Progettazione e Analisi Object Oriented Vademecum Design Patterns

Andrea Mansi UniUD - 137857

II° Semestre 2021/2022

Indice

Design patterns

Gli elementi per il riuso del software ad oggetti (design patterns - schemi progettuali) sono dei modelli logici che descrivono una soluzione generale a un problema di progettazione ricorrente, verificabile in fase di progetto e sviluppo del software.

Nello specifico, ciascun design pattern attribuisce un nome al problema risolto, astrae e identifica gli aspetti principali della struttura utilizzata per la soluzione del problema, identifica le classi e le istanze partecipanti e la distribuzione delle responsabilità, infine, descrive quando e come può essere applicato. In breve definisce un problema, i contesti tipici in cui si verifica e la soluzione ottimale allo stato dell'arte.

Il concetto di design pattern nasce con la pubblicazione, nel 1994, del libro "Design Patterns: Elements of Reusable Object-Oriented Software", i cui autori vengono spesso chiamati "Gang of Four (GoF)". Nel libro, vengono illustrati 23 pattern, suddivisi in 3 categorie:

- **Strutturali**
    1. Adapter (adattatore)
    2. Façade (facciata)
    3. Composite (composto)
    4. Decorator (decoratore)
    5. Bridge (ponte)
    6. Proxy (proxy)
    7. Flyweight (peso piuma)
- **Comportamentali**
    8. Factory method (metodo fabbrica)
    9. Abstract factory (fabbrica astratta)
    10. Singleton (singoletto)
    11. Prototype (prototipo)
    12. Builder (costruttore)
- **Creazionali**
    13. Template method (metodo sagoma)
    14. Strategy (strategia)
    15. State (stato)
    16. Command (comando)
    17. Observer (osservatore)
    18. Mediator (mediatore)
    19. Memento (ricordo)
    20. Iterator (iteratore)
    21. Visitor (visitatore)
    22. Chain of responsability (catena di responsabilità)
    23. Interpreter (interprete)

Le tre categorie possono essere descritte come segue:

- **Strutturali**: spiegano come organizzare e assemblare oggetti e classi in strutture più complesse, mantenendo un elevato livello di flessibilità ed efficienza;
- **Comportamentali**: si prendono cura di fornire una comunicazione efficace tra gli oggetti e di distribuire correttamente le singole responsabilità;
- **Creazionali**: forniscono meccanismi di creazione di oggetti che aumentano la flessibilità e il riuso del codice preesistente.

1 Adapter

1.1 Scopo

Il fine dell'adapter è di fornire una soluzione astratta al problema dell'interoperabilità tra interfacce differenti. Il problema si presenta ogni qual volta in fase di progetto del software si debbano utilizzare sistemi di supporto (come per esempio librerie/API) la cui interfaccia non è perfettamente compatibile con quanto richiesto da applicazioni già esistenti.

Invece di dover riscrivere parte del sistema, compito oneroso e non sempre possibile se non si ha a disposizione il codice sorgente, può essere comodo scrivere un adapter che faccia da tramite tra l'interfaccia già presente e il codice del cliente.

In breve, quindi, l'adapter adatta l'interfaccia di una classe già pronta (il cui codice sorgente spesso non è accessibile) all'interfaccia che il cliente si aspetta. La metafora è quella del riduttore/adattatore per prese di corrente.

1.2 Casi d'uso

- È richiesto l'utilizzo di una classe esistente che presenti un'interfaccia diversa da quella desiderata dal cliente;
- È richiesta la scrittura di una classe senza poter conoscere a priori le altre classi con cui dovrà operare, in particolare senza poter conoscere quale specifica interfaccia sia necessario che la classe debba presentare alle altre.

1.3 Diagramma UML

Distinguiamo due tipologie di adapter:

- **Object-Adapter:** basato su delega e composizione;
- **Class-Adapter:** basato su ereditarietà in cui l'adattatore eredita sia dall'interfaccia attesa (che deve essere un'interfaccia) sia dalla classe adattata.

Diagramma UML dell'Object-Adapter:

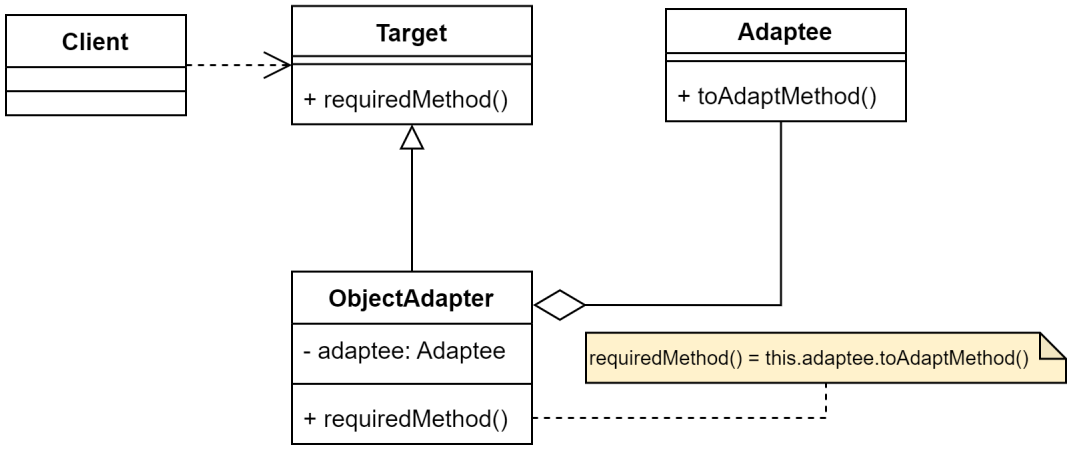
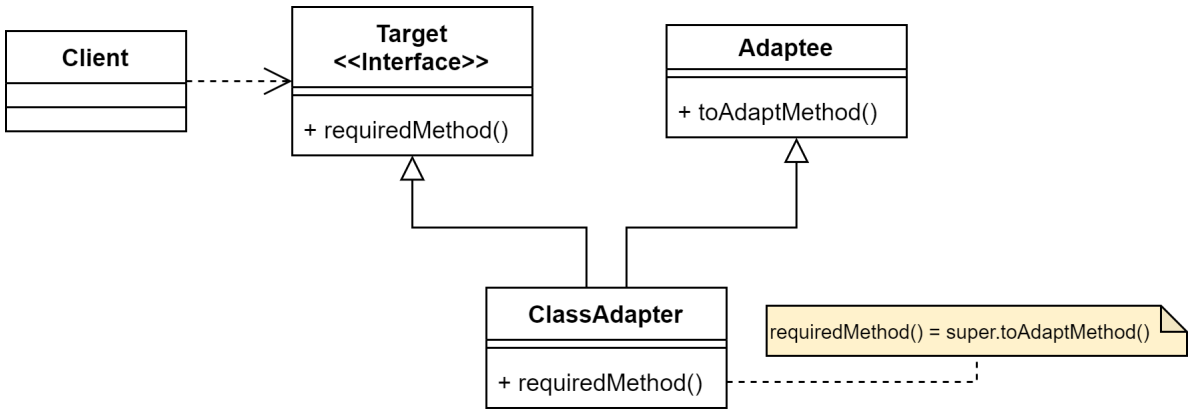


Diagramma UML del Class-Adapter:

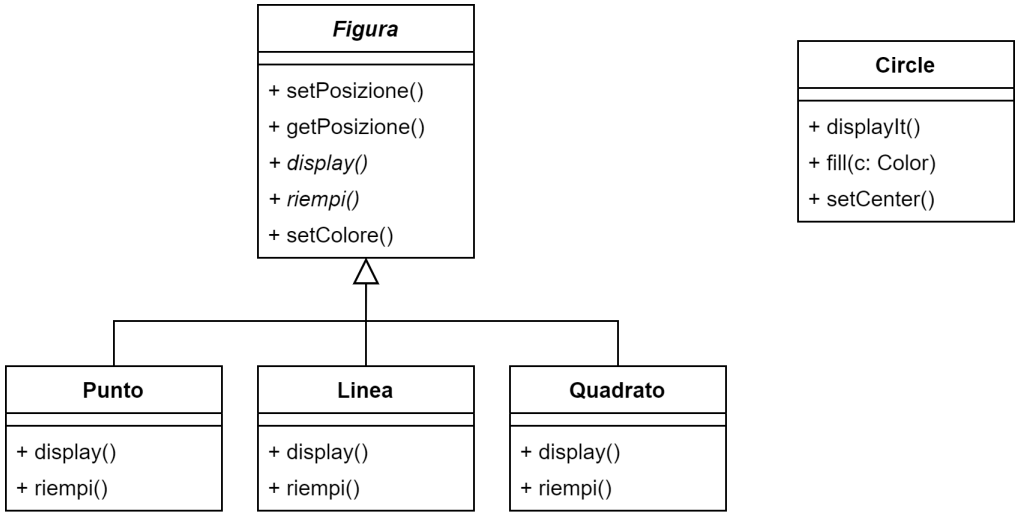


1.4 Partecipanti

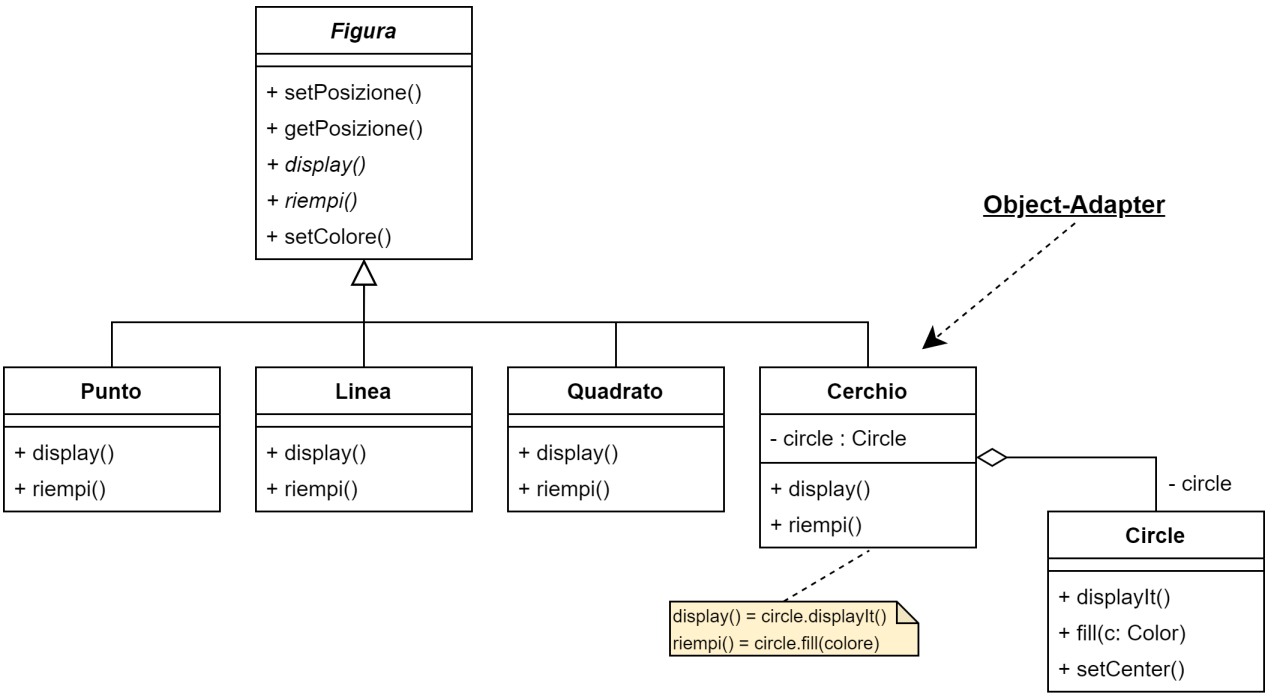
- **Adaptee:** l'interfaccia (o classe) che ha bisogno di essere adattata (ad esempio l'API);
- **Target:** definisce l'interfaccia che usa il cliente;
- **Client:** colui che utilizza indirettamente i servizi dell'interfaccia adattata (`Adaptee`) tramite l'interfaccia attesa `Target` (grazie all'adapter);
- **Adapter:** adatta l'interfaccia `Adaptee` all'interfaccia `Target`.

