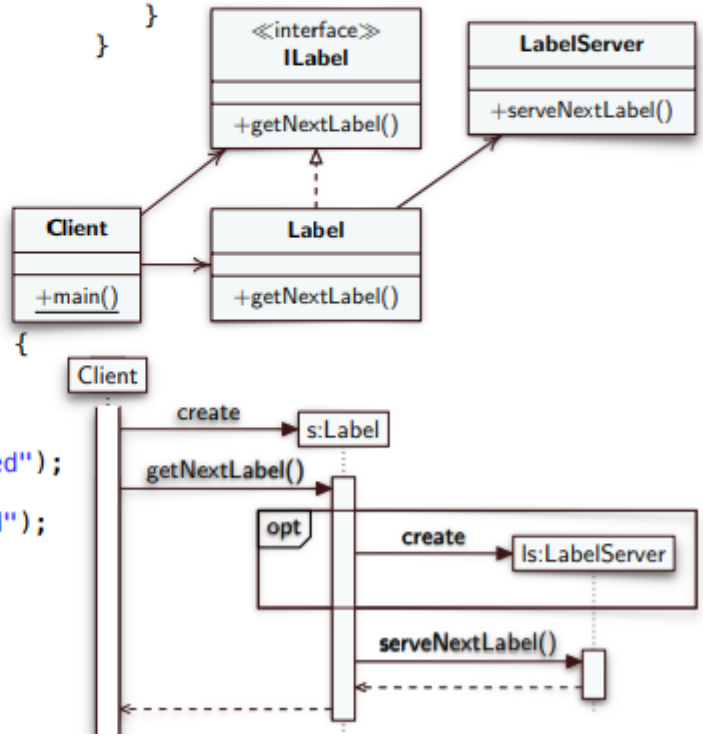
- **soluzione**: inserire la classe RobotAdapter che realizza Actuator trasformando le coordinate cartesiane in polari, e chiamando l'operazione move().



Il pattern **Adapter**

```java
public interface ILabel { // Target
   public String getNextLabel();
}

// Adapter
public class Label implements ILabel {
   private LabelServer ls;
   private String p;
   public Label(String prefix) {
      p = prefix;
   }
   public String getNextLabel() {
      if (ls == null)
         ls = new LabelServer(p);
      return ls.serveNextLabel();
   }
}

public class Client {
   public static void main(String args[]) {
      ILabel s = new Label("LAB");
      String l = s.getNextLabel();
      if (l.equals("LAB1"))
         System.out.println("Test 1:Passed");
      else
         System.out.println("Test1:Failed");
   }
}
```

```java
public class LabelServer { // Adaptee
   private int labelNum = 1;
   private String labelPrefix;
   public LabelServer(String prefix) {
      labelPrefix = prefix;
   }
   public String serveNextLabel() {
      return labelPrefix + labelNum++;
   }
}
```



Il pattern **Proxy virtuale**

- **scopo**: differire operazioni costose. Definire un sostituto o surrogato per un oggetto che controlla gli accessi all'oggetto target. I client comunicano con il surrogato anziché comunicare con l'oggetto target

- **Problema**:
  - All'apertura di un documento si vogliono ridurre i tempi di creazione di oggetti che contengono immagini, si usa un surrogato per le immagini non visibili
  - Si vuol controllare (ed in alcuni casi permettere) l'accesso ad un oggetto
  - Si vuol rendere più facile l'accesso ad un oggetto remoto

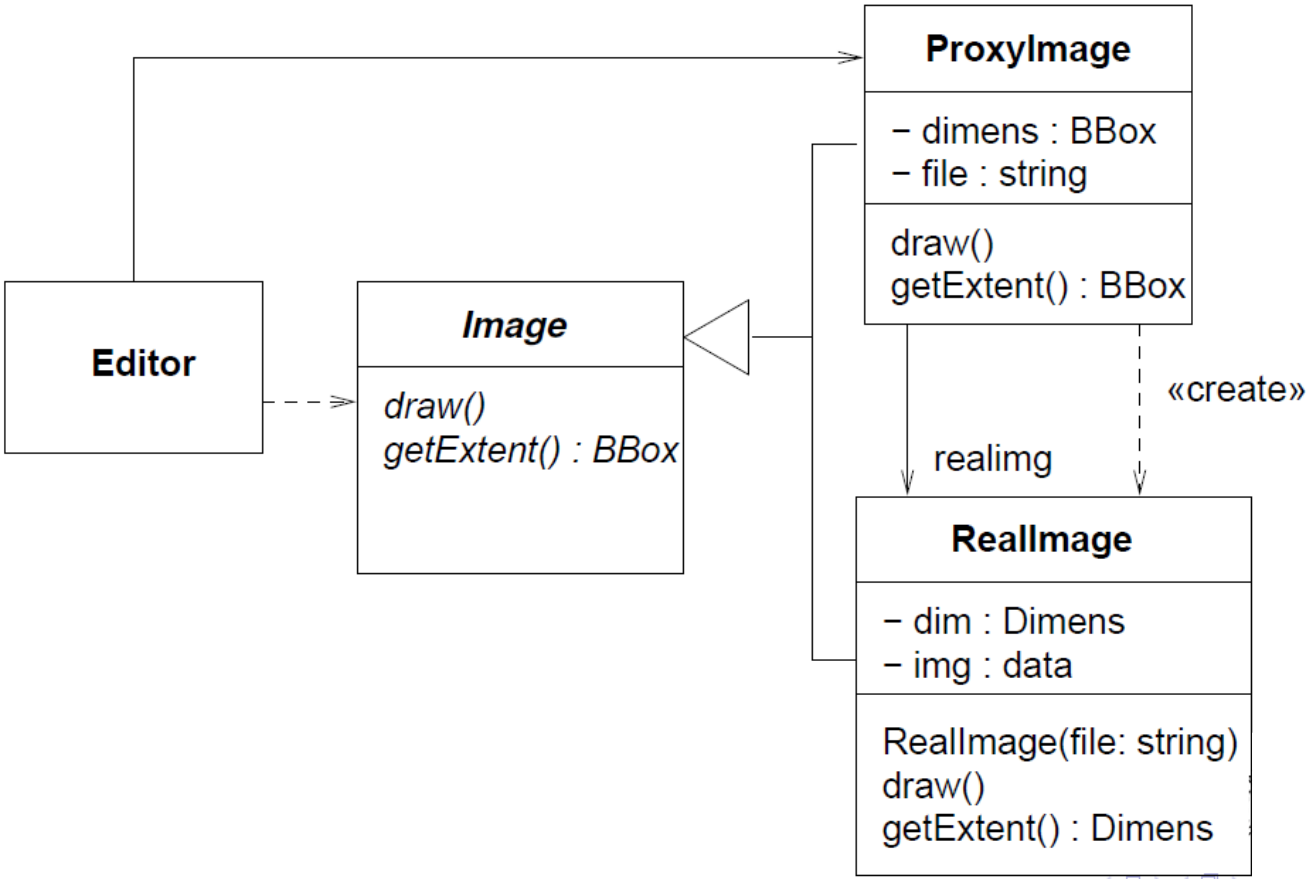
- **Motivazione**: l'accesso all'oggetto target dovrebbe essere trasparente e semplice per il client
  - Ovvero, per l'es. dell'apertura documento, si vuol nascondere al client che l'immagine sia stata creata solo quando serve, così da non complicare gli oggetti client
  - Il client non deve cambiare il modo in cui chiama gli oggetti che usa

Il pattern **Proxy virtuale**

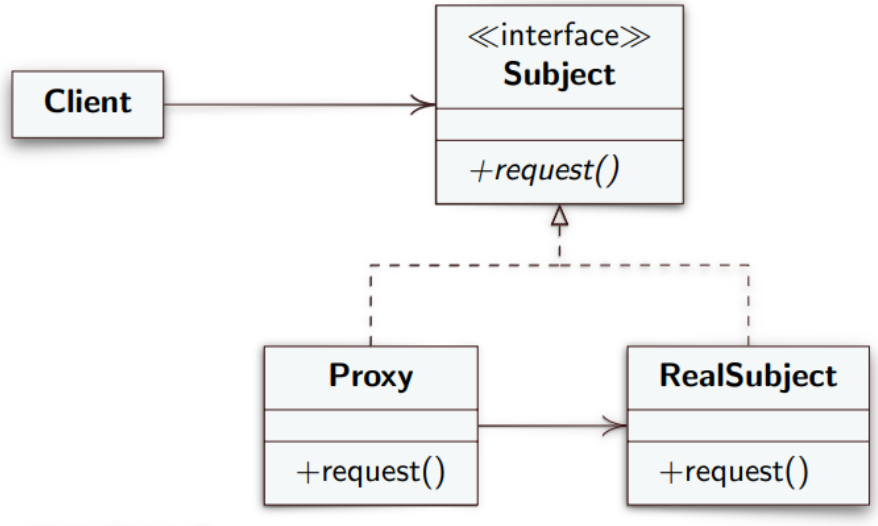
- **esempio**: Un programma di videoscrittura deve inserire nel testo delle immagini, implementate da istanze di una classe Image che offre le operazioni draw(), che carica in memoria l'immagine e la disegna, e getExtent(), che restituisce le dimensioni nell'immagine. Per impaginare il testo basta che siano note le dimensioni delle immagini, quindi conviene differire il caricamento dell'immagine fintanto che non è necessario visualizzarla.

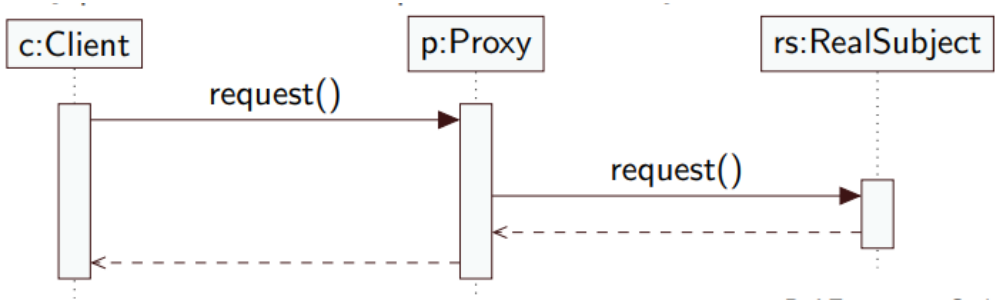
- **soluzione**: un'istanza della classe ProxyImage fa da segnaposto per RealImage, che contiene una struttura dati per rappresentare l'immagine. Le chiamate all'operazione getExtent() vengono eseguite direttamente da ProxyImage, mentre le chiamate a draw() vengono delegate a RealImage, che viene istanziata solo alla prima invocazione di draw().

Il pattern **Proxy virtuale**



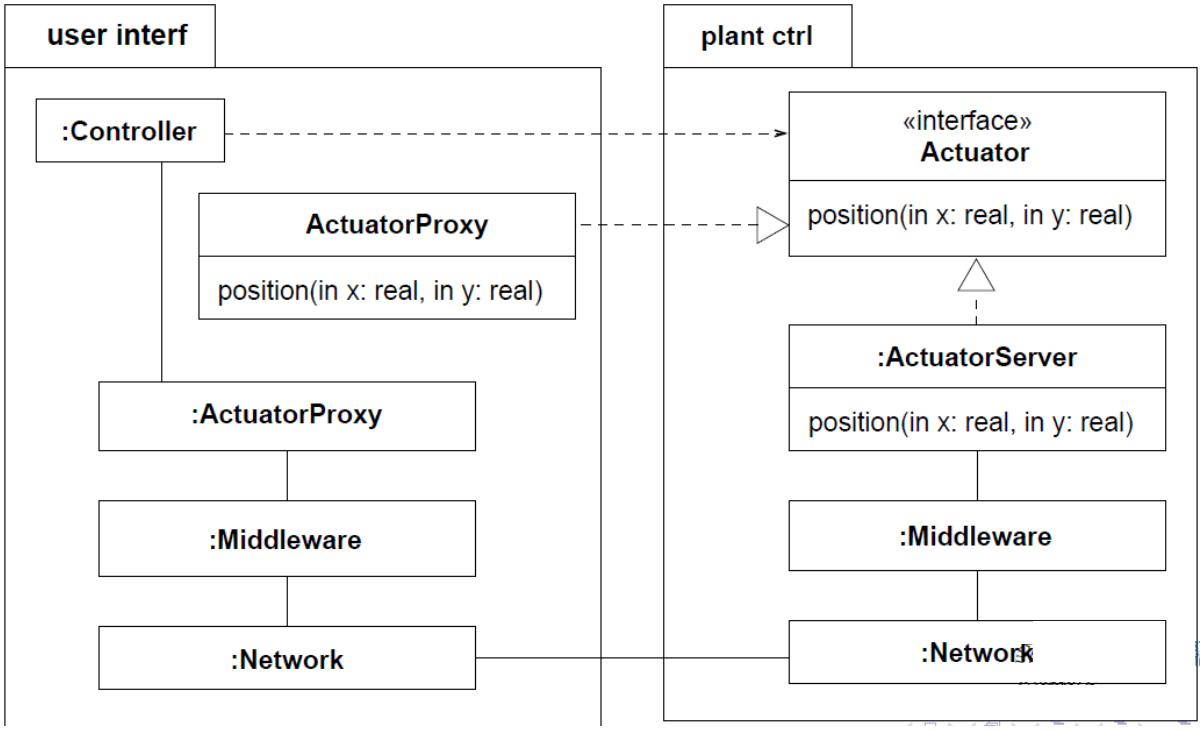
Il pattern **Proxy virtuale**





Il pattern **Proxy remoto**

- **scopo**: chiamare da remoto le operazioni di un oggetto.
- **esempio**: un impianto deve essere comandato remotamente.
- **soluzione**: l'interfaccia utente (locale) contiene un'istanza della classe ActuatorProxy che realizza l'interfaccia di programmazione dell'impianto da controllare. Le operazioni di ActuatorProxy inviano messaggi al sistema di controllo remoto, usando un'infrastruttura di comunicazione. Dal lato dell'impianto, i messaggi vengono convertiti in chiamate a un'istanza di ActuatorServer.



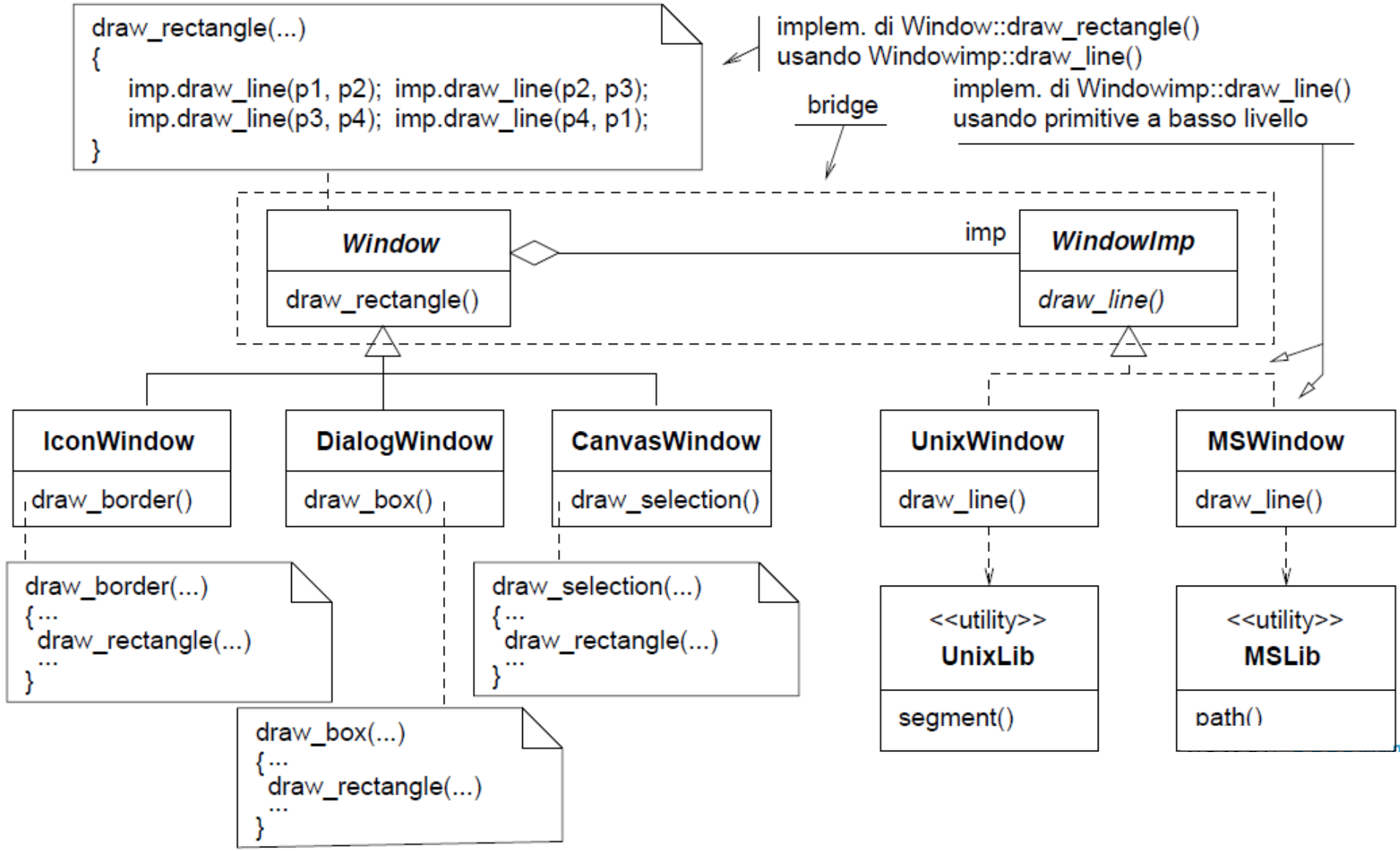
Il pattern **Proxy Bridge**

- **scopo**: disaccoppiare una famiglia di classi dalla loro implementazione.

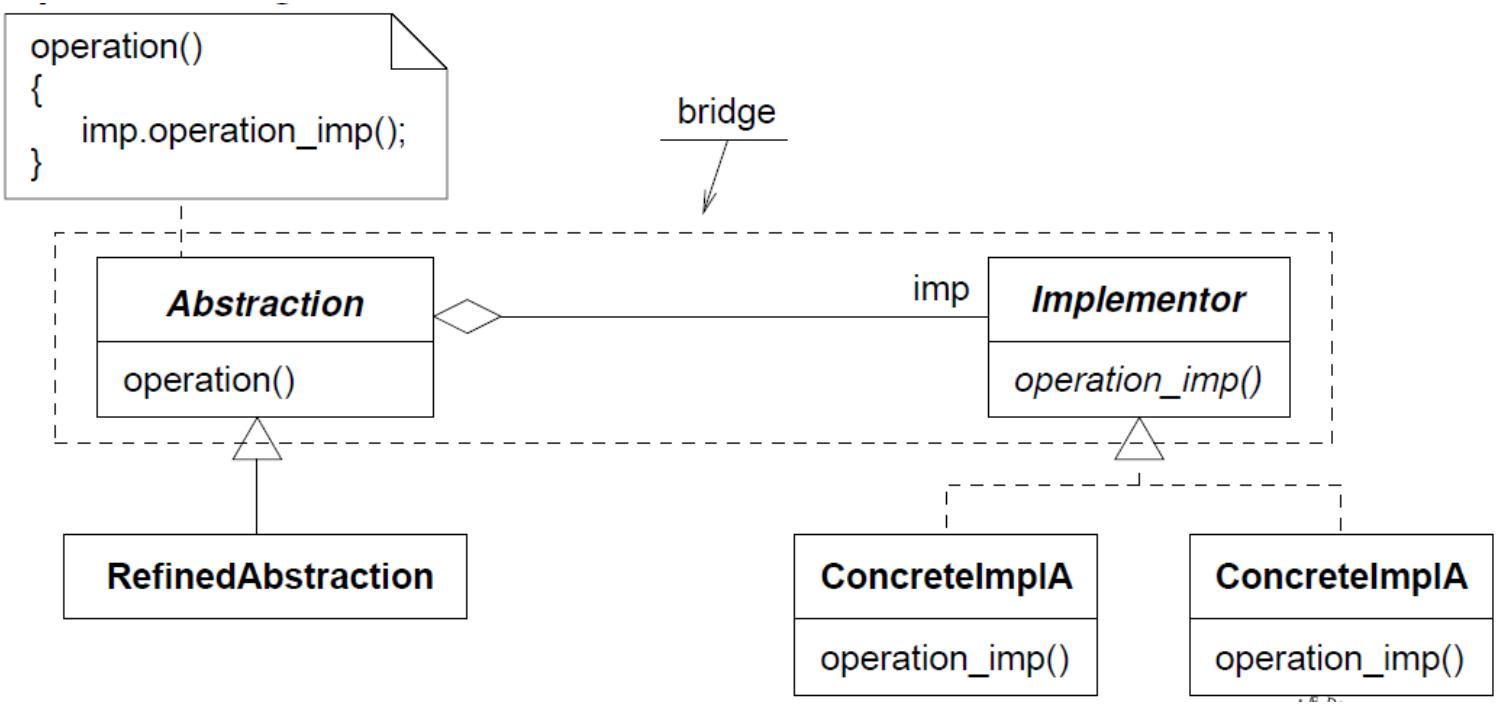
- **esempio**: un'interfaccia grafica (GUI) comprende una famiglia di classi derivate dalla clase Window; vogliamo poter implementare l'interfaccia grafica usando diverse librerie grafiche, modificando la GUI indipendentemente dalla libreria.

- **soluzione**: la classe Window, che offre operazioni ad alto livello (p.es., draw_rectangle()) dipende da una classe astratta (o interfaccia) WindowImp a piú basso livello (p.es., draw_line()). Le operazioni di Window e delle classi derivate sono implementate con operazioni (astratte) di WindowImp. Queste sono implementate con le operazioni delle librerie grafiche.

Il pattern **Proxy Bridge**



Il pattern **Proxy Bridge**



Il pattern **Factory Method**

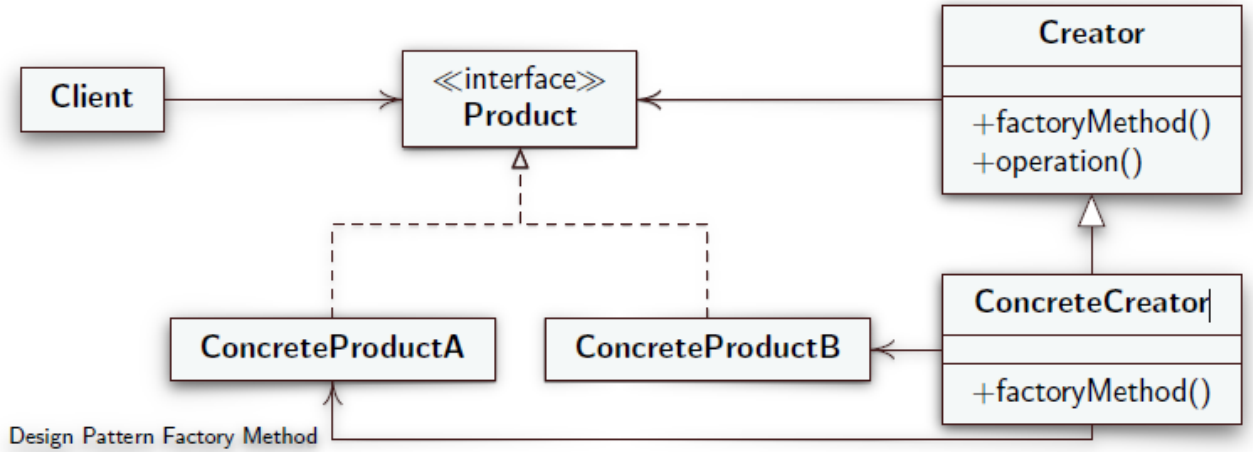
Scopo: Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi

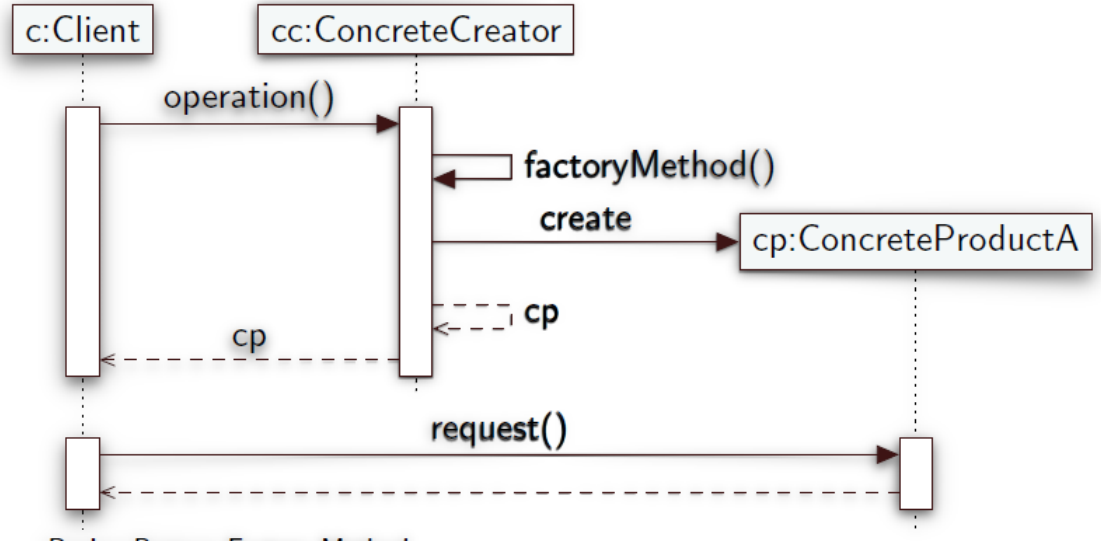
Esempio: Un framework usa classi astratte per definire e mantenere relazioni tra oggetti. Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare. Un metodo responsabile per l'istanziazione (detto factory, ovvero fabbricatore) incapsula la conoscenza su quale classe creare.