1.5 Esempio

Possediamo la gerarchia qui presente e vogliamo aggiungere la classe `Circle` (non compatibile e il cui codice sorgente non è accessibile).



Creiamo un adapter per la classe `Circle`:



La classe `Cerchio` deve avere una certa interfaccia attesa, imposta dall'ereditarietà della gerarchia indotta dalla superclasse astratta `Figura`. La classe `Circle` non è compatibile come sottoclasse e viene quindi adattata tramite un adapter. Questo è un esempio di Object-Adapter.

2 Façade

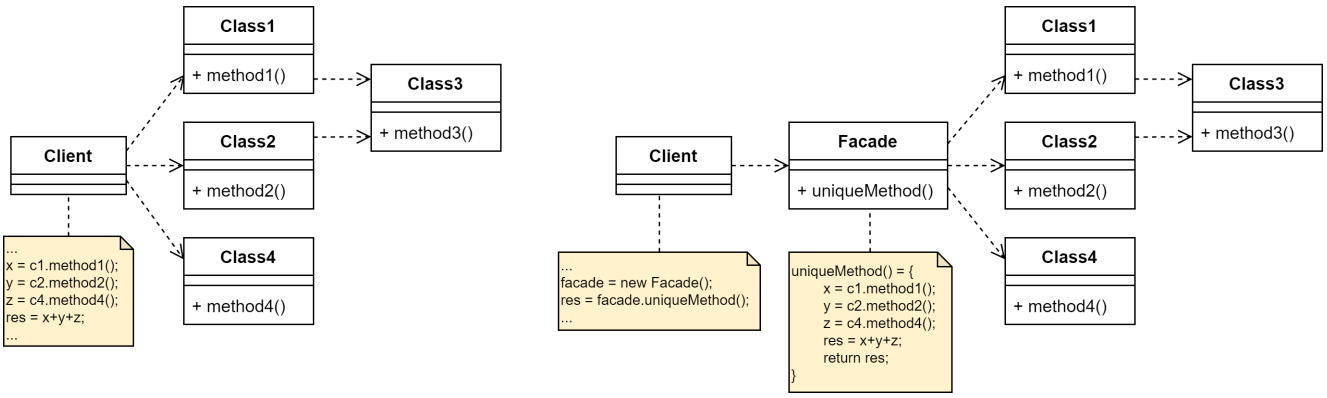
2.1 Scopo

Il fine del façade (facciata) è di fornire un'interfaccia più semplice di sottosistemi che presentano interfacce molto complesse e diverse/sparse tra loro. Promuove un accoppiamento debole fra cliente e sottosistema (la dipendenza non è transitiva), nascondendo al cliente le componenti complesse del sottosistema. Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema.

2.2 Casi d'uso

- Il client deve accedere a diverse classi molto differenti per compiere un'operazione;
- Si vuole rendere indipendente l'implementazione della classe client dalle varie classi utilizzate;
- Si vuole fornire un'interfaccia "contex-specific" per funzionalità più generiche;
- Si vuole migliorare la leggibilità e l'usabilità di una libreria software mascherando l'interazione con componenti più complesse dietro un'unica (e spesso semplificata) API.

2.3 Diagramma UML



2.4 Partecipanti

- **Classi:** le classi che forniscono i singoli servizi alla classe `Client`;
- **Façade:** colui che semplifica l'accesso e l'utilizzo dei servizi di classi multiple al client tramite un'unica interfaccia;
- **Client:** colui che utilizza i servizi delle classi per compiere un'operazione.

2.5 Esempio

Nel seguente esempio possiamo vedere come l'utilizzo di una façade (`ComputerFacade`) semplifichi l'utilizzo delle sottoclassi al `Client`, che non è più tenuto a interagire con le singole classi e a dipendere da esse per implementare il comportamento voluto.

```
class CPU {
  public void freeze() { ... }
  public void jump(long position) { ... }
  public void execute() { ... }
}

class Memory {
  public void load(long position, byte[] data) { ... }
}

class HardDrive {
  public byte[] read(long lba, int size) { ... }
}

class ComputerFacade {
  private CPU processor;
  private Memory ram;
  private HardDrive hd;

  public ComputerFacade() {
      this.processor = new CPU();
      this.ram = new Memory();
      this.hd = new HardDrive();
  }

  public void start() {
      processor.freeze();
      ram.load(BOOT_ADDRESS, hd.read(BOOT_SECTOR, SECTOR_SIZE));
      processor.jump(BOOT_ADDRESS);
      processor.execute();
  }
}

class Client {
  public static void main(String[] args) {
      ComputerFacade computer = new ComputerFacade();
      computer.start();
  }
}
```

Façade vs Adapter vs Wrapper
Il concetto di wrapper è molto generico, tendenzialmente viene definito come "un modulo sofware che ne riveste un altro", concetto applicabile a metodi, tipi, classi, oggetti, etc.

Façade e Adapter sono di fatto dei wrapper, entrambi si basano su un'interfaccia, ma:

- Façade la semplifica;
- Adapter la converte/adatta ad un'altra.

3 Composite

3.1 Scopo

Lo scopo del composite è di permettere di trattare un gruppo di oggetti come se fossero l'istanza di un oggetto singolo. Il design pattern composite organizza gli oggetti in una struttura ad albero per rappresentare gerarchie parte-tutto. In tale albero i nodi sono delle composite e le foglie sono oggetti semplici.

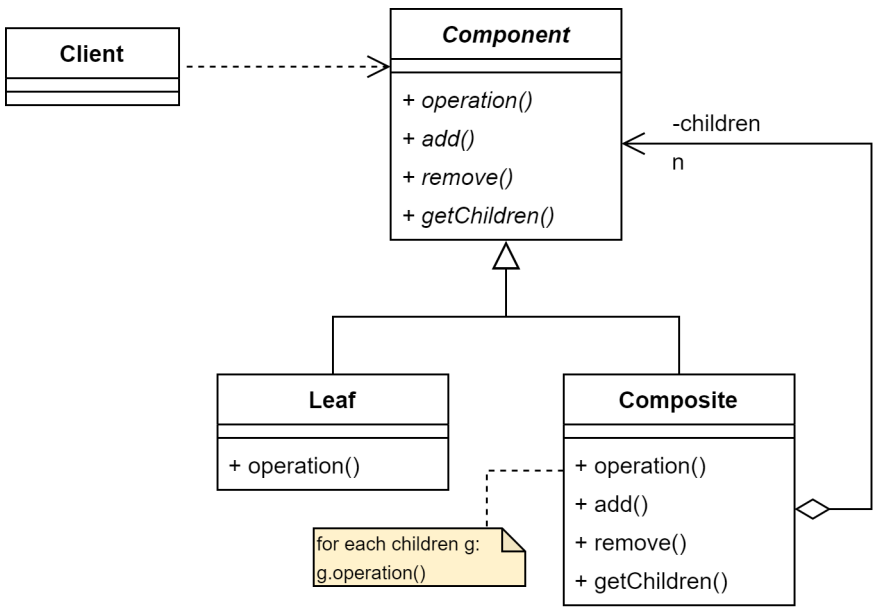
È utilizzato per dare la possibilità al cliente di manipolare oggetti singoli e composizioni in modo uniforme, senza necessità di distinzioni.

Il concetto fondamentale è che il programmatore manipola ogni oggetto dell'insieme dato nello stesso modo: sia esso un "raggruppamento" o un oggetto singolo.

3.2 Casi d'uso

- Il client ha la necessità di ignorare la differenza tra oggetti compositi e oggetti singoli;
- Si verificano casi in cui molti oggetti vengono utilizzati in modo simile, compreso il codice per la loro gestione.

3.3 Diagramma UML



3.4 Partecipanti

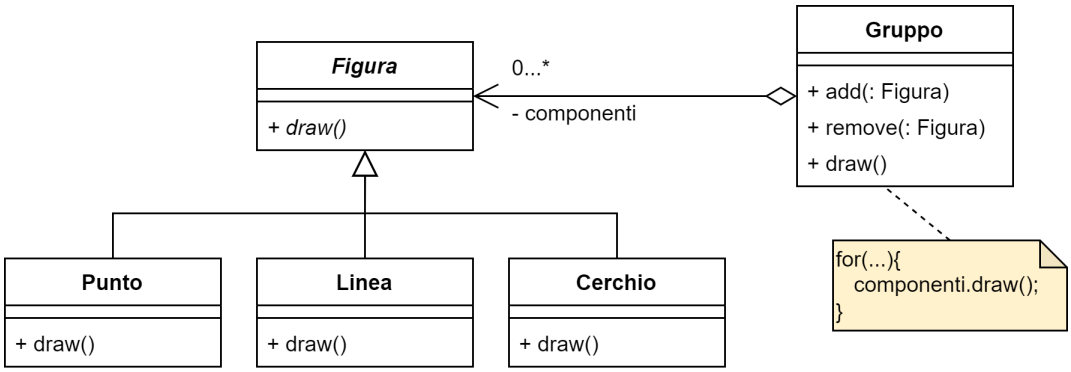
- **Client:** manipola gli oggetti attraverso l'interfaccia `Component`.
- **Component:** dichiara l'interfaccia per gli oggetti nella composizione, per accedervi e manipolarli, imposta un comportamento di default per l'interfaccia comune a tutte le classi. Può definire un'interfaccia per l'accesso al padre del componente e la implementa se è appropriata.
- **Composite:** definisce il comportamento per i componenti aventi figli, salva i figli e implementa le operazioni ad essi connesse nell'interfaccia `Component`.
- **Leaf:** definisce il comportamento degli oggetti primitivi, cioè delle foglie.

3.5 Esempio

Si consideri un programma di grafica che deve consentire di:

- Creare oggetti semplici (punto, linea, cerchio, etc.);
- Raggruppare dinamicamente oggetti semplici in oggetti compositi (ad esempio: rettangolo = 4 linee, etc.);
- Si ha l'esigenza di trattare gli oggetti compositi come se fossero oggetti semplici nel svolgere operazioni di spostamento, ridimensionamento, etc.

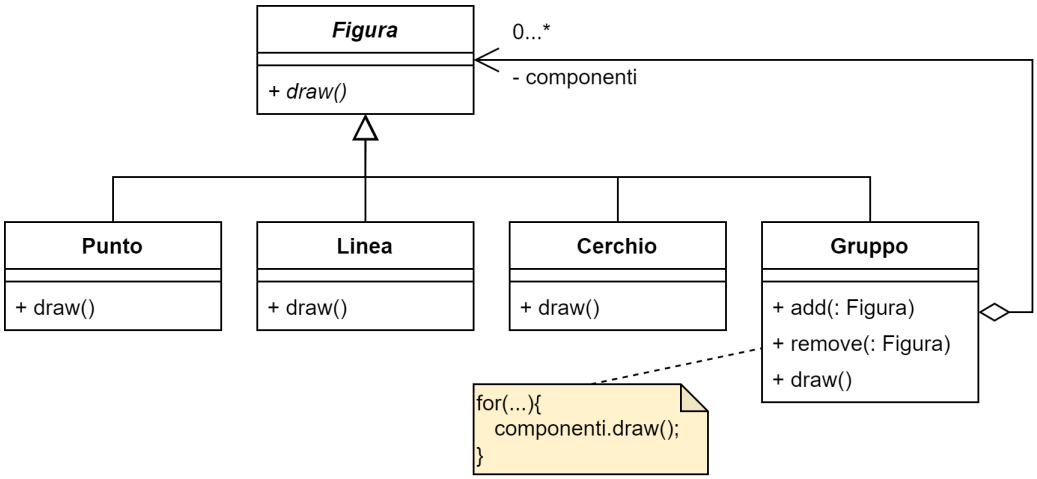
Un esempio che non fa uso del composite:



Questa implementazione non è ottimale in quanto la gestione dei gruppi (ad esempio `Rettangolo`) e delle figure primitive è diversa.

Facendo uso del pattern, `Gruppo` diventa sottoclasse di `Figura`. È un pattern "ricorsivo" in quanto `Gruppo` contiene `Figura` (e siccome un gruppo è una figura, ossia alcune figure sono gruppi; un gruppo può contenere gruppi).

Il composite semplifica il cliente, che tratta strutture composte e singoli oggetti allo stesso modo (permettendo così di passare oggetti composti anche dove oggetti primitivi sono richiesti). `Leaf` e `Composite` possono essere astratte ed avere sottoclassi; `Component` può essere un'interfaccia.



Un esempio di implementazione:

```
interface Graphic {
  public void print();
}

class CompositeGraphic implements Graphic {
  private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

  public void print() {
      for (Graphic graphic : mChildGraphics) {
          graphic.print();
      }
  }

  public void add(Graphic graphic) {
      mChildGraphics.add(graphic);
  }

  public void remove(Graphic graphic) {
      mChildGraphics.remove(graphic);
  }
}

class Ellisse implements Graphic {
  public void print() {
      System.out.println("Ellisse");
  }
}

class Rettangolo implements Graphic {
  public void print() {
      System.out.println("Rettangolo");
  }
}
```

Il codice del client potrebbe essere il seguente; in cui si può notare come per eseguire il print di una figura (`Graphic`) semplice o un gruppo di figure (`CompositeGraphic`) non ci siano differenze (si ignora la differenza tra oggetti compositi e singoli).

```
  public static void main(String[] args) {

    Ellisse ellisse1 = new Ellisse();
    Ellisse ellisse2 = new Ellisse();
    Rettangolo rettangolo1 = new Rettangolo();
    Rettangolo rettangolo2 = new Rettangolo();

    CompositeGraphic graphicEllissi = new CompositeGraphic();
    CompositeGraphic graphicRettangoli = new CompositeGraphic();
    CompositeGraphic graphicMixed = new CompositeGraphic();

    graphicEllissi.add(ellisse1);
    graphicEllissi.add(ellisse2);

    graphicRettangoli.add(rettangolo1);
    graphicRettangoli.add(rettangolo2);

    graphicMixed.add(graphicEllissi);
    graphicMixed.add(graphicRettangoli);

    // Stampa tutte le figure
    graphicMixed.print();

    // Stampa solo le elissi
    graphicElissi.print();

    // Stampa solo il primo rettangolo
    rettangolo1.print();
}
```

4 Decorator

4.1 Scopo

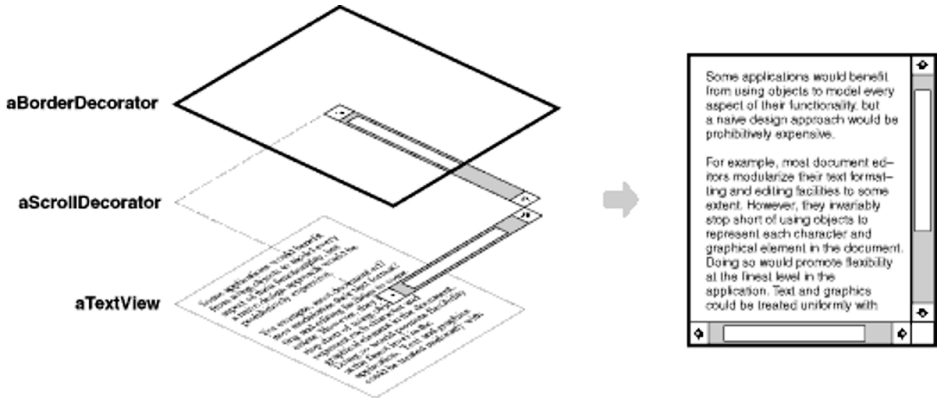
Consente di aggiungere nuove funzionalità ad oggetti già esistenti. Questo avviene costruendo una nuova classe decoratore che "avvolge" l'oggetto originale.

Al costruttore del decoratore si passa come parametro l'oggetto originale. È altresì possibile passarvi un differente decoratore. In questo modo, più decoratori possono essere concatenati l'uno all'altro, aggiungendo così in modo incrementale funzionalità alla classe concreta (che è rappresentata dall'ultimo anello della catena).

Il decorator si pone come valida alternativa all'uso dell'ereditarietà singola o multipla (alternativa più flessibile all'uso delle sottoclassi per estendere le funzionalità).

Con l'ereditarietà, infatti, l'aggiunta di funzionalità avviene staticamente secondo i legami definiti nella gerarchia di classi e non è possibile ottenere al run-time una combinazione arbitraria delle funzionalità, né la loro aggiunta/rimozione.

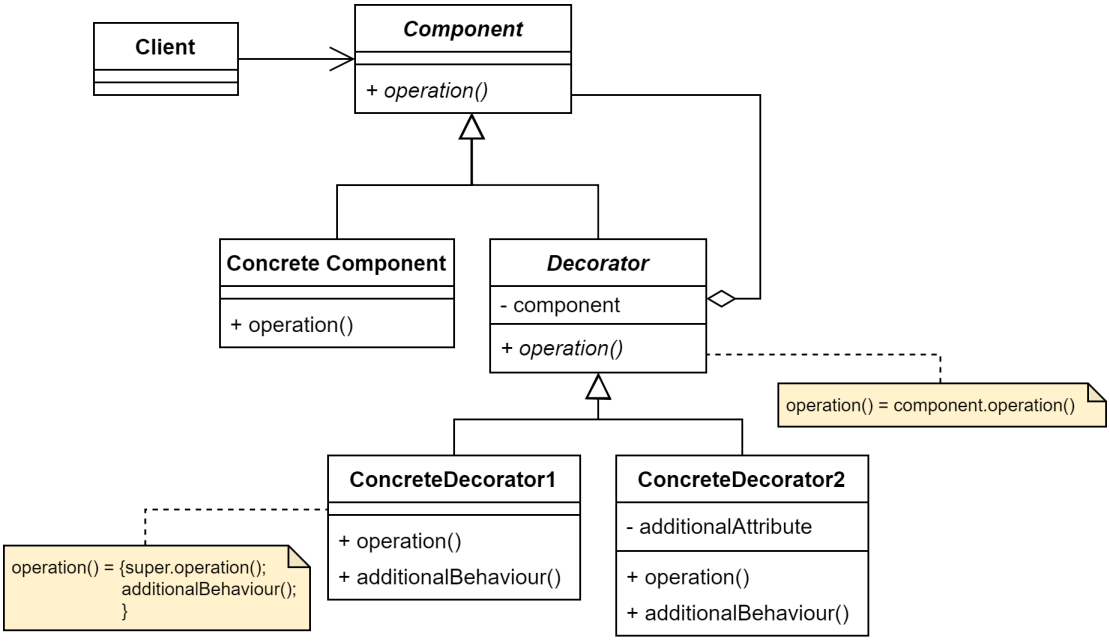
Decoratore	**Ereditarietà**
Dinamico (run-time)	Statico (compile-time)
Non vincolante per il cliente	Vincolante per il cliente, che non può inventarsi combinazioni non previste
Aggiunge responsabilità ad un singolo oggetto	Aggiunge responsabilità a tutte le istanze di una classe
Evita l'esplosione combinatoria	Incline all'esplosione combinatoria



4.2 Casi d'uso

- Si vuole estrema libertà a run-time relativamente alla gestione (aggiunta/rimozione) di funzionalità agli oggetti/istanze;
- Il numero di combinazioni di comportamenti distinti per i vari oggetti possibili è troppo numeroso da gestire "staticamente" tramite una gerarchia di classi.

4.3 Diagramma UML

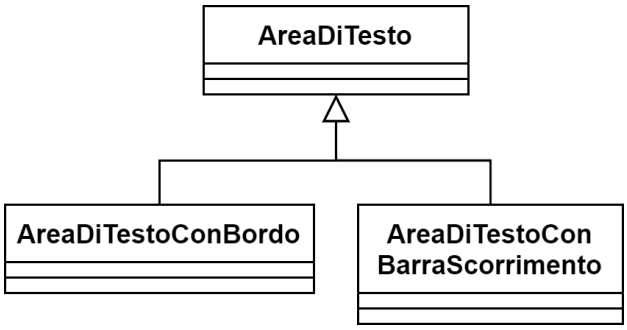


4.4 Partecipanti

- **Component:** definisce l'interfaccia dell'oggetto a cui verranno aggiunte nuove funzionalità;
- **ConcreteComponent:** definisce l'oggetto concreto al quale aggiungere le funzionalità;
- **Decorator:** mantiene un riferimento a un'istanza di `Component` e definisce un'interfaccia ad esso conforme;
- **ConcreteDecorator:** aggiunge funzionalità al `Component`, implementando l'interfaccia definita da `Decorator`.