Soluzione:

- Product è l'interfaccia comune degli oggetti creati da factoryMethod()
- ConcreteProduct è un'implementazione di Product
- Creator dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Creator può avere un'implementazione si default del factoryMethod() che ritorna un certo ConcreteProduct
- ConcreteCreator implementa il factoryMethod(), o ne fa override, sceglie quale ConcreteProduct istanziare e ritorna tale istanza

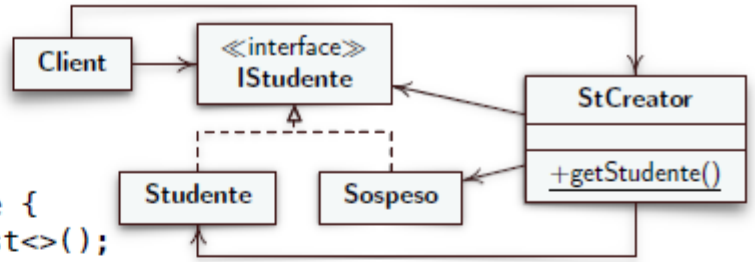
Il pattern **Factory Method**





Il pattern **Factory Method**

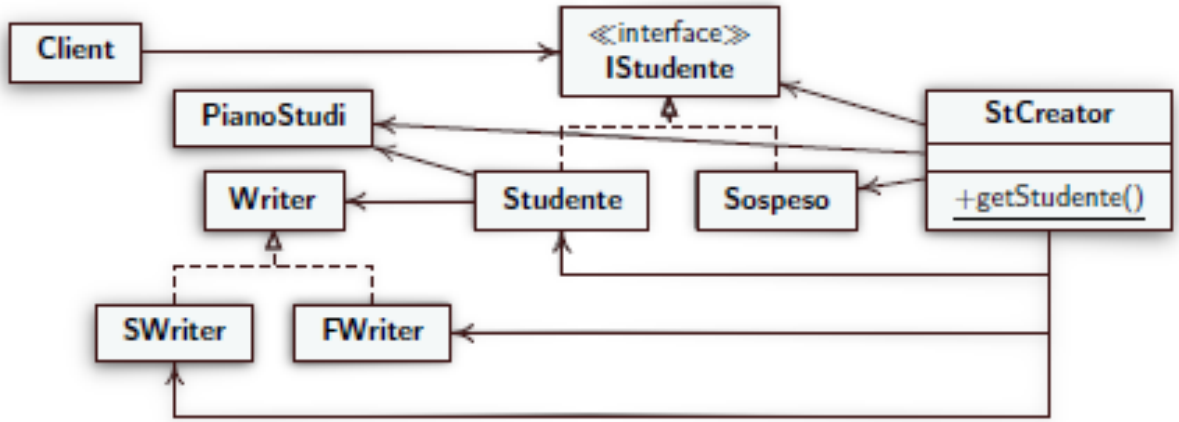
```
public interface IStudente {
  public void nuovoEsame(String m, int v);
  public float getMedia();
}

public class Studente implements IStudente {
  private List<Esame> esami = new ArrayList<>();
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
}

public class Sospeso implements IStudente {
  private float media;
  public Sospeso(float m) {
    media = m;
  }
  public void nuovoEsame(String m, int v) {
    System.out.println("Non e' possibile sostenere esami");
  }
  public float getMedia() {
    return media;
  }
}
```




```
public class StCreator {
  private static boolean a = true;
  public static
         IStudente getStudente() {
    if (a)
       return new Studente();
    return new Sospeso(0);
  }
}

public class Client {
  public void registra() {
    IStudente s =
          StCreator.getStudente();
    s.nuovoEsame("Maths", 8);
  }
}
```

Il pattern **Factory Method – Dependency injection**

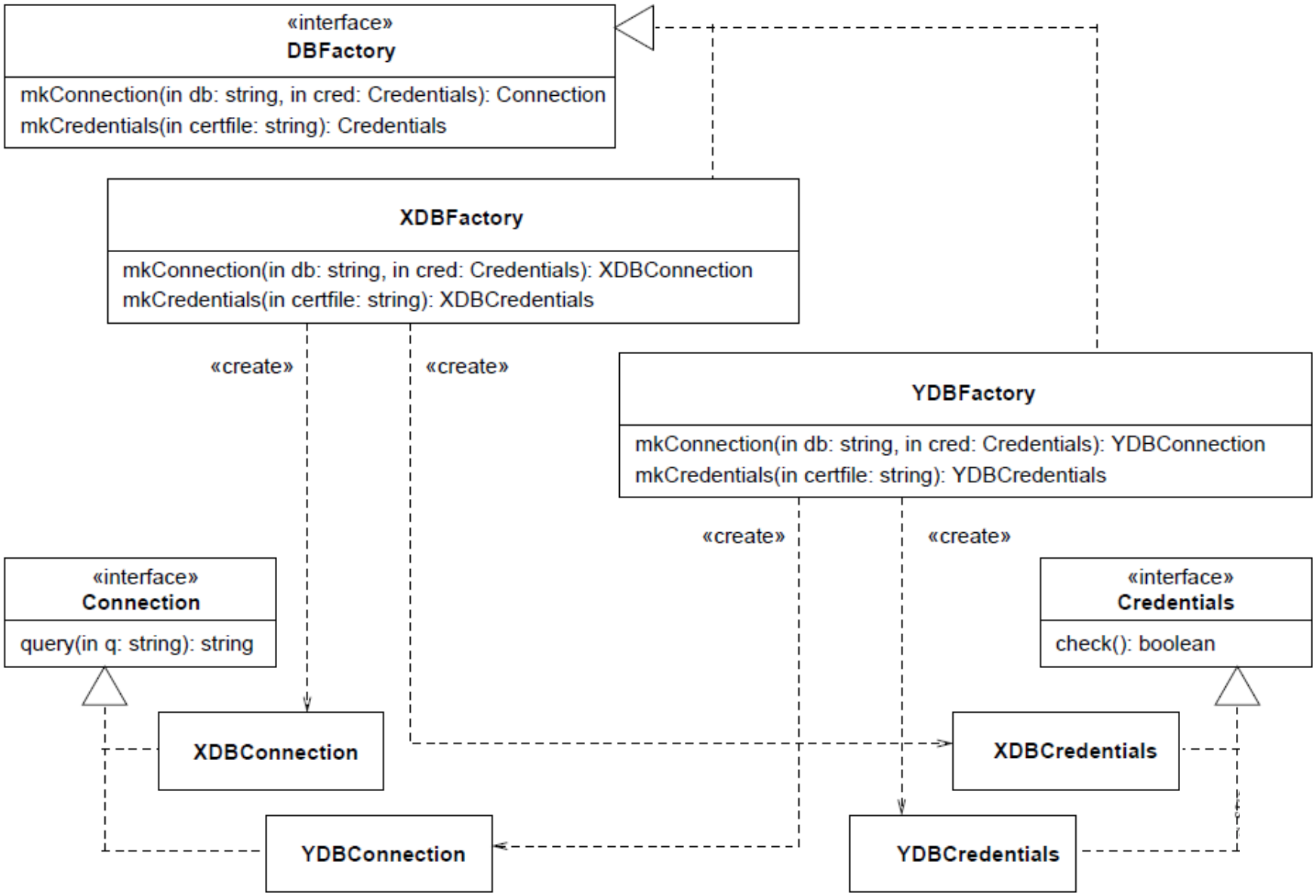
- Il design pattern Factory Method può essere usato per inserire le dipendenze (dependency injection) necessarie alle istanze di ConcreteProduct
- Tramite la Dependency Injection un oggetto (client) riceve altri oggetti da cui dipende, questi altri oggetti sono detti dipendenze
- La tecnica di Dependency Injection permette di separare la costruzione delle istanze dal loro uso
- Il client non crea l'istanza di cui ha bisogno
- Le dipendenze sono iniettate al client per mezzo di parametri nel suo costruttore. Questo permette di evitare complicazioni derivanti da metodi setter e da controlli per verificare che le dipendenze non siano null, di conseguenza il codice è più semplice
- L'oggetto che fa Dependency Injection si occupa di connettere (fa wiring di) varie istanze. In un unico posto vediamo le connessioni fra gli oggetti



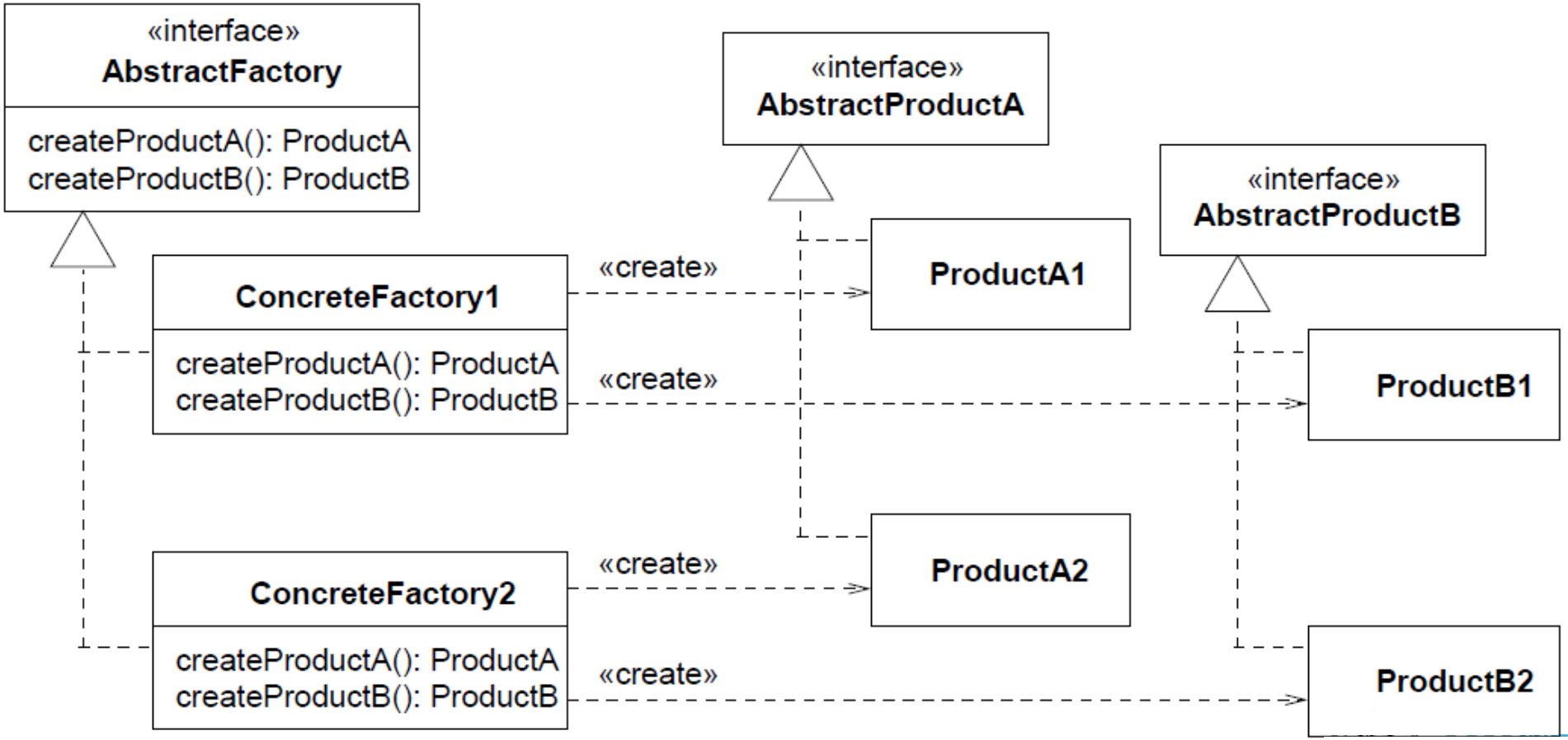
Il pattern **Abstract Factory**

- **scopo**: creare famiglie di oggetti interrelati in modo che i clienti non dipendano dalla loro implementazione.
- **esempio**: un'applicazione accede a un database usando le interfacce Connection e Credentials. Queste sono implementate da due database diversi, XDB e YDB, scelte dall'applicazione all'inizio dell'esecuzione.
- **soluzione**: si definisce un'interfaccia DBFactory per creare istanze di Connection e Credentials, realizzata dalle classi XDBFactory e YDBFactory, una per ciascuna delle librerie alternative.