4.5 Esempio

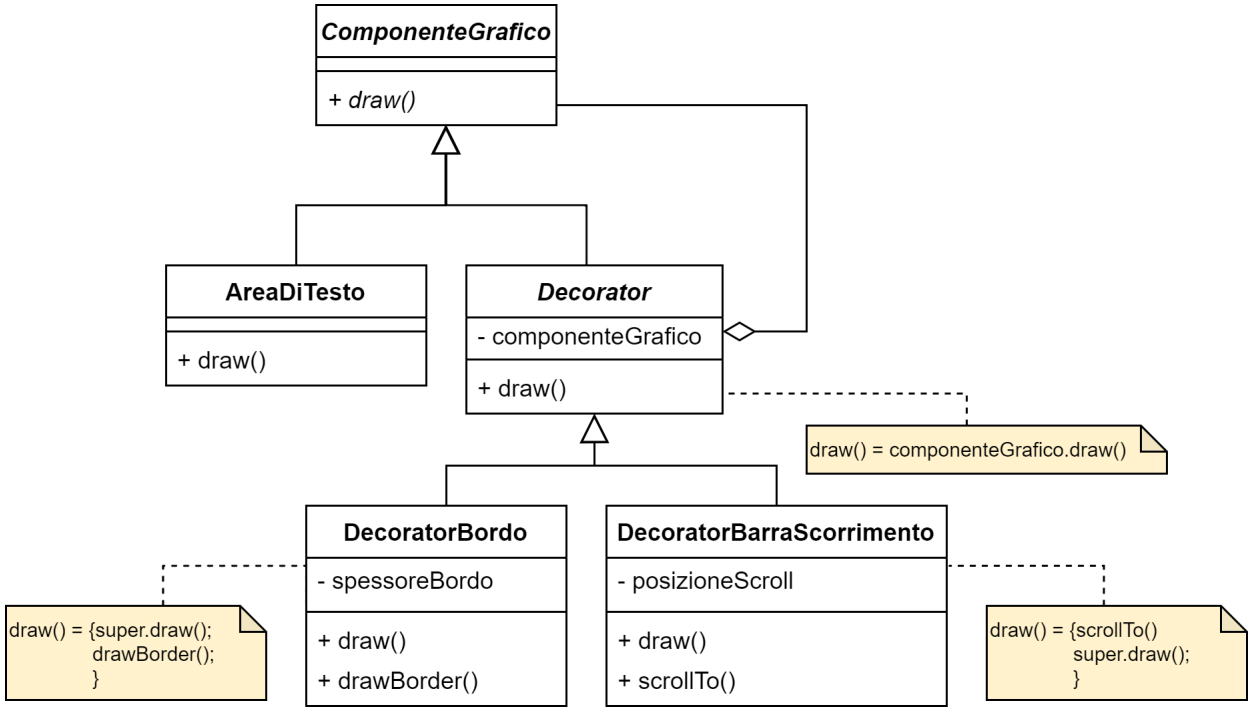
In ambiente grafico, se vogliamo due o più tipologie di `AreaDiTesto`, potremmo, tramite l'ereditarietà, costruire la seguente gerarchia di classi:



Così facendo posso creare le istanze della classe voluta. Ma cosa succede se voglio un'area di testo con bordo e barra di scorrimento? creo un'altra sottoclasse? si ha un'esplosione combinatoria. Contrariamente all'approccio statico appena descritto, si può seguire un approccio dinamico, racchiudendo l'oggetto da "decorare" in un altro oggetto, responsabile di gestirne la decorazione: il decorator.

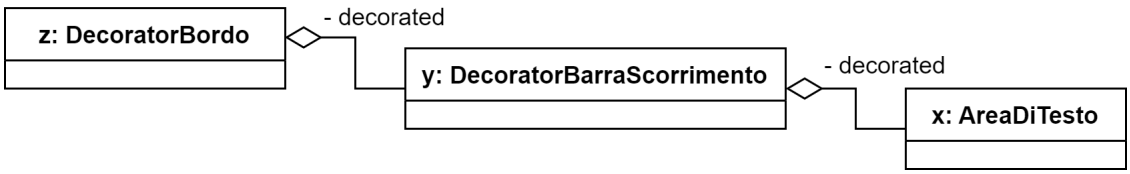
Il decorator ha il compito di trasferire/delegare e aggiungere funzionalità all'oggetto. Il decorator ha l'interfaccia conforme all'oggetto decorato, e quindi è trasparente al client che può non sapere se sta parlando con un oggetto decorato o non decorato.

Adottando il decorator al nostro esempio otteniamo:



Possiamo ora ottenere una catena di decorazioni, con alla fine un'istanza di `AreaDiTesto`.

```
ComponenteGrafico x =
  new DecoratorBarraScorrimento(
    new DecoratorBordo(
      new AreaDiTesto()
    )
  );
```



5 Bridge

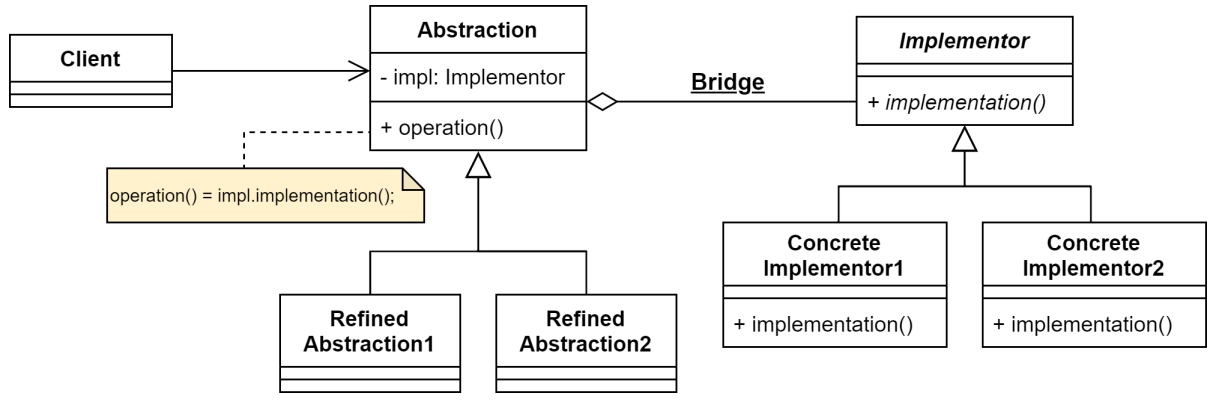
5.1 Scopo

Lo scopo del pattern bridge ("ponte") è di seperare l'interfaccia di una classe dalla sua implementazione. In tal modo è possibile sfruttare l'ereditarietà per far evolvere l'interfaccia o l'implementazione in modo separato e indipendente. Il bridge quindi disaccoppia ("decoupling") un'astrazione dalla sua implementazione, rendendolo uno meno dipendente dall'altro.

5.2 Casi d'uso

- Si vuole dividere e organizzare una classe monolitica che ha più varianti di alcune funzionalià;
- Si vuole estendere con maggiore libertà una classe in diverse modalità indipendenti;
- Si vuole cambiare implementazione a run-time;

5.3 Diagramma UML



5.4 Partecipanti

- **Abstraction:** definisce l'interfaccia astratta e mantiene una referenza dell'`Implementor`;
- **Implementor:** definisce l'interfaccia per le classi che implementano le operazioni;
- **Concrete Implementor:** implementa l'interfaccia `Implementor` e le singole operazioni;
- **RefinedAbstraction:** estende l'interfaccia definita dall'`Abstraction`.

5.5 Esempio

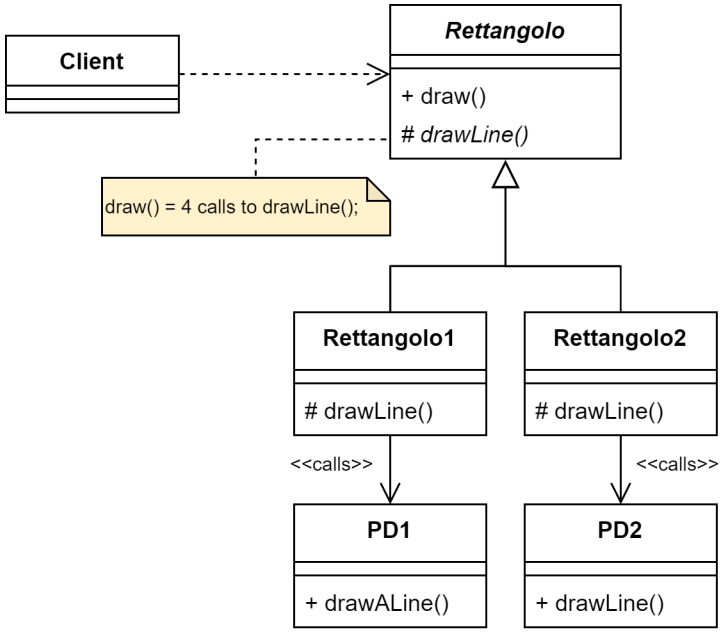
Si richiede un programma che permetta di disegnare rettangoli, usando in alternativa uno dei due programmi di disegno:

- PD1: `drawALine(x1, y1, x2, y2)`
- PD2: `drawLine(x1, x2, y1, y2)`

Si vuole far sì che una voltra creato un rettangolo, non sia necessario distinguere che questo sia stato creato con PD1 o PD2. Il codice del cliente è il seguente:

```
Rettangolo[] r = new Rettangolo[...];
r[0] = new Rettangolo1(...); // di PD1
r[1] = new Rettangolo2(...); // di PD2
r[2] = new Rettangolo2(...); // di PD2
...
for(i = ...){
  r[i].draw();
}
```

Abbiamo quindi 2 tipologie di rettangoli, uno usa PD1 e l'altro PD2; una prima soluzione è astrarre `Rettangolo`; all'istanziazione so se istanziare `Rettangolo1` o `Rettangolo2`.