Il pattern **Abstract Factory**

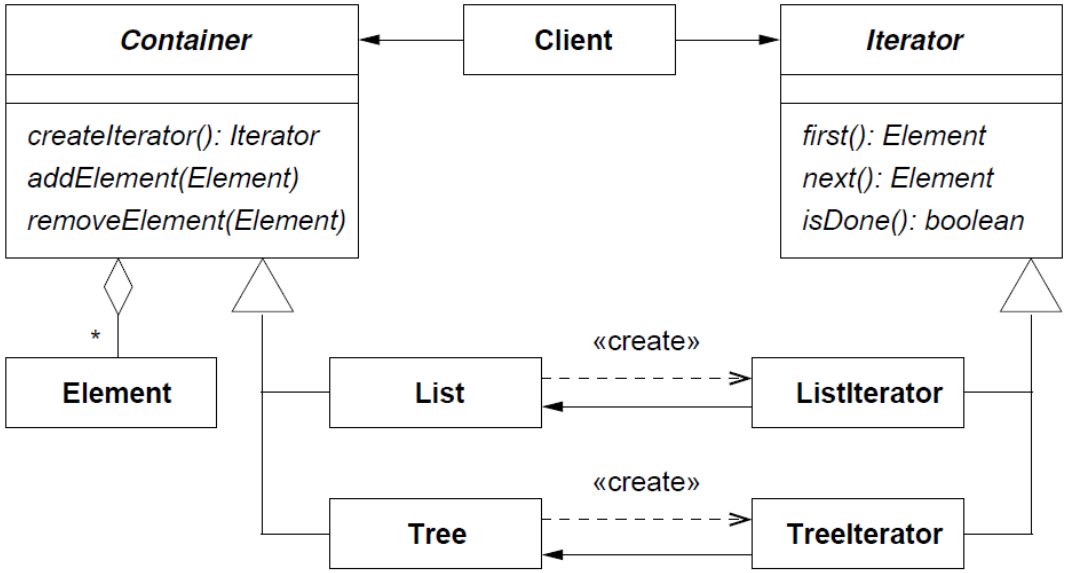


Il pattern **Abstract Factory**

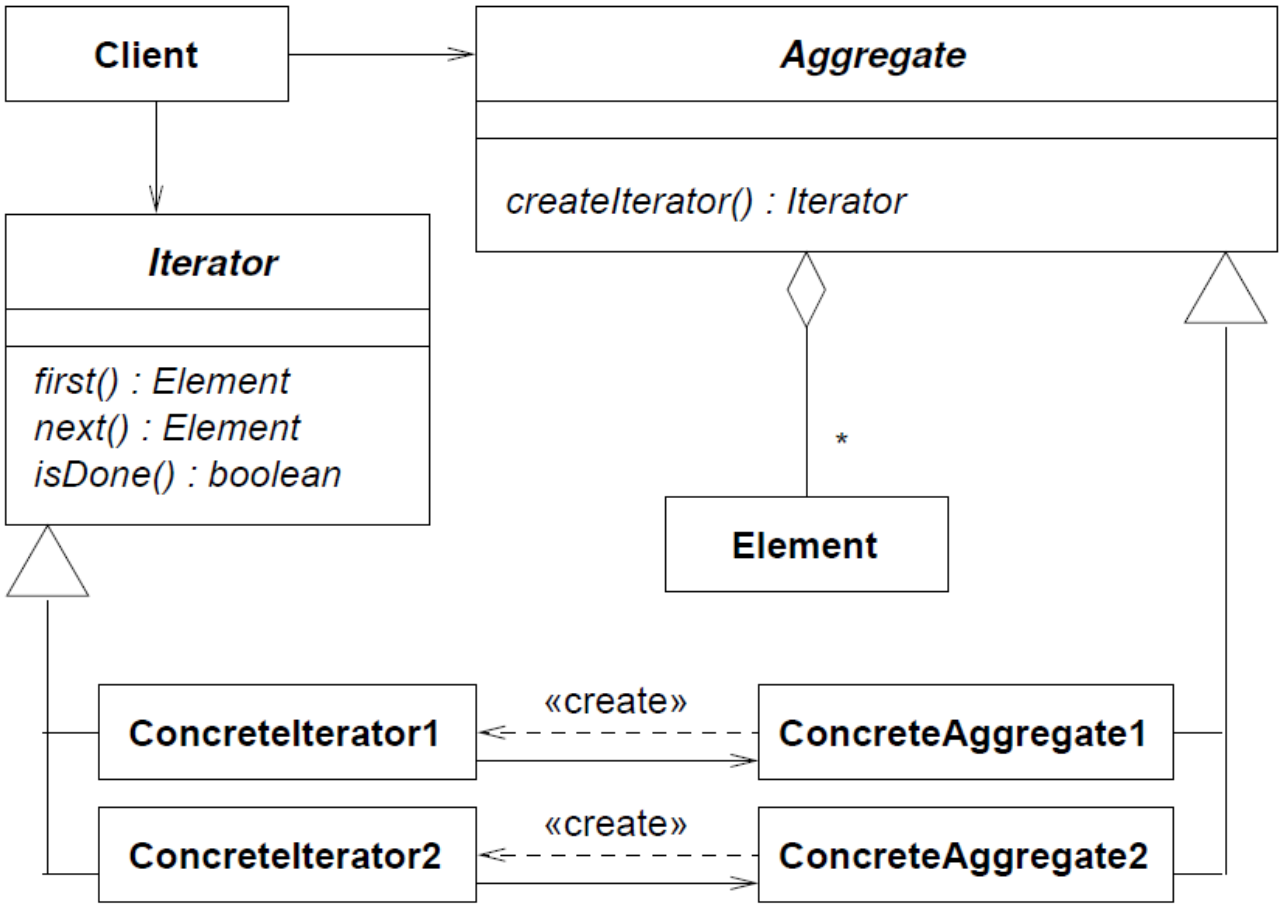


Il pattern **Iterator**

- **scopo**: accedere in sequenza agli elementi di una struttura dati senza dipendere dall'implementazione.
- **esempio**: Una raccolta di oggetti Element può essere implementata usando strutture dati diverse, per esempio liste (List) o alberi (Tree). Un'applicazione deve poter accedere agli elementi di questa raccolta indipendentemente da come viene implementata.
- **soluzione**: si definisce una classe astratta (o un'interfaccia) Iterator che offre le operazioni necessarie per accedere in sequenza agli elementi della raccolta, implementate da classi specifiche per ciascuna implementazione (ListIterator e TreeIterator). La classe astratta (o interfaccia) Container offre le operazioni per costruire la raccolta e per creare un iteratore, e viene implementata da List o Tree.

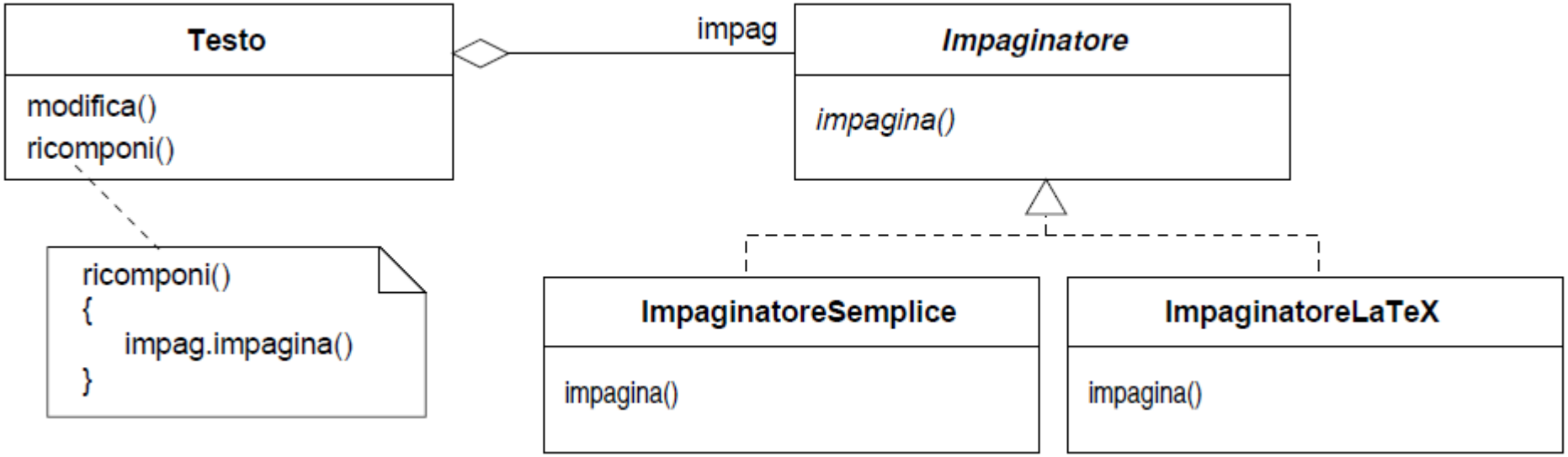


Il pattern **Iterator**



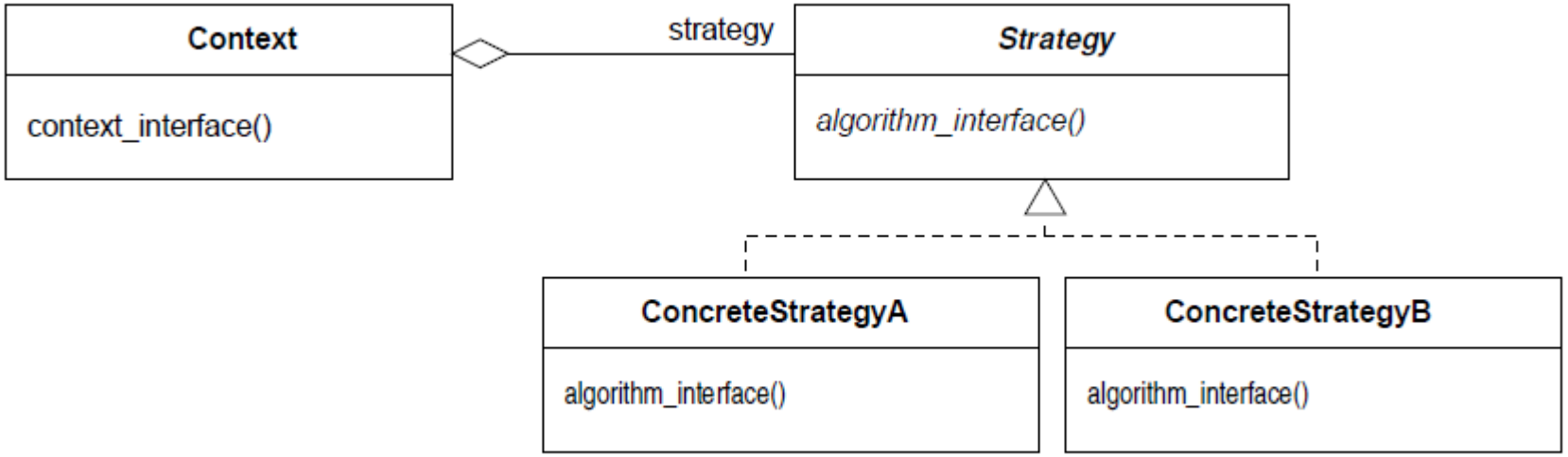
Il pattern **Strategy**

- **scopo**: definire una famiglia di algoritmi intercambiabili.
- **esempio**: un testo deve essere (re)impaginato dopo le modifiche usando algoritmi alternativi.
- **soluzione**: si definisce un'interfaccia comune per i diversi algoritmi, implementata da classi diverse.
- **Oss.**: un algoritmo di impaginazione deve riempire la pagina in modo uniforme, tenendo conto delle spaziature, delle dimensioni dei caratteri, delle illustrazioni etc.



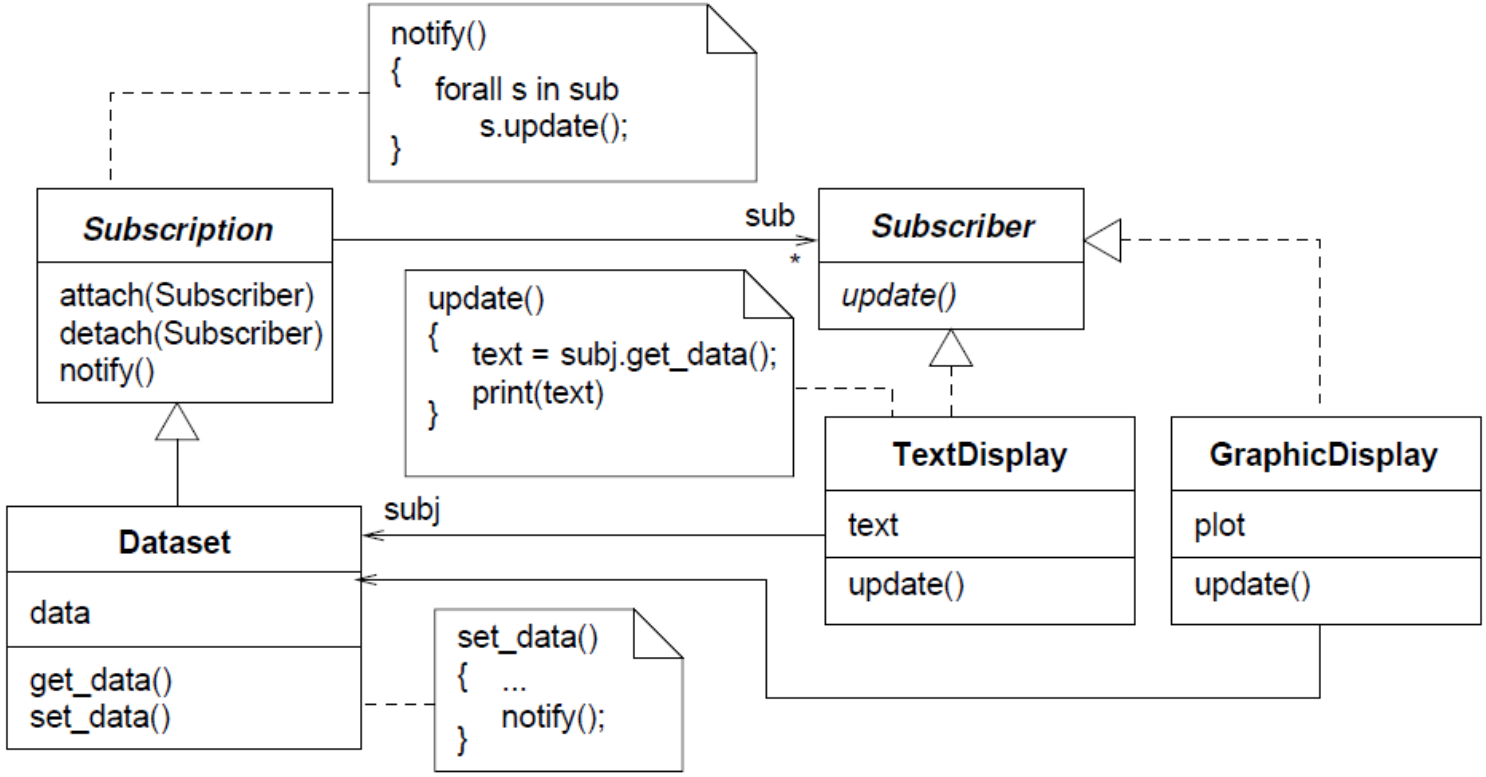
Il pattern **Strategy**

- **scopo**: definire una famiglia di algoritmi intercambiabili.
- **esempio**: un testo deve essere (re)impaginato dopo le modifiche usando algoritmi alternativi.
- **soluzione**: si definisce un'interfaccia comune per i diversi algoritmi, implementata da classi diverse.
- **Oss.**: un algoritmo di impaginazione deve riempire la pagina in modo uniforme, tenendo conto delle spaziature, delle dimensioni dei caratteri, delle illustrazioni etc.