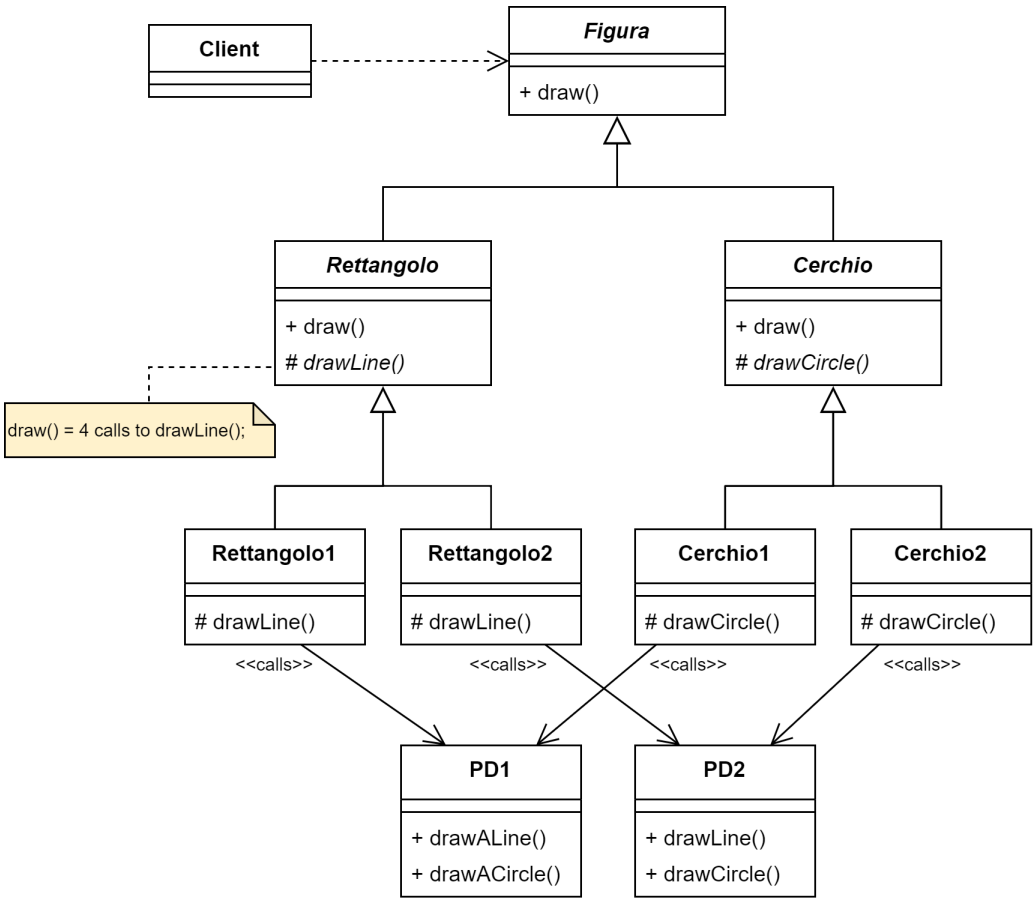


La soluzione sembra funzionare; ma immaginiamo di voler disegnare anche cerchi, quadrati, etc. Avremmo quindi per ciascuno di essi una funzione `draw()` del PD1 e del PD2, similmente ai rettangoli; consentendo sempre di astrarre le figure al cliente.

Aggiungendo ad esempio il `Cerchio` avremmo casi come il seguente codice del cliente:

```
Figura[] f = new Figura[...];
f[0] = new Rettangolo1(...); // di PD1
f[1] = new Rettangolo2(...); // di PD2
f[2] = new Cerchio1(...); // di PD1
f[3] = new Cerchio1(...); // di PD1
f[4] = new Cerchio2(...); // di PD2
...
for(i = ...){
  f[i].draw();
}
```

Con la seguente organizzazione delle classi:

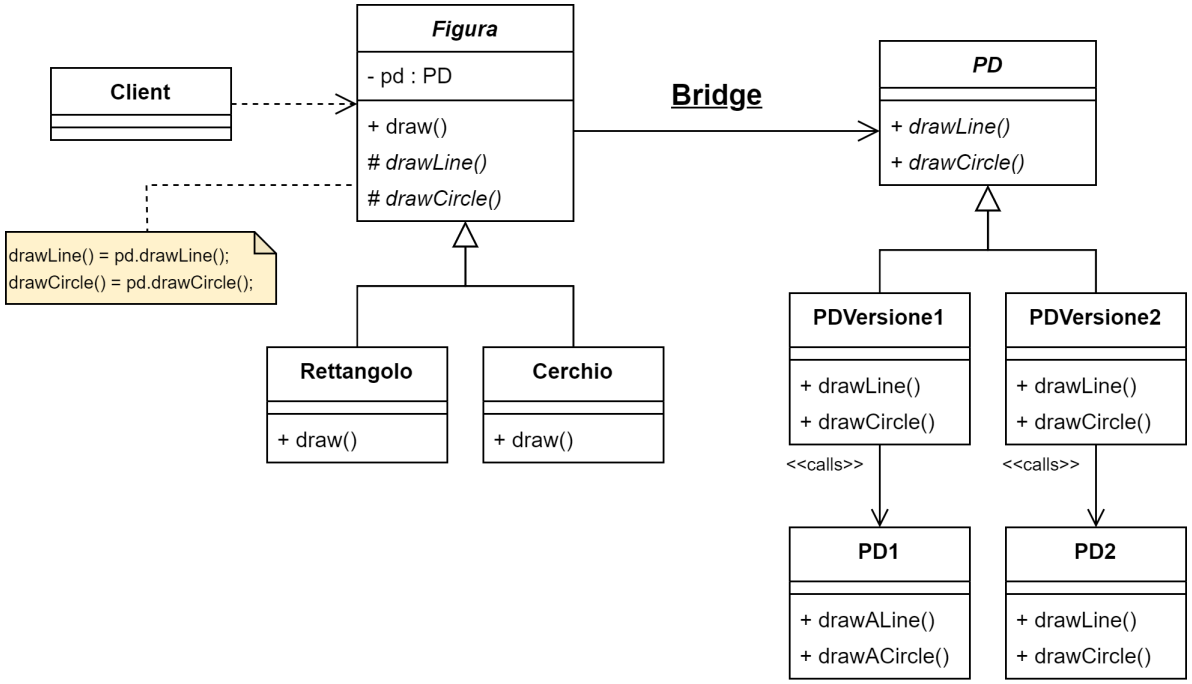


La soluzione sembra ok, ma se devo gestire molte figure? per ciascuna devo aggiungere una piccola gerarchia: una classe astratta e una sottoclasse per ciascun PD. E se ho molti altri programmi di disegno PD? le gerarchie diventano più estese. Di per sè la soluzione funziona ma può esplodere a livello combinatorio.

L'alternativa duale è scomporre prima per tipologia di PD e poi per tipo di figura; scambiando l'ordine. Non cambia molto, l'esplosione combinatoria è la stessa.

Queste due soluzioni presentano alto accoppiamento (non fra classi ma fra le astrazioni delle figure e le implementazioni - o viceversa nella soluzione duale).

C'è ridondanza e non c'è indipendenza tra le due. Appreso il problema, analizziamo una soluzione: usiamo un bridge pattern per separare le astrazioni sulle figure dalle loro implementazioni dei vari PD.



Il codice del cliente diventa, ad esempio:

```
Figura[] f = new Figura[...];
PD p1 = new PDVersione1();
PD p2 = new PDVersione2();
f[0] = new Rettangolo(p1); // di PD1
f[1] = new Rettangolo(p2); // di PD2
f[2] = new Cerchio(p1); // di PD1
...
for(i = ...){
  f[i].draw();
}
```

Quando si crea un'istanza di `Figura` specifico anche con quale implementazione. Questa può essere omessa e impostarne una come scelta di default.

6 Proxy

6.1 Scopo

Fornire un surrogato o un segnaposto per un oggetto, al fine di controllarne l'accesso.

Un proxy, nella sua forma più generale è una classe che fa da interfaccia per qualcos'altro. In breve, un proxy può essere visto come un wrapper che viene chiamato dal cliente per accedere a oggetti molteplici. Si noti come nel proxy, oltre a richiami a servizi dell'oggetto "wrappato", può essere fornita logica aggiuntiva, ad esempio del codice per il controllo di alcune precondizioni.

Per il cliente, l'uso del proxy e dell'oggetto reale è molto simile, in quanto implementano la stessa interfaccia.

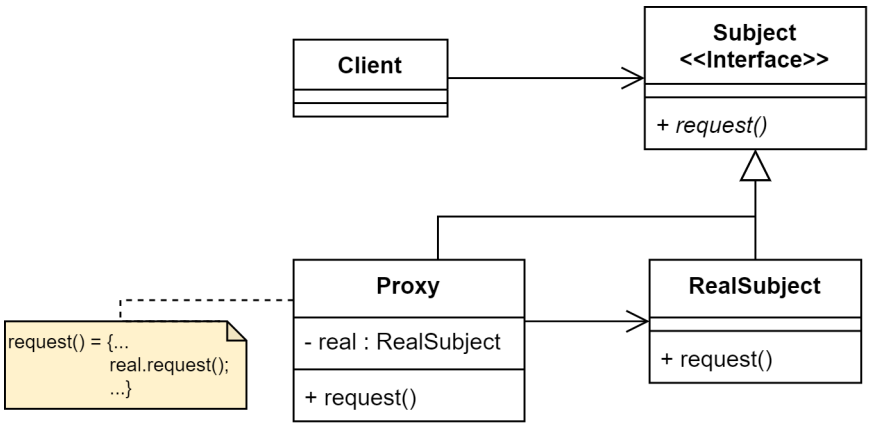
Esistono diverse tipologie di proxy:

- **Virtuale:** gestisce la creazione su richiesta di oggetti "costosi/pesanti"; si parla di "lazy initialization".
- **Protezione:** gestisce diritti di accesso per diversi oggetti;
- **Remoto:** rappresenta localmente un oggetto di uno spazio diverso;
- **Riferimento intelligente:** sostituisce un puntatore.

6.2 Casi d'uso

- Si vuole controllare l'accesso a un oggetto (sicurezza e/o efficienza);
- Si vogliono fornire funzionalità agguntive ad un oggetto quando gli si accede.

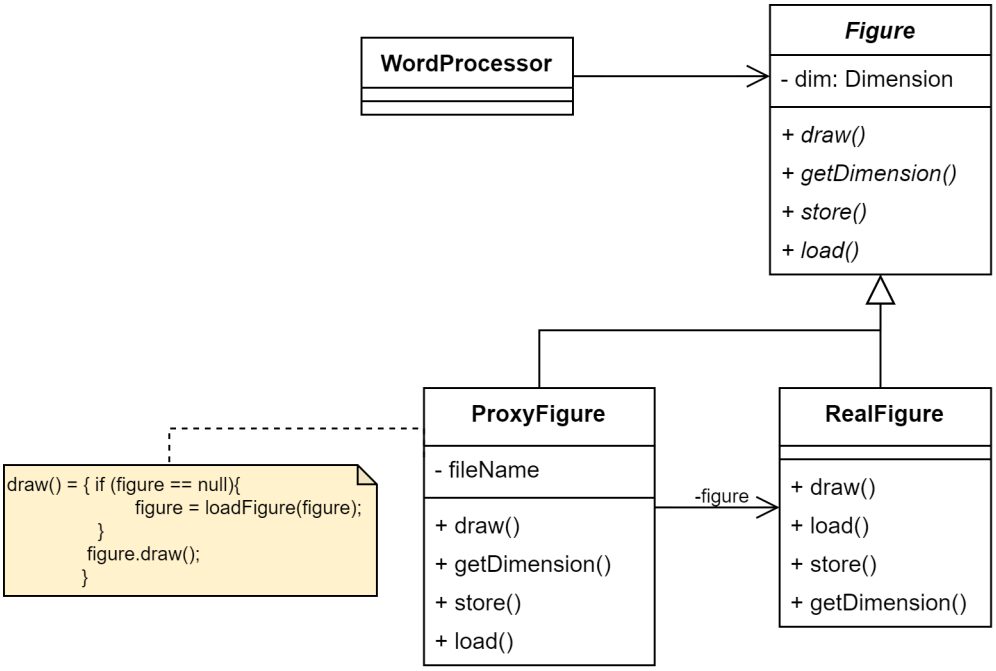
6.3 Diagramma UML



6.4 Partecipanti

- **Subject:** dichiara l'interfaccia per i singoli servizi; il `Proxy` deve essere congruo a tale interfaccia;
- **Proxy:** mantiene una referenza che punta all'istanza di un servizio reale (`RealSubject`). Dopo che il proxy ha finito i suoi compiti (ad esempio: lazy initialization, logging, access control, caching, etc.), inoltra la vera richiesta all'oggetto che fornisce il servizio concreto;
- **RealSubject:** classe che implementa gli effettivi servizi/business logic.