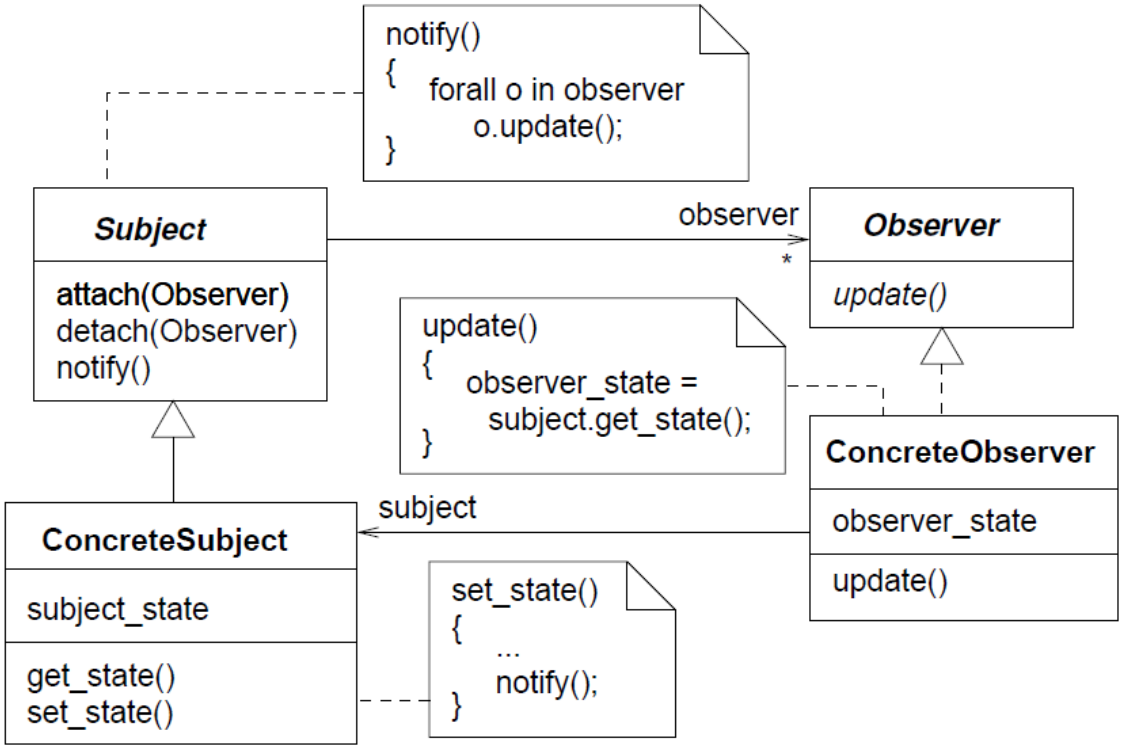
Il pattern **Observer (Publish-subscribe)**

- **scopo**: definire un sistema di dipendenze in modo che un oggetto possa notificare altri oggetti dei suoi cambiamenti di stato.
- **esempio**: un insieme di dati deve essere visualizzato in diversi modi.
- **soluzione**: si definisce un'interfaccia comune per aggiornare i sistemi di visualizzazione, ed un interfaccia per informare l'insieme dei dati che dei sistemi di visualizzazione devono essere notificati dei cambiamenti di stato.



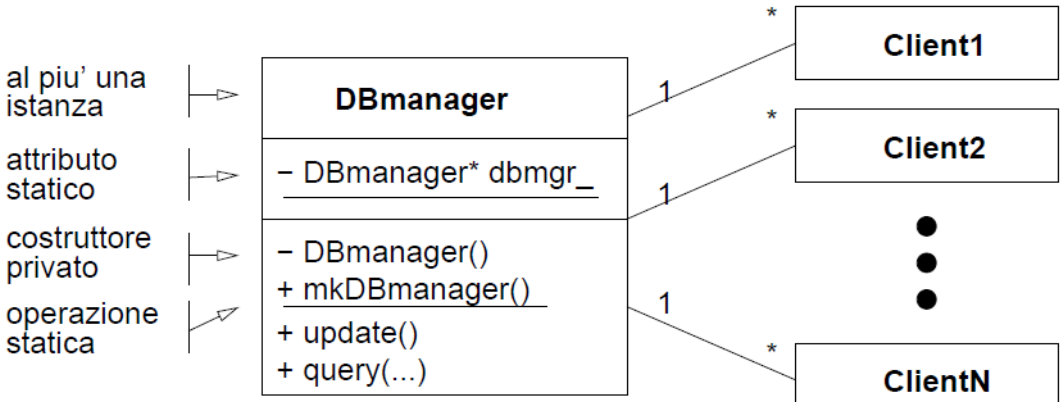
Il pattern **Observer (Publish-subscribe)**

- **scopo**: definire un sistema di dipendenze in modo che un oggetto possa notificare altri oggetti dei suoi cambiamenti di stato.
- **esempio**: un insieme di dati deve essere visualizzato in diversi modi.
- **soluzione**: si definisce un'interfaccia comune per aggiornare i sistemi di visualizzazione, ed un interfaccia per informare l'insieme dei dati che dei sistemi di visualizzazione devono essere notificati dei cambiamenti di stato.



Il pattern **Singleton**

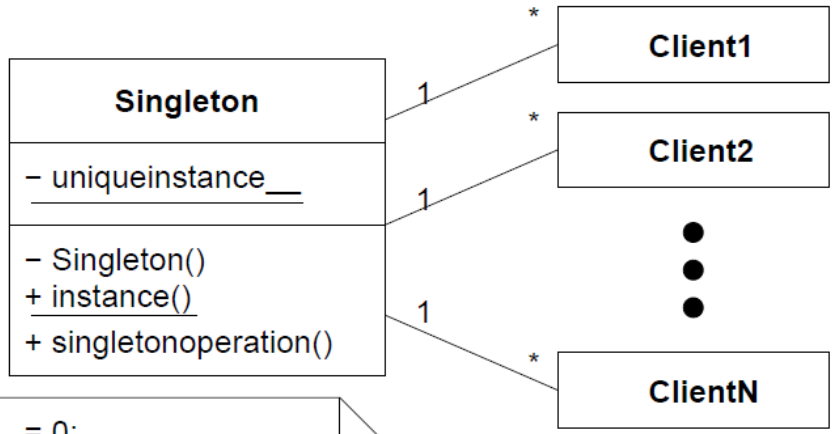
- **scopo**: garantire che nel sistema esista una sola istanza di una certa classe.
- **esempio**: un un certo numero di applicazioni deve accedere ad un gestore di un database.
- **soluzione**: si assegna la visibilità privata (o protetta) al costruttore della classe che implementa il gestore. Le applicazioni clienti ottengono un puntatore all'unica istanza per mezzo di un'operazione statica. Questa alloca un'istanza del gestore in memoria dinamica solo la prima volta che viene invocata.




```
DBmanager* DBmanager::dbmgr_ = 0;

DBmanager* DBmanager mkDBmanager()
{
    if (dbmgr_ == 0)
        dbmgr_ = new DBmanager;
    return dbmg_;
}
```

Il pattern **Singleton**

- **scopo**: garantire che nel sistema esista una sola istanza di una certa classe.
- **esempio**: un un certo numero di applicazioni deve accedere ad un gestore di un database.
- **soluzione**: si assegna la visibilità privata (o protetta) al costruttore della classe che implementa il gestore. Le applicazioni clienti ottengono un puntatore all'unica istanza per mezzo di un'operazione statica. Questa alloca un'istanza del gestore in memoria dinamica solo la prima volta che viene invocata.




```
uniqueinstance_ = 0;

instance()
{
    if (uniqueinstance_ == 0)
        uniqueinstance_ = new Singleton;
    return uniqueinstance_;
}
```

Il pattern **Object pool**

- Un object pool è un deposito di istanze già create, una istanza sarà estratta dal pool quando una classe client ne fa richiesta
- Il pool può crescere o può avere dimensioni fisse. Dimensioni fisse: se non ci sono oggetti disponibili al momento della richiesta, non ne creo di nuovi
- Il client restituisce al pool l'istanza usata quando non più utile
- Il design pattern Factory Method può implementare un object pool
- I client richiedono istanze, come visto per il Factory Method
- I client dovranno indicare quando l'istanza non è più in uso, quindi riusabile
- Lo stato dell'istanza da riusare potrebbe dover essere riscritto
- L'object pool dovrebbe essere unico: potrei usare un Singleton

```java
import java.util.ArrayList;
import java.util.List;

// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool {
    private List<Shape> pool = new ArrayList<>();

    // metodo factory che ritorna un oggetto prelevato dal pool
    public Shape getShape() {
        if (pool.size() > 0)
            return pool.remove(0);
        return new Circle();
    }

    // inserisce un oggetto nel pool
    public void releaseShape(Shape s) {
        pool.add(s);
    }
}
```

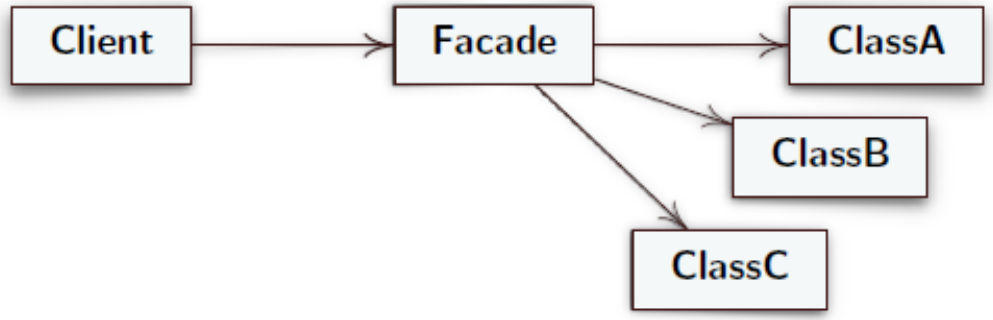
Il pattern **Facade**

- **scopo**: Fornire un'interfaccia unificata al posto di un insieme di interfacce in un sottosistema (consistente di un insieme di classi). Definire un'interfaccia di alto livello (semplificata) che rende il sottosistema più facile da usare

- **Problema**:
  - Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complesso
  - Può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi
  - Si vogliono ridurre le comunicazioni e le dipendenze dirette fra i client ed il sottosistema

- **Soluzione**:
  - Facade fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate. Facade invoca i metodi degli oggetti che nasconde
  - Client interagisce solo con l'oggetto Facade

Il pattern **Facade**

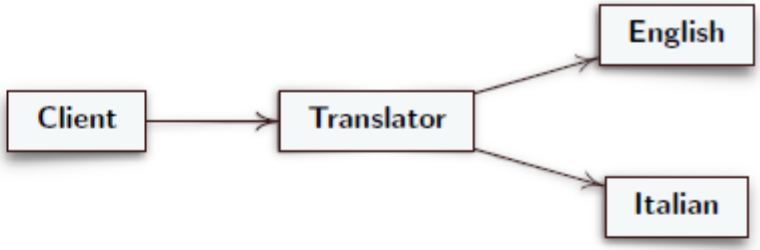


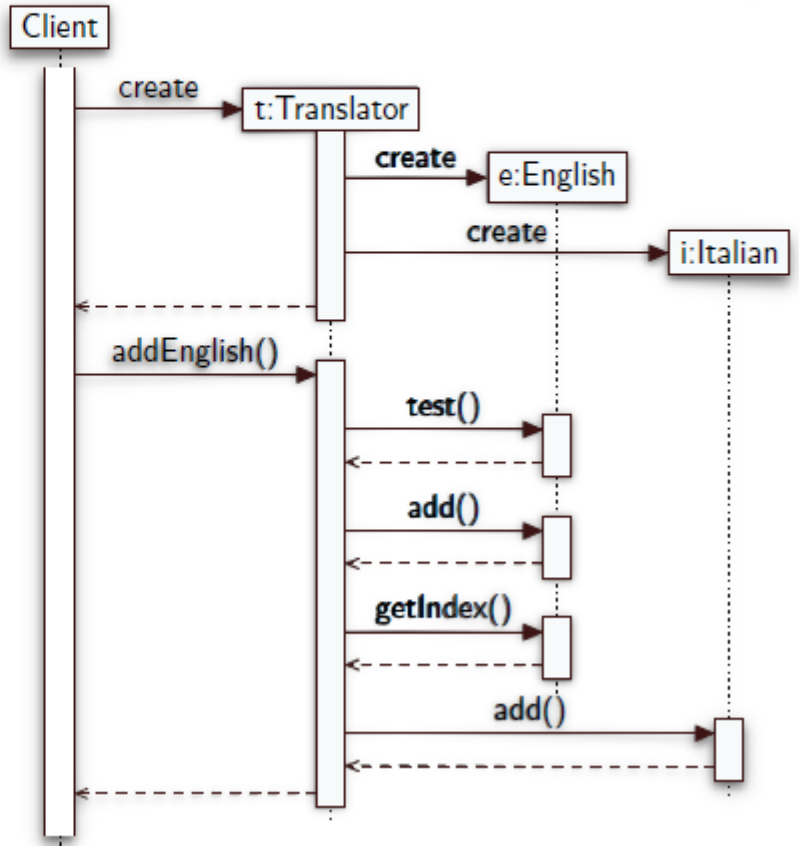
- Conseguenze
  - Nasconde ai client l'implementazione del sottosistema
  - Promuove l'accoppiamento debole tra sottosistema e client
  - Riduce le dipendenze di compilazione in sistemi grandi. Se si cambia una classe del sottosistema, si può ricompilare la parte di sottosistema fino al facade, quindi non i vari client
  - Non previene l'uso di client più complessi, quando occorre, che accedono ad oggetti del sottosistema

- Implementazione
  - Per rendere gli oggetti del sottosistema non accessibili al client le corrispondenti classi possono essere annidate dentro la classe Facade

Il pattern **Facade**

```
public class Client {
   public static void main(String args[]) {
      Translator t = new Translator();
      t.addEnglish("Hello");
      t.multiPrinting();
   }
}

public class Translator { // Ruolo Facade
   private English e = new English();
   private Italian i = new Italian();

   public void addEnglish(String s) {
      if (e.test(s)) {
         e.add(s);
         i.add(e.getIndex(s));
      }
   }

   public void multiPrinting() {
      System.out.print("Italiano:  ");
      i.printText();
      System.out.print("English:   ");
      e.printText();
   }
}
```





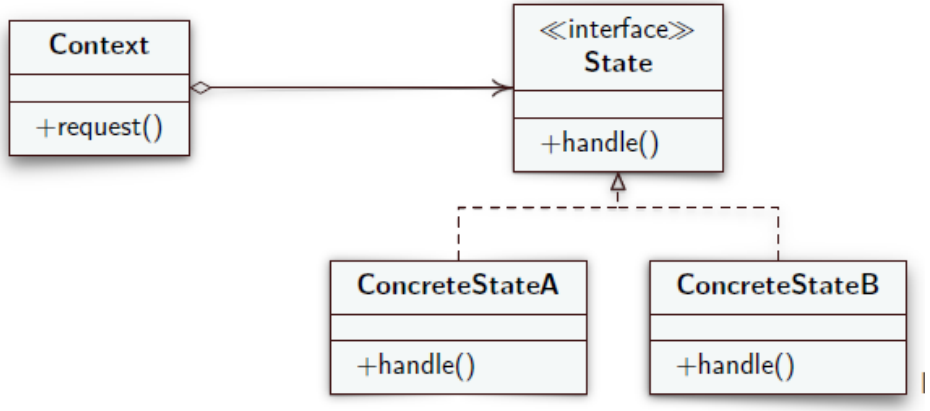
Il pattern **State**

- **Intento**: Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Far sembrare che l'oggetto abbia cambiato la sua classe

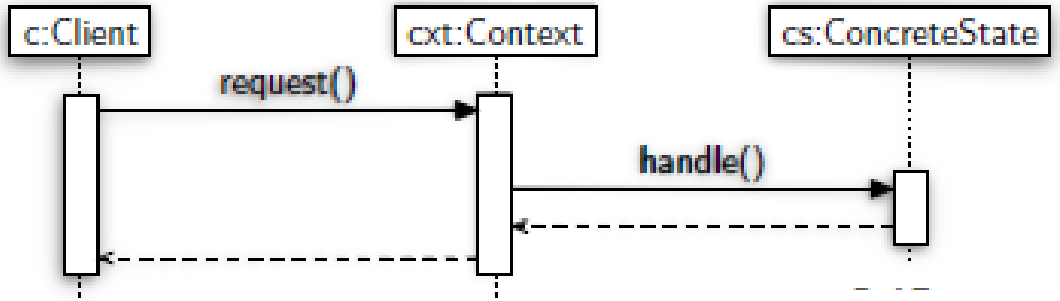
- **Problema**
  - Il comportamento di un oggetto dipende dal suo stato e il comportamento deve cambiare a run-time in base al suo stato
  - Le operazioni da svolgere hanno alcuni grandi rami condizionali che dipendono dallo stato
  - Lo stato è spesso rappresentato dal valore di una o più variabili enumerative costanti
  - Spesso varie operazioni contengono la stessa struttura condizionale

- **Soluzione**
  - Inserire ogni ramo condizionale in una classe separata
  - Context definisce l'interfaccia che interessa ai client, e mantiene un'istanza di una classe ConcreteState che definisce lo stato corrente
  - State definisce un'interfaccia che incapsula il comportamento associato ad un particolare stato del Context
  - ConcreteState sono le sottoclassi che implementano ciascuna il comportamento associato ad uno stato del Context

Il pattern **State**



- Collaborazioni
  - Il Context passa le richieste dipendenti da un certo stato all'oggetto ConcreteState corrente
  - Un Context può passare se stesso come argomento all'oggetto ConcreteState per farlo accedere al contesto se necessario
  - Il Context è l'interfaccia per le classi client
  - Il Context o i ConcreteState decidono quale stato è il successivo ed in quali circostanze

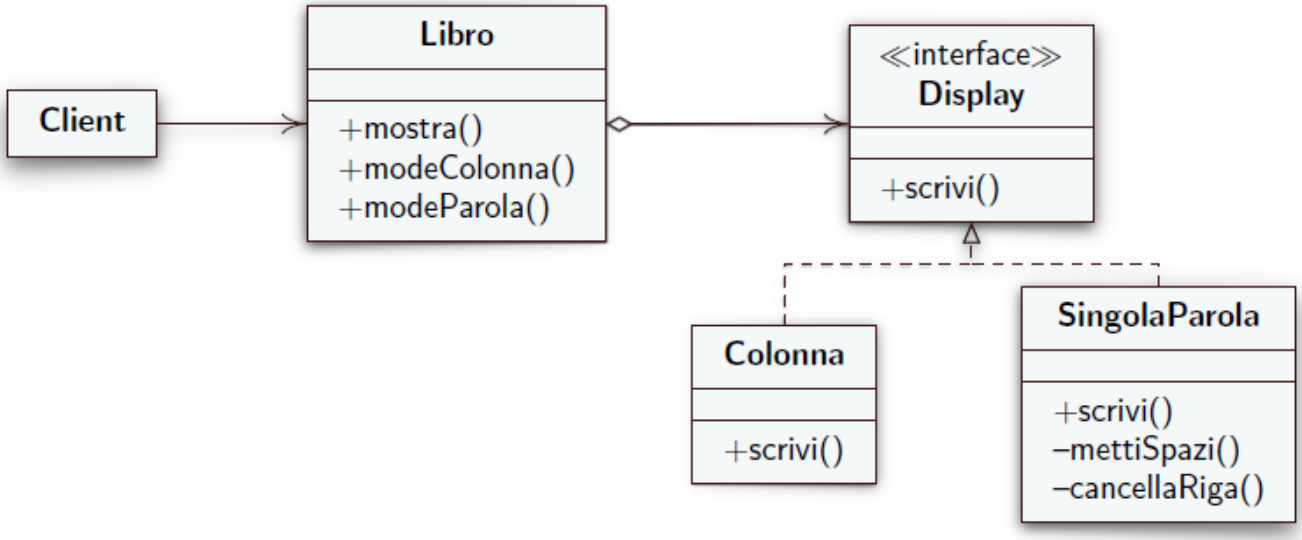


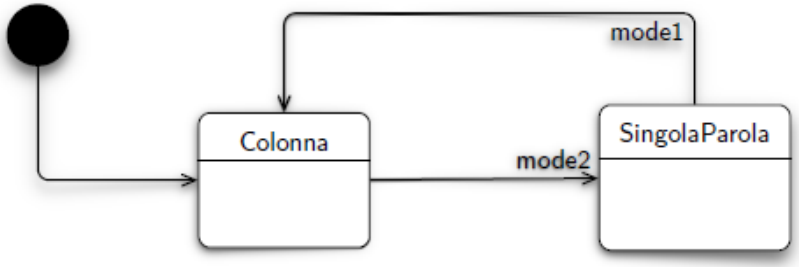
Il pattern **State**

- Conseguenze
  - Il comportamento associato a uno stato è localizzato in una sola classe (ConcreteState) e si partiziona il comportamento di stati differenti. Per tale motivo, si posso aggiungere nuovi stati e transizioni facilmente, creando nuove sottoclassi. Incapsulare le azioni di uno stato in una classe impone una struttura e rende più chiaro lo scopo del codice
  - La logica che gestisce il cambiamento di stato è separata dai vari comportamenti ed è in una sola classe (Context), anziché (con istruzioni if o switch) sulla classe che implementa i comportamenti. Tale separazione aiuta a evitare stati inconsistenti, poiché i cambiamenti di stato vengono decisi da una sola classe e non da tante
  - Il numero di classi totale è maggiore, le classi sono più semplici

Il pattern **State**

- Esempio
  - Si vogliono avere vari modi per scrivere il testo di un libro su un display: in modalità una colonna, due colonne, o una singola parola per volta





Il pattern **State**

```
public class Libro { // Context
    private String testo = "Darwin's _Origin of Species_ persuaded the world that the "
        + "difference between different species of animals and plants is not the fixed "
        + "immutable difference that it appears to be.";
    private List<String> lista = Arrays.asList(testo.split("[\\s+]+"));
    private Display mode = new Colonna();
    public void mostra() {
        mode.scrivi(lista);
    }
    public void modeColonna() {
        mode = new Colonna();
    }
    public void modeParola() {
        mode = new SingolaParola();
    }
}
```

```
public interface Display { // State
 public void scrivi(List<String> testo);
}
```

```
public class Colonna implements Display { // ConcreteState
    private final int numCar = 38;
    private final int numRighe = 12;
    public void scrivi(List<String> testo) {
        int riga = 0;
        int col = 0;
        for (String p : testo) {
            if (col + p.length() > numCar) {
                System.out.println();
                riga++;
                col = 0;
            }
            if (riga == numRighe) break;
            System.out.print(p + " ");
            col += p.length() + 1;
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        Libro l = new Libro();
        l.mostra();
        l.modeParola();
        l.mostra();
    }
}
```

Il pattern **State**

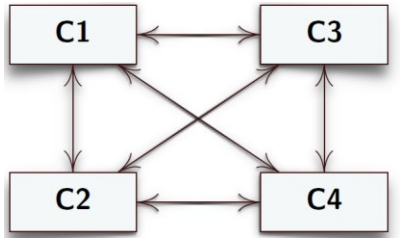
```
public class SingolaParola implements Display { // ConcreteState
    private int maxLung;
    public void scrivi(List<String> testo) {
        System.out.println();
        trovaMaxLung(testo);
        for (String p : testo) {
            int numSpazi = (maxLung - p.length()) / 2;
            mettiSpazi(numSpazi);
            System.out.print(p);
            if (p.length() % 2 == 1) numSpazi++;
            mettiSpazi(numSpazi);
            aspetta();
            cancellaRiga();
        }
        System.out.println();
    }
    private void mettiSpazi(int n) {
        System.out.print(" ".repeat(n));
    }
    private void cancellaRiga() {
        System.out.print("\b".repeat(maxLung));
    }
    private void trovaMaxLung(List<String> testo) {
        for (String p : testo) if (maxLung < p.length()) maxLung = p.length();
    }
    private static void aspetta() {
        try {
            Thread.sleep(300);
        } catch (InterruptedException e) { }
    }
}
```

Il pattern **State**

```
public class LibroPrimaDiState {
   private String testo = " ... ";
   private List<String> lista = Arrays.asList(testo.split("[\\s+]+"));
   private int mode = 2;

   public void mostra() {
      switch (mode) {
      case 1:

         // vedi metodo scrivi della classe SingolaParola

         break;
      case 2:

         // vedi metodo scrivi della classe Colonna

         break;
      }
   }

   public void setMode(int x) {
      mode = x;
   }
}
```

Il pattern **Mediator**

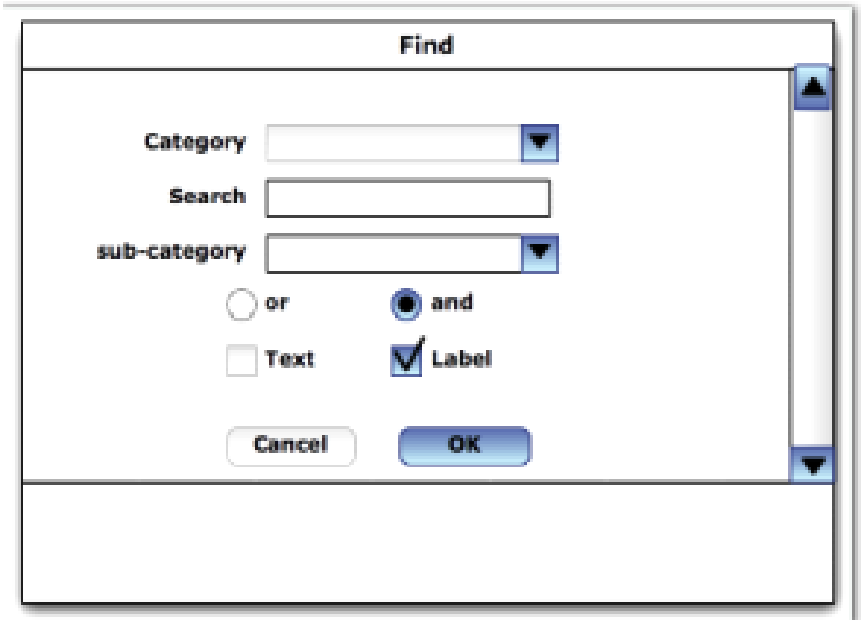
- **Intento**: Definire un oggetto che incapsula come un gruppo di oggetti interagisce. Il Mediator promuove il lasco accoppiamento fra oggetti poiché evita che essi interagiscano direttamente, e permette di modificare le loro interazioni indipendentemente da essi

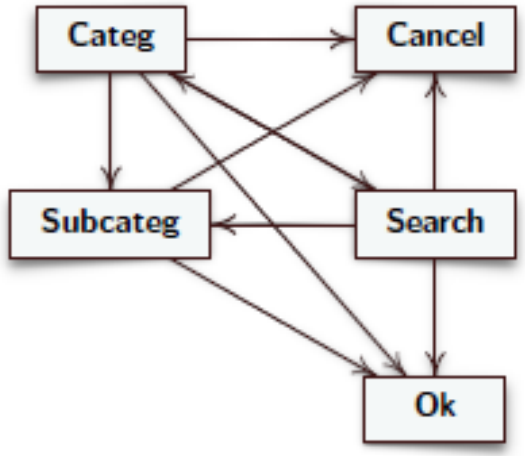


- **Motivazione**
  - La distribuzione di responsabilità fra vari oggetti può risultare in molte connessioni fra oggetti, nel caso peggiore un oggetto conosce tutti gli altri
  - Molte connessioni rendono un oggetto dipendente da altri e l'intero sistema si comporta come se fosse monolitico. Inoltre potrebbe essere difficile cambiare il comportamento del sistema poiché il comportamento è distribuito fra oggetti
  - Si possono evitare questi problemi incapsulando il comportamento collettivo in un oggetto mediatore separato. Il mediatore serve da intermediario ed evita che gli oggetti dipendano fra loro

Il pattern **Mediator**

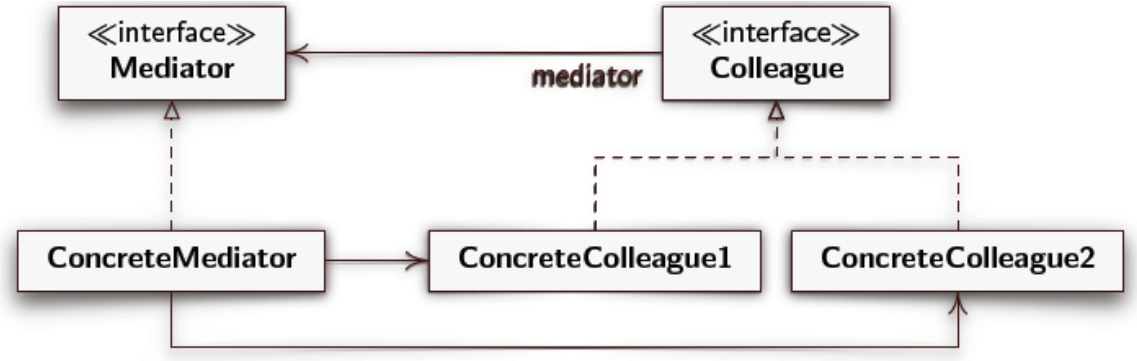
- Per la finestra di ricerca mostrata
  - Ogni elemento visualizzato (testo, lista, bottone, etc.) è controllato da una corrispondente classe
  - Ciascuna classe deve comunicare il suo stato alle altre per far aggiornare la visualizzazione
  - Ciascuna classe (senza un Mediator) chiamerà i metodi di tutte le altre classi, e quindi ciascuna classe è dipendente dalle altre