6.5 Esempio



7 Flyweight

7.1 Scopo

Permettere di separare la parte variabile di una classe dalla parte che può essere riutilizzata, in modo tale da condividere quest'ultima fra differenti istanze.

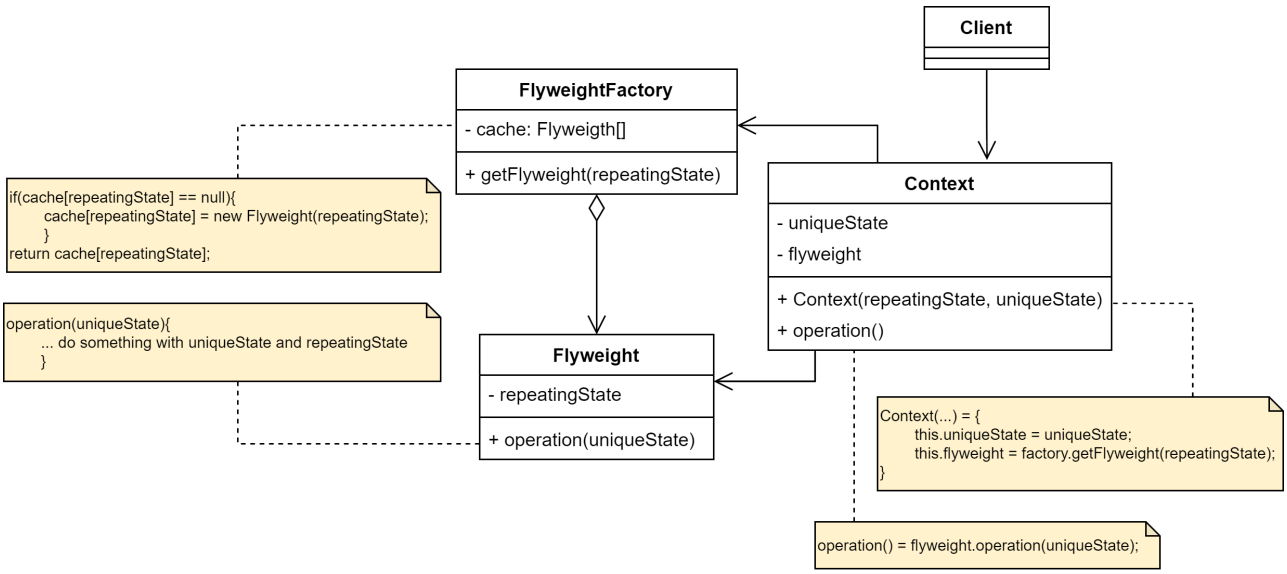
L'oggetto flyweight deve essere un oggetto immutabile, per permettere la condivisione tra diversi client e thread.

Si ottiene così un risparmio in utilizzo della memoria, visto che le parti uguali tra gli oggetti sono condivise e non replicate.

7.2 Casi d'uso

- Si hanno molte istanze della stessa classe che variano di poco o che hanno la maggioranza degli attributi sempre uguale;

7.3 Diagramma UML

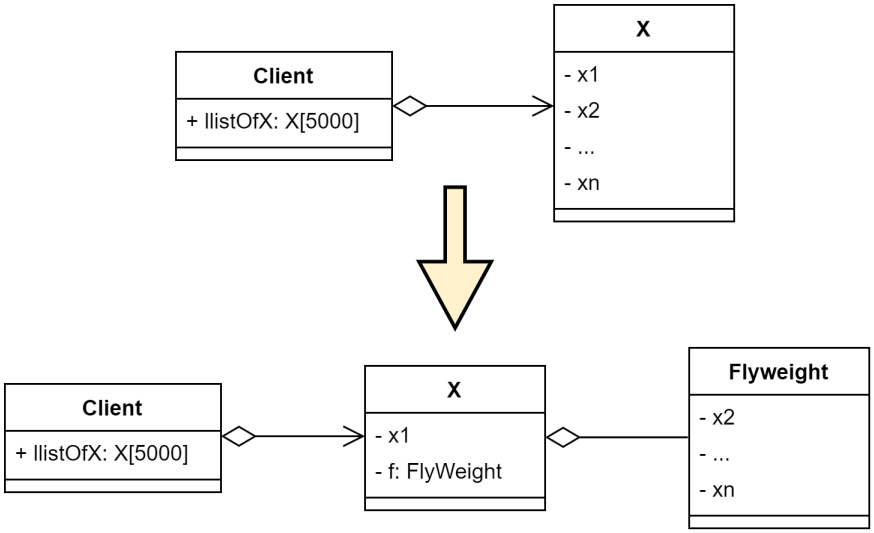


7.4 Partecipanti

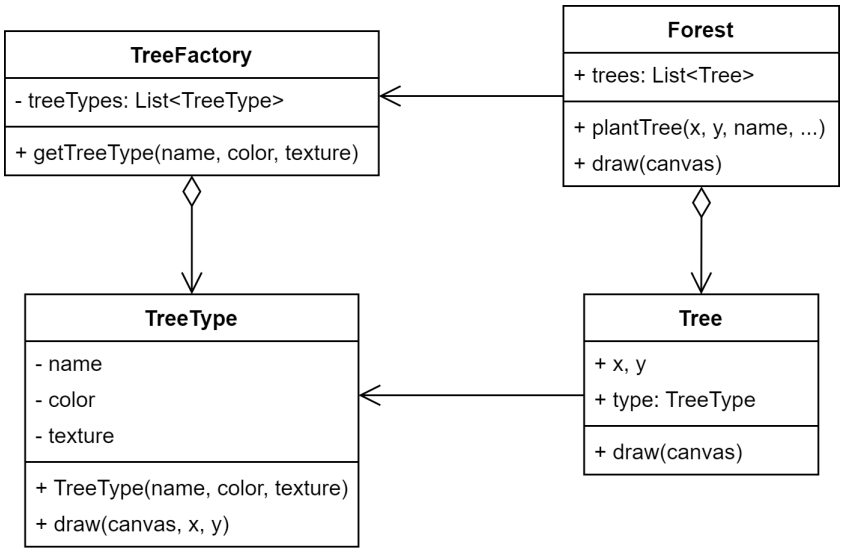
- **Context:** contiene lo stato effettivo dell'oggetto (composto dallo stato unico "intrinseco" e da una versione di quelli condivisi - contenuta nel flyweight); unico tra tutti gli oggetti;
- **FlyweightFactory:** gestice un insieme di flyweight esistenti e ne crea di nuovi quando necessari;
- **Flyweight:** contiene una porzione dello stato della classe/istanza iniziale (`Context`), condiviso con altre istanze. Questa porzione di stato è detta estrinseca.

7.5 Esempio

Nel seguente esempio generico, si ipotizzi di avere una classe `X` con n attributi, l'attributo x_1 è quasi sempre diverso (stato estrinseco), mentre gli attributi $x_2, ..., x_n$ sono spesso simili (pochi/condivisi - stato estrinseco). Invece di creare molte istanze di `X` con gli attributi simili duplicati, è buona idea riorganizzare i parametri $x_2, ..., x_n$ in una classe flyweight, istanziata poche volte (per i vari casi) e utilizzarla come attributo di `X`:



Un esempio più complesso è il seguente: si vuole ridurre l'uso di memoria nel rendering di milioni di oggetti "albero" in un canvas. Applicando il flyweight pattern, si estrae lo stato estrinseco dalla classe `Tree` e lo si sposta nella classe flyweight `TreeType`.



Implementazione dell'esempio:

```
class TreeType {
  public String name;
  public Color color;
  public Texture texture;

  public TreeType(name, color, texture){ ... }

  public void draw(canvas, x , y){ ... }
}

class TreeFactory {
  public List<TreeTypes> types = ...

  public TreeType getTreeType(name, color, texture){
    type = types.find(name, color, texture);
    if(type == null){
      type = new TreeType(name, color, texture);
      types.add(type);
    }
    return type;
  }
}

class Tree {
  public float x, y;
  public TreeType type;

  public Tree(x, y, type){ ... }

  public void draw(canvas, this.x, this.y){ ... }
}

class Forest {
  public List<Tree> trees = ...
  public TreeFactory = ...

  public void plantTree(x, y, name, color, texture){
    type = TreeFactory.getTreeType(name, color, texture);
    tree = new Tree(x, y, type);
    trees.add(tree);
  }

  public void draw(canvas){ foreach tree ... }
}
```

8 Factory Method

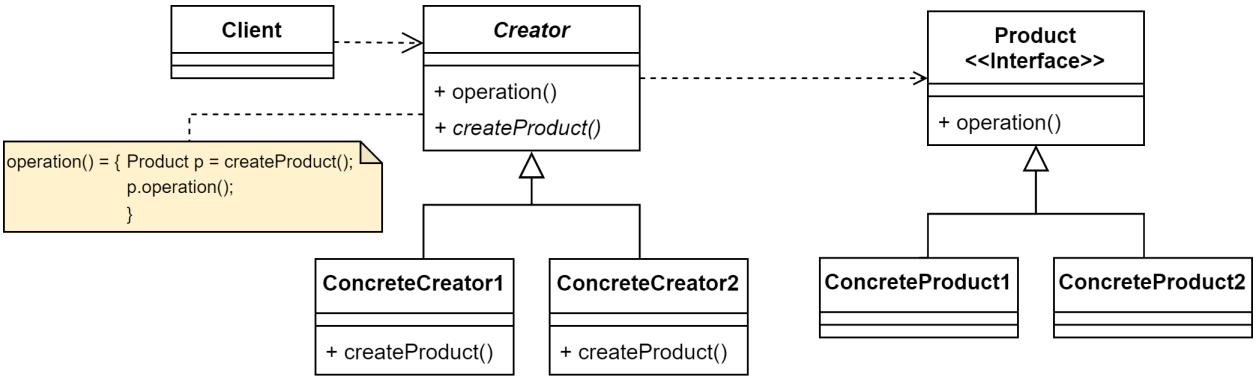
8.1 Scopo

L'obiettivo del pattern è di delegare l'istanziazione di una classe a una propria sottoclasse, che deciderà quale istanza creare e quale costruttore utilizzare. La creazione di un oggetto può spesso richiedere processi complessi la cui collocazione all'interno della classe di composizione potrebbe non essere appropriata; il pattern indirizza questo problema fornendo un metodo separato delegato alla creazione. Il factory method è tipicamente utilizzato nei framework object-oriented.

8.2 Casi d'uso

- La creazione di un oggetto preclude il suo riuso senza una significativa duplicazione di codice;
- La creazione di un oggetto richiede l'accesso ad informazioni o risorse che non dovrebbero essere contenute/esposte nella classe di composizione/Client;
- La gestione del ciclo di vita degli oggetti deve essere centralizzata in modo da assicurare un comportamento coerente all'interno dell'applicazione.

8.3 Diagramma UML



8.4 Partecipanti

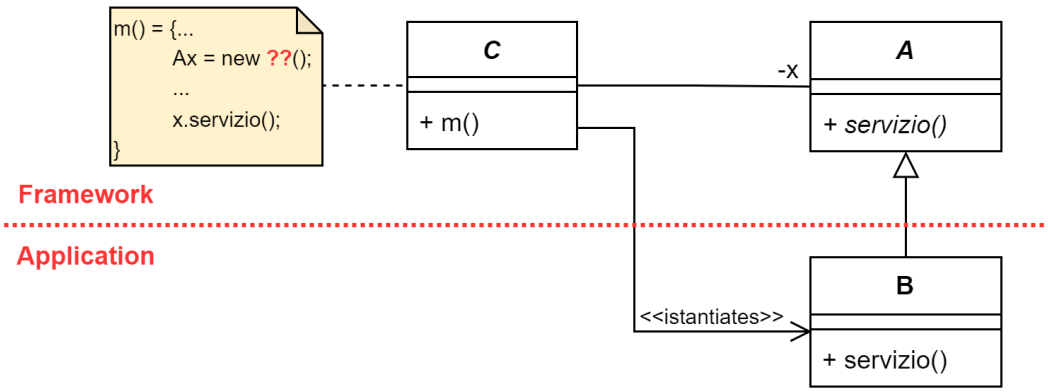
- **Creator:** dichiara il factory method `createProduct()` e ritorna il nuovo prodotto corretto. Il factory method può essere dichiarato astratto così da forzare le sottoclassi a implementare la propria versione del metodo.
- **Concrete Creators:** Sovrascrivono il factory method `createProduct()` così da ritornare il prodotto corretto;

- **Product:** dichiara l'interfaccia che è comune a tutti gli oggetti che possono venir creati dal `Creator` e le sue sottoclassi;
- **Concrete Products:** le differenti implementazioni dell'interfaccia `Product`.

8.5 Esempio

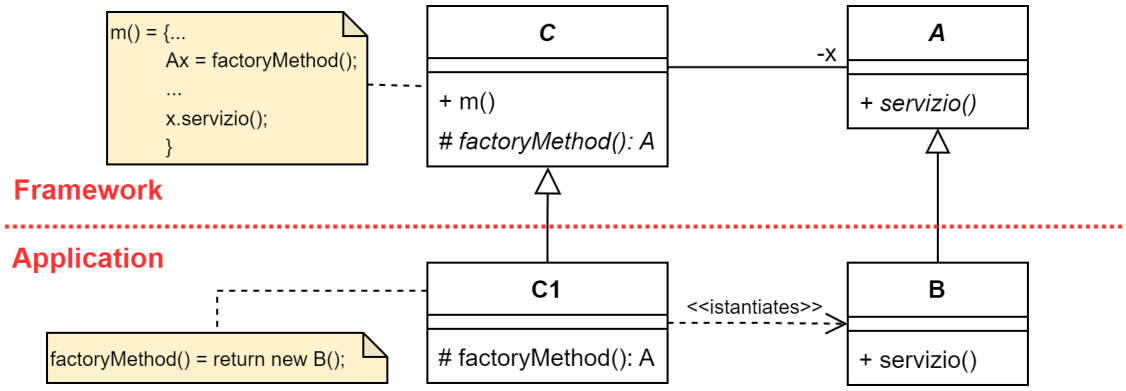
Si analizzerà un esempio tipico, relativo ai framework. Il programmatore del framework scrive un metodo in una classe `C` che deve creare, in un certo punto preciso, un'istanaza di una sottoclasse `B` di un'altra classe astratta `A`.

Quale sottoclasse (`B`) utilizzare è ignoto, perché non deciso dal programmatore del framework ma dal programmatore dell'applicazione (che fa uso del framework). `B` viene addirittura scritta dopo, se il framework è di tipo white-box.



Soluzioni:

- Il programmatore del framework delega tutto al programmatore dell'applicazione... non è una soluzione;
- Il programmatore del framework scrive quello che può sulla base delle informazioni a lui fornite: un'invocazione a un metodo astratto `factoryMethod()` che deve creare l'istanza, e chiede al programmatore dell'applicazione di scrivere, oltre alla classe `B`, anche una sottoclasse `C1` di `C` che implementi il metodo `factoryMethod()`.



Così facendo, invece di avere la conoscenza esplicita sulla creazione di una classe (`C`), la delego a una sottoclasse `C1`. Si noti come il `factoryMethod()` potrebbe non essere astratto ma fornire un'implementazione di default (eventualmente sovrascritta nelle sottoclassi).

Esempio di implementazione:

```
public interface IPerson
{
    string GetName();
}

public class Villager : IPerson
{
    public string GetName()
    {
        return "Village Person";
    }
}

public class CityPerson : IPerson
{
    public string GetName()
    {
        return "City Person";
    }
}

public class IslandPerson : IPerson
{
  public string GetName()
  {
      return "Island Person";
  }
}
```

Abbiamo la gerarchia sopra descritta, tre tipologie di `IPerson`. Nel codice del cliente, senza uso di un factory method, dovremmo distinguere la tipologia di persona, prima di chiamare il costruttore corretto, come ad esempio:

```
public enum PersonType{Rural, Urban, Island}

public static void main(String args[]){
  ...
  PersonType type = getTypeFromUser();
  IPerson newPerson;

  if(type == PersonType.Rural){
     newPerson = new Villager();
  } else if(type == PersonType.Urban){
     newPerson = new CityPerson();
  } else if(type == PersonType.Island){
     newPerson = new IslandPerson();
  }}
```

Questa implementazione è chiaramente non ideale. Con l'uso di un factory method (`getPerson()`), il codice del cliente diventerebbe il seguente:

```
public enum PersonType{Rural, Urban, Island}

public class Factory
{
    public IPerson GetPerson(PersonType type)
    {
        switch (type)
        {
            case PersonType.Rural:
                return new Villager();
            case PersonType.Urban:
                return new CityPerson();
            default:
                throw new NotSupportedException();
        }
    }
}

public static void main(String args[]){
  ...
  PersonType type = getTypeFromUser();
  IPerson newPerson;

  Factory personFactory = new Factory();

  newPerson = personFactory.GetPerson(type);
}
```

In base al tipo passato all'oggetto `Factory`, stiamo restituendo l'oggetto concreto corretto. Qui, la scelta di quale sottoclasse utilizzare è delegata al factory method `getPerson()`.

L'aggiunta di nuove tipologie di persone, oltre che essere più semplice (è necessario estendere l'enumeratore e il factory method e basta, invece che tutti i controlli richiesti in n sezioni di codice dove è richiesta una `IPerson`).

Il codice del `main()` (e qualsiasi porzione di codice che crea istanze di `IPerson`) rimarrebbe invariato in caso di aggiunta/rimozione di sottoclasi di `IPerson`.

9 Abstract Factory

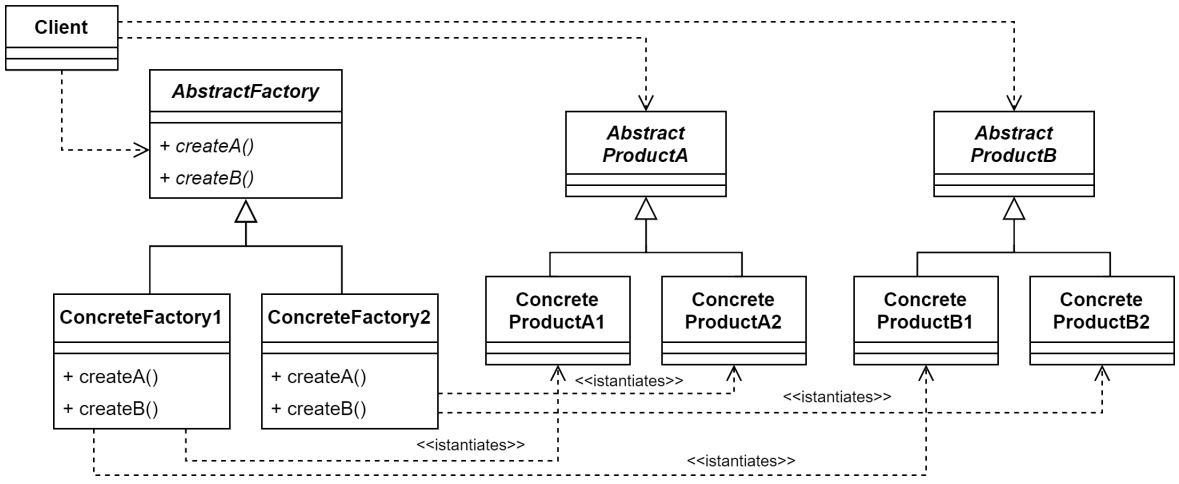
9.1 Scopo

Il pattern fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro, in modo che non ci sia necessità da parte dei client di specificare i nomi delle classi concrete all'interno del proprio codice. In questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti. Rispetto al factory method, qui l'enfasi è sulle famiglie di oggetti correlati.

9.2 Casi d'uso

- Si vuole rendere il sistema/client indipendente da come gli oggetti vengono creati, composti e rappresentati;
- Si vuole permettere la configurazione del sistema come scelta tra diverse famiglie di oggetti/prodotti;
- Si vuole che gli oggetti organizzati in famiglie siano vincolati ad essere utilizzati con oggetti della stessa famiglia;
- Si vuole fornire una libreria di classi mostrando solo le interfacce e nascondendo le implementazioni.