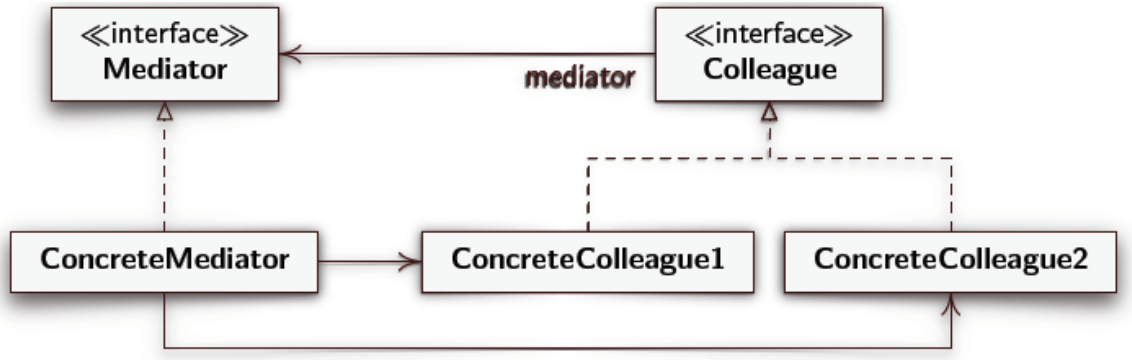
Il pattern **Mediator**

- Applicabilità
  - Usare il Mediator quando
    - Un insieme di oggetti comunicano in modo ben definito ma complesso. Le interdipendenze che ne risultano sono non strutturate e difficili da comprendere
    - Riusare un oggetto è difficile poiché esso comunica con tanti altri oggetti
    - Un comportamento che è distribuito fra tante classi dovrebbe essere modificabile senza dover ricorrere a sottoclassi



Il pattern **Mediator**

- Soluzione
  - Isolare le comunicazioni (complesse) tra oggetti dipendenti reando una classe separata per esse
  - Mediator definisce una interfaccia (punto di incontro) per gli oggetti connessi Colleague
  - ConcreteMediator implementa il comportamento cooperativo e coordina oggetti Colleague
  - Ogni Colleague conosce il Mediator, e comunica con il Mediator quando avrebbe comunicato con un altro Colleague
  - I ConcreteColleague mandano richieste, e ricevono richieste, a un oggetto Mediator. Il Mediator implementa il comportamento cooperativo inoltrando le richieste a opportuni ConcreteColleague



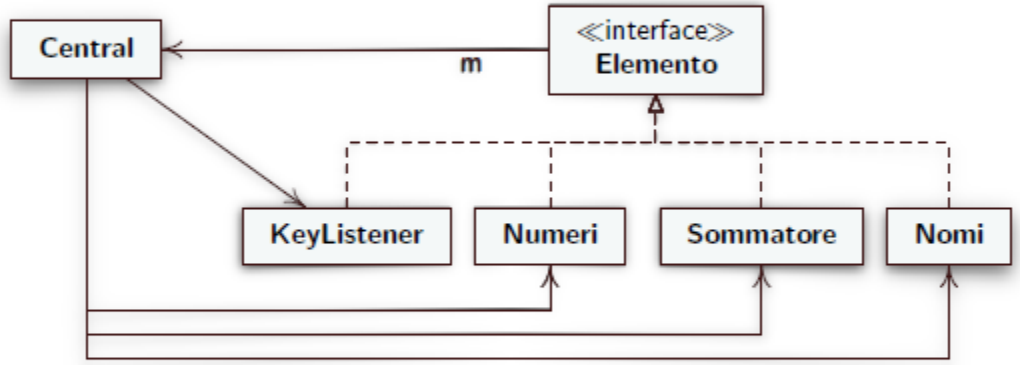
Il pattern **Mediator**

- Conseguenze
  - La maggior parte della complessità che risulta nella gestione delle dipendenze è spostata dagli oggetti cooperanti al Mediator.
    Questo rende gli oggetti più facili da implementare e mantenere
  - Le classi Colleague sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con il codice che gestisce le dipendenze
  - Il codice del Mediator non è in genere riusabile poiché la gestione delle dipendenze implementata è solo per una specifica applicazione

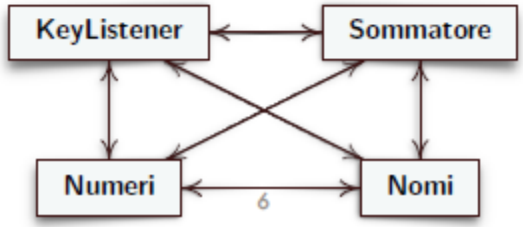
Il pattern **Mediator**

- esempio

  - Si legge un dato dalla tastiera e si compiono delle operazioni, su numeri o su stringa in base al dato letto
  - La classe `KeyListener` legge da tastiera un dato e ritorna il valore letto a `Central` (un Mediator), quest'ultima chiama i metodi dei ConcreteCollegue `Numeri`, `Sommatore`, `Nomi`, in base al tipo di dato letto

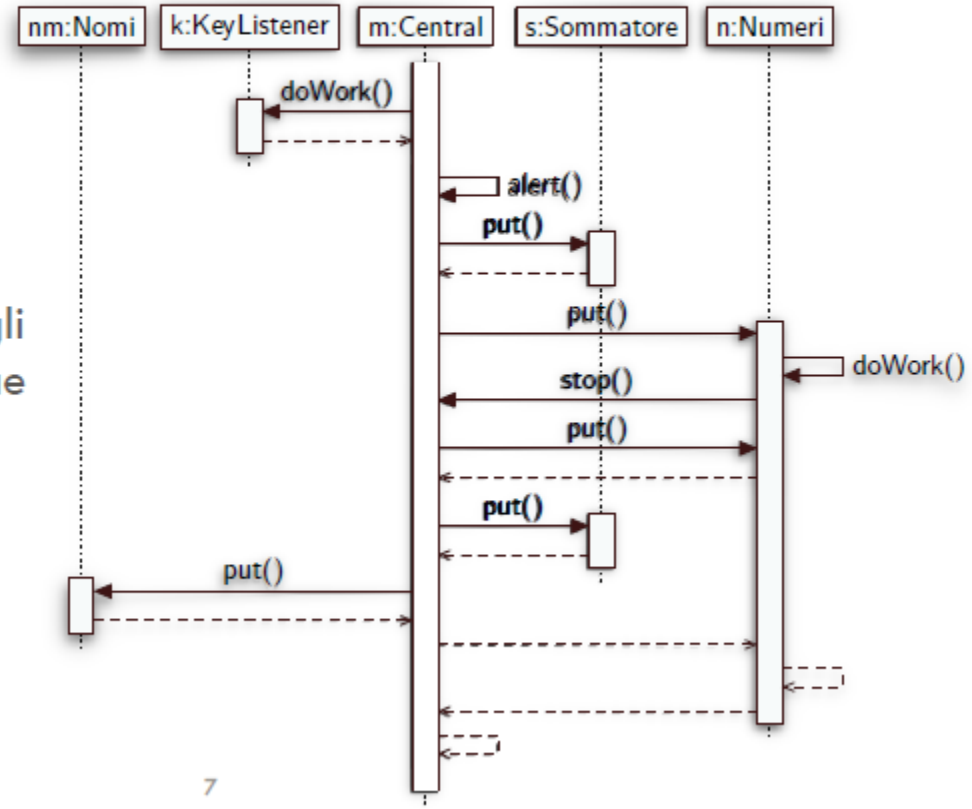


  - Nel caso non si adottasse il Mediator, le varie classi si chiamerebbero fra loro



Il pattern **Mediator**

- esempio
  - Il Mediator `Central` avvia la lettura da tastiera tramite il metodo `doWork()` di `KeyListener` e ottiene da esso il valore letto, quindi `Central` chiama `put()` sugli oggetti interessati al valore letto
  - Quando un oggetto ConcreteColleague riconosce una condizione di arresto, chiama `stop()` su `Central`, che avvisa gli altri ConcreteColleague
  - In figura si mostra il caso in cui `Numeri` chiama `stop()` su `Central`



Il pattern **Chain of Responsability**

- **Intento**: evitare di accoppiare il mandante di una richiesta col suo ricevente, dando così a più di un oggetto la possibilità di gestire la richiesta. Concatena gli oggetti riceventi e passa la richiesta lungo la catena fino a che un oggetto la gestisce

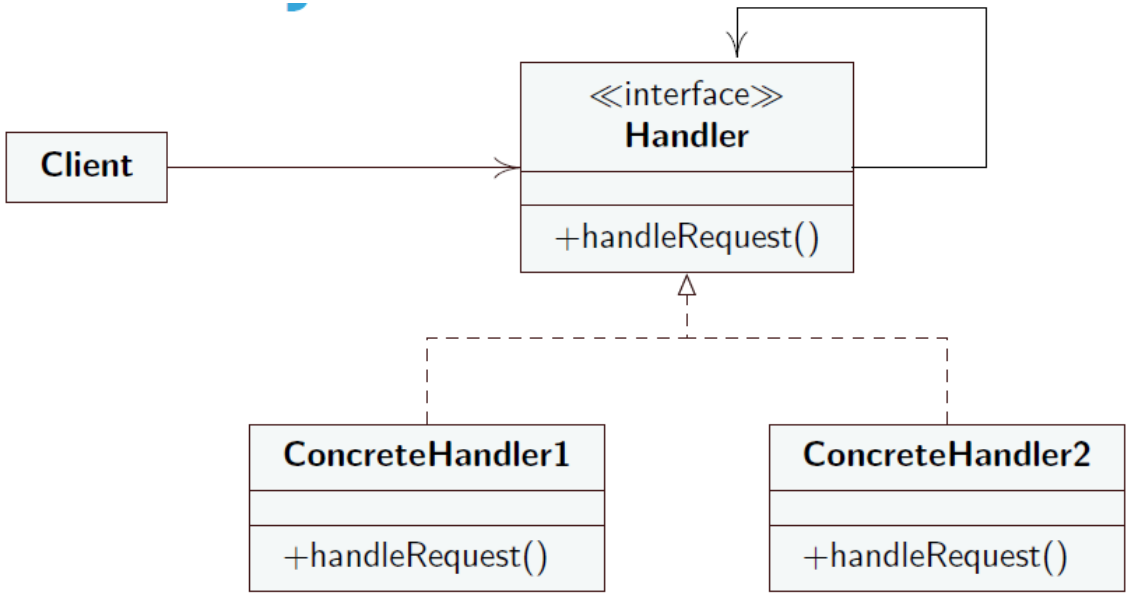
- **Motivazione**:
  - Consideriamo un'interfaccia utente in cui l'utente può chiedere aiuto su parti dell'interfaccia. Per es. un bottone può fornire informazioni di aiuto. Se non esiste un'informazione specifica allora si dovrebbe fornire il messaggio di aiuto del contesto più vicino (per es. la finestra)

  - Problema: l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto che effettua la richiesta iniziale

  - Disaccoppiare mandante e ricevente

Il pattern **Chain of Responsability**

- Struttura



- Partecipanti
  - Handler definisce l'interfaccia per gestire le richieste
  - ConcreteHandler gestisce la richiesta di cui è responsabile, può accedere al suo successore, inoltra la richiesta se non può gestirla
  - Client inizia effettuando la richiesta a un ConcreteHandler

- La richiesta si propaga lungo la catena finché un ConcreteHandler la gestisce

Il pattern **Chain of Responsability**

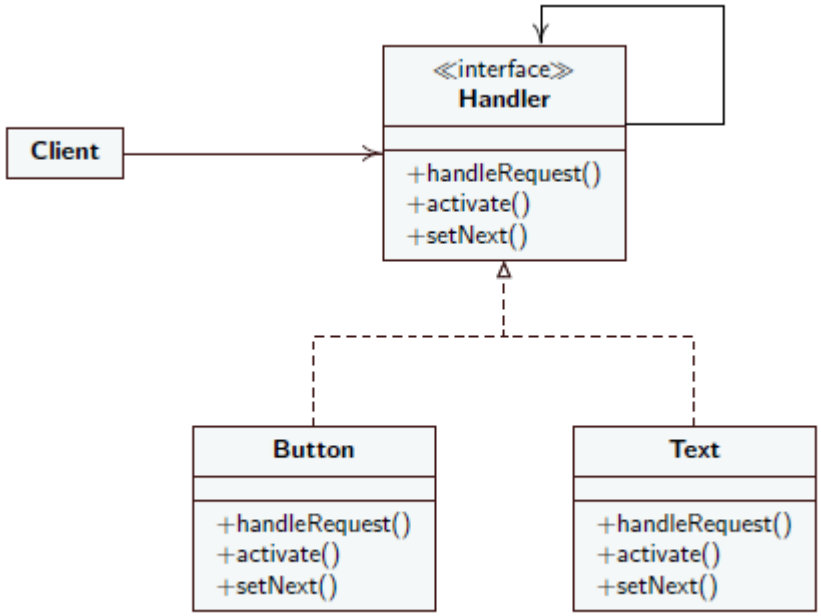
- Conseguenze
  - Riduce l'accoppiamento: chi effettua una richiesta non conosce il ricevente
  - Un oggetto della catena non deve conoscere la struttura della catena
  - Gli oggetti handler mantengono solo il riferimento al successore (non a una lista di oggetti)
  - Si aggiunge flessibilità nella distribuzione di responsabilità agli oggetti. Si può cambiare o aggiungere responsabilità nella gestione di una richiesta cambiando la catena a runtime
  - Non vi è garanzia che una richiesta sia gestita, poiché non c'è un ricevente esplicito, la richiesta potrebbe arrivare alla fine della catena senza essere gestita

Il pattern **Chain of Responsability**

- 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 (il design pattern Composite è usato insieme a Chain)
  - Ciascun tipo di richiesta può corrispondere a un'operazione, in questo caso il set di richieste è definito dall'Handler, oppure
  - Ciascun tipo di richiesta corrisponde a un codice e solo un'operazione è definita dall'Handler. Il set di richieste non è fissato: richiedenti e riceventi prendono accordi sulla codifica delle richieste; si hanno più controlli a runtime

Il pattern **Chain of Responsability**

Button e Text sono ConcreteHandler




```java
public interface Handler {
    public void handleRequest();
    public void activate();
    public void setNext(Handler next);
}

public class Text implements Handler {
    private Handler next;
    public Text(Handler h) {
        next = h;
    }
    @Override public void handleRequest() {
        if (next != null) next.handleRequest();
        else System.out.println("Text: assistenza");
    }
    @Override public void activate() {
        System.out.println("sono un frammento di testo");
    }
    @Override public void setNext(Handler next) {
        this.next = next;
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Text t = new Text(null);
        Button b = new Button(t);
        t.activate();
        b.activate();
        b.handleRequest();
        b.handleRequest();
    }
}
```

Il pattern **Protoype**

- **Intento**: specificare i tipi di oggetti da creare usando una istanza di un prototipo, e creare nuovi oggetti copiando questo prototipo

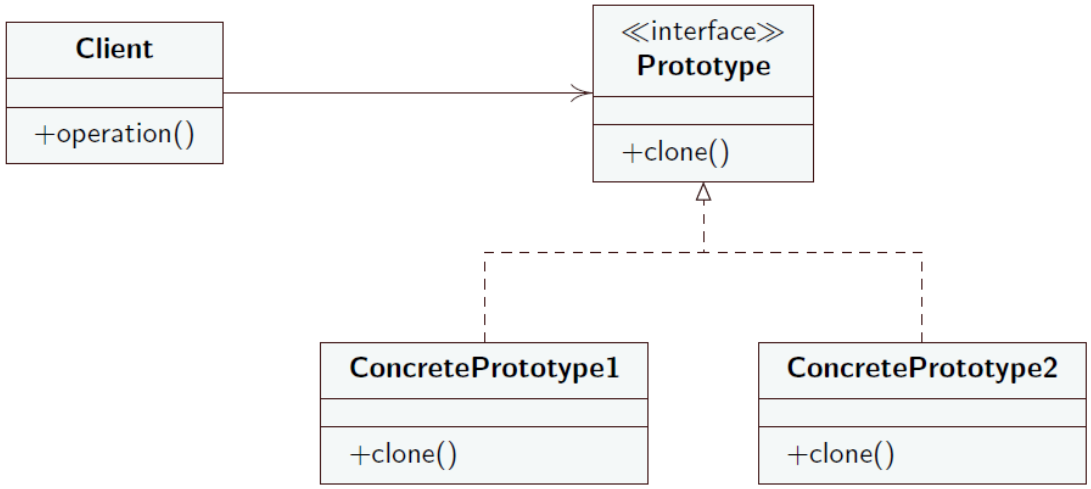
- **Motivazione**
  - Durante la creazione di un nuovo oggetto può essere necessario avere uno stato iniziale che è simile a quello di un altro oggetto esistente, e che quest'ultimo ha ottenuto per mezzo di operazioni eseguite su esso

  - Quando si crea un oggetto bisogna indicare una specifica classe, e così la classe che chiede l'istanza è strettamente accoppiata con la classe da istanziare

  - Lo stretto accoppiamento fra classi da istanziare e chi istanzia rende difficile l'aggiunta di nuovi tipi, per es. nel Factory Method bisogna creare una gerarchia di Creator per aggiungere nuovi tipi

Il pattern **Protoype**

- **Struttura**:



- Partecipanti
  - Prototype definisce l'interfaccia per le operazioni comuni e dichiara un metodo per clonare se stessa
  - ConcretePrototype è una classe che implementa l'operazione di clonazione
  - Client è una classe che crea nuovi oggetti tramite clonazione dell'istanza prototipo, quindi senza indicare la classe esplicitamente. Client conosce solo l'interfaccia Prototype

Il pattern **Prototype**
Conseguenze:

- Il client non conosce la classe usata, poiché il client usa l'interfaccia comune, quindi il numero di dipendenze del client è ridotto
- Il client può usare classi specifiche (sottoclassi) pur non conoscendone il nome
- Si può registrare un nuovo prototipo (o rimuovere un prototipo) a runtime
- Si possono specificare nuovi oggetti che tengono valori diversi, senza essere obbligati a creare nuove classi
- Un nuovo prototipo può essere usato per tenere una struttura diversa di oggetti (composizione). Gli oggetti della composizione sono attributi del prototipo, come in un Composite
- Pur avendo tante classi diverse da poter usare (i ConcretePrototype), non si ha necessità di implementare una gerarchia di Creator (come invece avviene per il Factory Method), in quanto un nuovo oggetto si ottiene tramite clonazione
- E' possibile creare una configurazione (un prototipo) dinamicamente

Il pattern **Protoype**

Implementazione:

- L'uso di un manager di prototipi (che è un registro di istanze) permette ai client di immagazzinare e recuperare i prototipi, prima di clonarli

- L'implementazione dell'operazione di clonazione può essere la parte più difficile quando le strutture contengono riferimenti circolari

    - Sebbene un'operazione di clonazione potrebbe essere fornita dal linguaggio di programmazione, l'operazione di clonazione potrebbe effettuare una copia superficiale (shallow copy) anziché una copia profonda (deep copy)

    - In una shallow copy i riferimenti tenuti negli attributi dell'oggetto da clonare sarebbero condivisi fra originale e clone, e questo spesso non è sufficiente, quindi occorre una deep copy

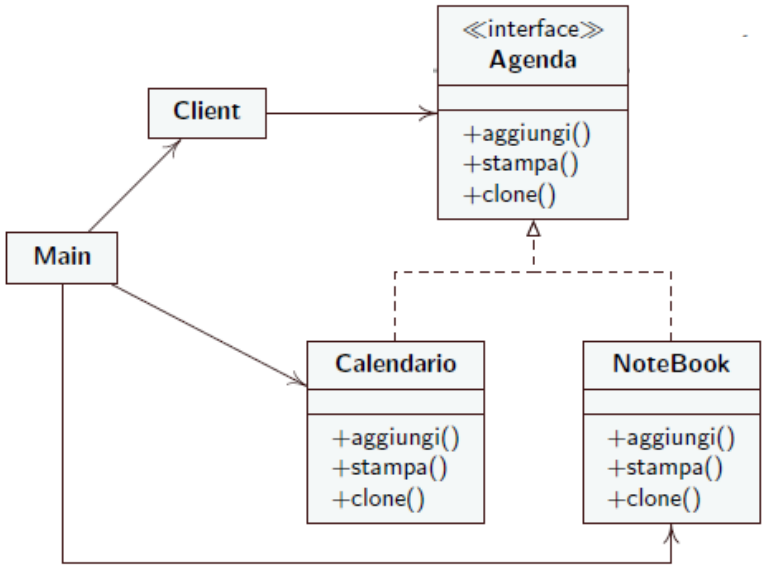
- I cloni potrebbero necessitare di essere inizializzati: la classe prototipo potrebbe definire operazioni per impostare dati

Il pattern **Protoype**

Esempio:

- Si vuol tenere un'agenda di impegni, varie voci possono essere aggiunte all'agenda
- Lo stato dell'agenda è l'insieme di impegni presi
- Una nuova istanza dell'agenda deve poter avere accesso agli impegni precedentemente inseriti
- In modo simile, si vogliono inserire note in un quaderno e accedere alle note precedenti quando si ha una nuova istanza di quaderno

```
public interface Agenda {
    public void aggiungi(String evento, LocalDateTime data)
    public void stampa();
    public Agenda clone();
}
public class NoteBook implements Agenda {
    private List<String> note = new ArrayList<>();
    @Override public void aggiungi(String evento,
                                   LocalDateTime t) {
        note.add(evento + ", " + t.getDayOfWeek() + " "
                 + t.getHour() + ":" + t.getMinute());
    }
    @Override public void stampa() {
        note.forEach(e -> System.out.println(e));
    }
    @Override public Agenda clone() {
        // deep copy
        List<String> n = new ArrayList<>();
        n.addAll(note);
        NoteBook notenew = new NoteBook();
        notenew.note = n;
        return notenew;
    }
}
```



Il pattern **Command**

- **Intento**: incapsulare una richiesta in un oggetto, permettendo quindi di parametrizzare i client con differenti richieste, code o log di richieste, e supportare l'annullamento di operazioni

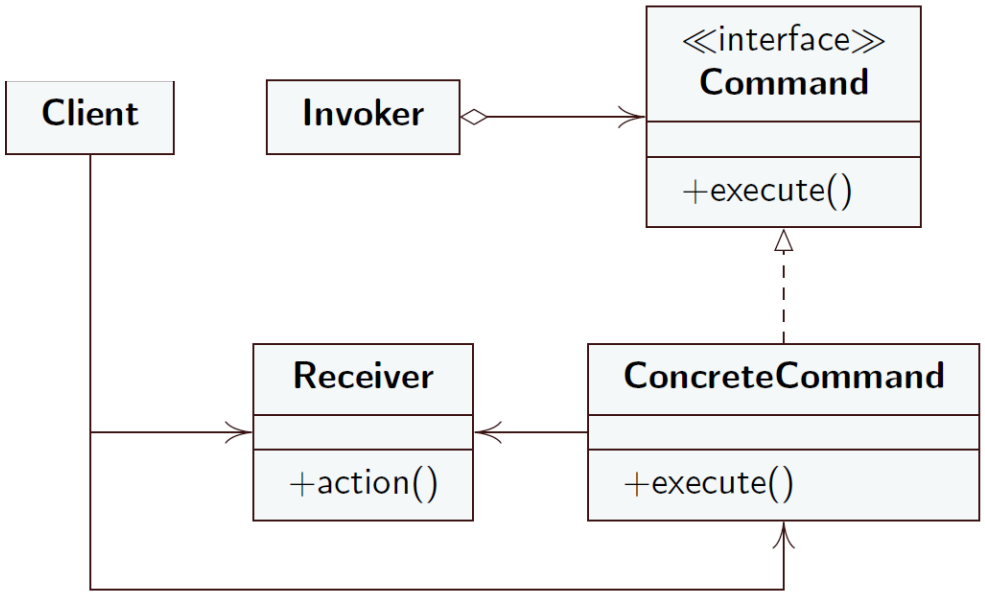
- **Motivazione**
  - Qualche volta può essere necessario mandare una richiesta a oggetti senza conoscere alcunché dell'operazione richiesta o del ricevente della richiesta

  - La richiesta viene trasformata in oggetto e quest'oggetto può essere immagazzinato o passato come altri oggetti

  - La chiave di questo pattern è una classe astratta Command che dichiara un'interfaccia per eseguire operazioni. Le sottoclassi specificano la coppia ricevente e azione, immagazzinando il ricevente come attributo e implementando il metodo che effettua le richieste. Il ricevente conosce ciò che è necessario per servire la richiesta

Il pattern **Command**

- **Struttura**:



- **Partecipanti**
  - Command definisce un'interfaccia per eseguire un'operazione
  - ConcreteCommand definisce il collegamento fra un oggetto Receiver e un'azione; implementa l'operazione execute chiamando le corrispondenti operazioni sul Receiver
  - Client crea un oggetto ConcreteCommand e imposta il suo Receiver
  - Invoker chiede al comando di effettuare la richiesta
  - Receiver conosce come effettuare le operazioni associate alla richiesta. Qualunque classe può essere un Receiver

Il pattern **Command**

- **Collaborazioni**:
  - Il client crea un oggetto ConcreteCommand e specifica il suo oggetto Receiver
  - Un oggetto Invoker tiene l'oggetto ConcreteCommand
  - L'oggetto Invoker effettua una richiesta chiamando execute() sull'oggetto Command. Quando i comandi sono reversibili, il ConcreteCommand conserva lo stato per cancellare il comando e ritornare a prima della chiamata a execute()
  - L'oggetto ConcreteCommand chiama le operazioni sul suo Receiver per portare a termine la richiesta

- **Conseguenze**:
  - Il pattern Command disaccoppia l'oggetto che invoca l'operazione da quello che conosce come effettuarla
  - I comandi diventano oggetti e possono essere manipolati ed estesi come altri oggetti
  - Si possono assemblare comandi in un comando composto (MacroCommand). In generale, i comandi composti sono un'istanza del design pattern Composite
  - E' facile aggiungere nuovi comandi, poiché non bisogna cambiare classi esistenti

Il pattern **Command**

- **Implemetazione**:
  - Un Command può avere un'ampia gamma di abilità. Potrebbe solo definire il legame con un ricevente e l'azione da eseguire.
    Oppure, implementare tutto facendo così a meno del ricevente

  - I Command possono supportare undo se hanno la capacità di rendere reversibile la loro esecuzione. Per questo la classe ConcreteCommand potrebbe aver bisogno di conservare uno stato aggiuntivo. Il ricevente deve fornire le operazioni che permettono al command di far tornare il ricevente allo stato precedente

  - Per un livello di undo il command deve conservare l'ultima attività eseguita, per più livelli di undo (e redo), sarà necessario avere una lista di comandi eseguiti

Il pattern **Command**

- **Esempio**:
  - Si vuol avere un calendario e un menù per aggiungere o togliere note sul calendario
  - Menu è un Invoker
  - Calendario è un Receiver
  - Updater è un ConcreteCommand