9.3 Diagramma UML



9.4 Partecipanti

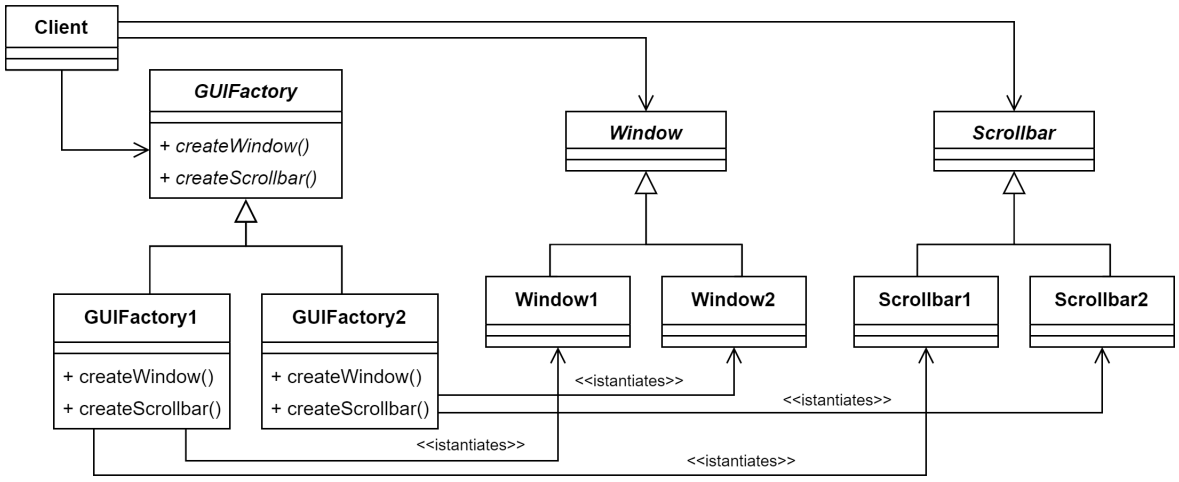
- **AbstractFactory:** dichiara l'interfaccia per le operazioni che creano gli oggetti;
- **ConcreteFactory:** implementa le operazioni per creare gli oggetti concreti;
- **AbstractProduct:** dichiara l'interfaccia per la famiglia di oggetti producibili;
- **ConcreteProduct:** implementa l'interfaccia astratta `AbstractProduct` e definisce l'oggetto prodotto che deve essere creato dalla factory concreta corrispondente;
- **Client:** utilizza solo le interfacce dichiarate `AbstractFactory` e `AbstractProduct` per gestire una famiglia di prodotti.

9.5 Esempio

Stiamo progettando un framework per interfacce grafiche multipiattaforma. Abbiamo la seguente situazione:

- se eseguito su `architettura1`, abbiamo `Window1` e `ScrollBar1`;
- se eseguito su `architettura2`, abbiamo `Window2` e `ScrollBar2`;
- ... e cosi via.

Indipendentemente dalla piattaforma, il client (che istanziaerà quelle classi) non deve sapere quali classi istanzia; bisogna evitare, ad esempio, che il client sbagli e accoppi `Window1` e `ScrollBar2`. Il client deve poter astrarre dalla singola architettura.



Senza fabbrica astratta abbiamo:

```
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
```

Con la fabbrica astratta abbiamo:

```
Factory f = new GUIFactory1();
...
Window w = f.createWindow();
...
Scrollbar s = f.createScrollbar();
```

Senza Abstract Factory	Con Abstract Factory
Client deve conoscere Window1 e ScrollBar1	Client chiede una fabbrica "di tipo 1"
Client crea una Window1	Client chiede alla fabbrica una window
Client crea una ScrollBar1	Client chiede alla fabbrica una ScrollBar
Client ha la responsabilità di accoppiare Window1 e ScrollBar1	La responsabilità di accoppiare Window1 e ScrollBar1 è delegata alla fabbrica

Si ottiene seperazione delle responsabilità: si disaccoppia la responsabilità della creazione (nella fabbrica) dalla responsabilità dell'uso (nel client). Si dice "astratta" perché il cliente conosce solo le classi astratte.

Factory Method vs Abstract Factory

Questi due pattern vengono spesso confusi. La differenza sostanziale tra il factory method e l'abstract factory sta nel fatto che, di fatto, il primo è un metodo, il secondo è un oggetto.

Il factory method è un semplice metodo utilizzato per creare oggetti in/di una classe. Tipicamente viene aggiunto nella classe principale.

L'abstract factory, invece, crea una classe base con metodi astratti che definiscono gli oggetti che possono venir creati. Ciascuna factory (concreta) deriva dalla classe base e può creare/avere la propria implementazione per ciascun oggetto.

Il factory method è pensato per creare un oggetto singolo (esponendo al cliente un metodo), l'abstract factory è pensata per creare una famiglia di oggetti dipendenti (esponendo una famiglia di oggetti correlati).

Il factory method nasconde la costruzione/implementazione di un singolo oggetto, l'abstract factory di una intera famiglia di oggetti.

Si noti come l'abstract factory è tipicamente implementata mediante l'uso molteplice del factory method.

L'abstract factory utilizza la composizione per delegare la responsabilità di creare un oggetto mentre il factory method utilizza l'ereditarietà da classe a sottoclasse.

10 Singleton

10.1 Scopo

Fornire la certezza che una classe abbia un'unica istanza e fornire un punto di accesso globale a quest'ultima. Si delegano alla classe le responsabilità di consentire un'unica istanza (impedire creazioni di più istanze) e consentirne l'accesso.

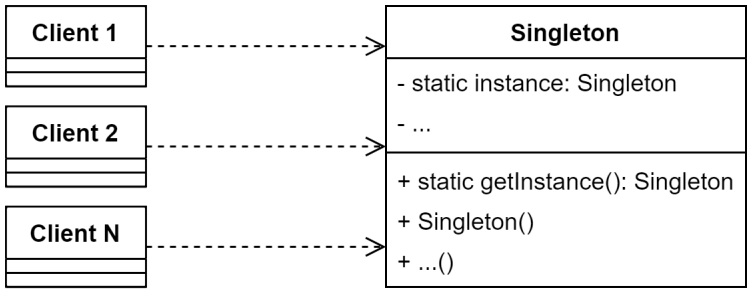
L'implementazione più semplice di questo pattern prevede che la classe singleton abbia un unico costruttore privato, in modo da impedire l'istanziazione diretta della classe. La classe fornisce inoltre un metodo "getter" statico che restituisce l'istanza della classe (sempre la stessa), creandola preventivamente o alla prima chiamata del metodo, e memorizzandone il riferimento in un attributo privato anch'esso statico.

Il secondo approccio si può classificare come basato sul principio della lazy initialization (letteralmente: "inizializzazione pigra") in cui la creazione dell'istanza della classe viene rimandata nel tempo e messa in atto solo quando ciò diventa strettamente necessario (al primo tentativo di uso).

10.2 Casi d'uso

- Ci si vuole assicurare che una certa classe venga istanziata una singola volta (o che in un certo istante abbia una sola istanza attiva);
- Si vuole fornire un accesso globale a un'istanza unica di una classe.

10.3 Diagramma UML



10.4 Partecipanti

- **Singleton:** la classe unica durante l'esecuzione;
- **Clienti:** i clienti dei servizi della classe unica `Singleton`.

10.5 Esempio

Il singleton è un pattern molto orientato al codice, vediamo un esempio di implementazione con "lazy initialization":

```
public class MySingleton {
  private static MySingleton istanza = null;

  // Il costruttore private impedisce l'istanza di oggetti
  // da parte di classi esterne
  private MySingleton() {...}

  // Metodo della classe impiegato per accedere al singleton
  public static synchronized MySingleton getMySingleton() {
      if (istanza == null) {
          istanza = new MySingleton();
      }
      return istanza;
  }
}
```

Una versione non lazy (con "early initialization") è:

```
public class MySingleton {
  private static MySingleton istanza = new MySingleton();

  // Il costruttore private impedisce l'istanza di oggetti
  // da parte di classi esterne
  private MySingleton() {...}

  // Metodo della classe impiegato per accedere al singleton
  public static synchronized MySingleton getMySingleton() {
      return istanza;
  }
}
```

Bisogna comunque precisare che questo tipo di approccio risolutivo potrebbe presentare dei difetti (per esempio questa implementazione non è thread safe), infatti il progettista della classe che è chiamato tra l'altro a rispettare il pattern singleton, potrebbe, per esempio, erroneamente dichiarare all'interno della classe un metodo non statico con visibilità public che restituisca un'istanza di un nuovo oggetto della stessa classe, ed ecco che allora il vincolo viene violato in contraddizione a quanto sopra detto.

Qualora questo metodo venga chiamato da più thread e si verifichi una race-condition non opportunamente gestita (tramite meccanismi di sincronizzazione), potremmo avere due o più istanze del singleton.

In definitiva questo significa che in realtà per adempiere pienamente bisogna gestire il tutto con il meccanismo delle eccezioni.

11 Prototype

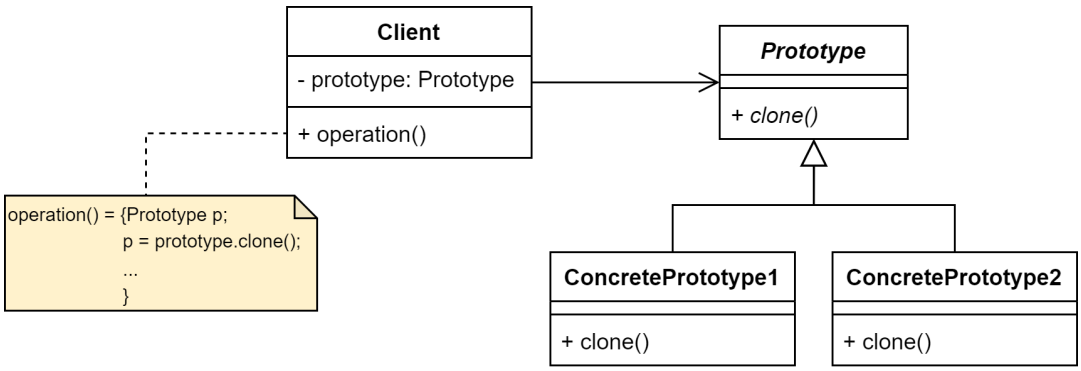
11.1 Scopo

Il fine del pattern è di rendere possibile la creazione di nuovi oggetti clonando un oggetto iniziale, detto prototipo. Si ottiene il vantaggio di nascondere le classi concrete al cliente, che comunica solo con la classe prototipo.

11.2 Casi d'uso

- Le classi da istanziare sono specificate solamente a tempo d'esecuzione, per cui un codice statico non può occuparsi della creazione dell'oggetto;
- Si vuole evitare di costruire una gerarchia di factory in parallelo a una gerarchia di prodotti, come avviene utilizzando abstract factory e factory method;
- Quando le istanze di una classe possono avere soltanto un limitato numero di stati, per cui può essere più conveniente clonare al bisogno il prototipo corrispondente piuttosto che creare l'oggetto e configurarlo ogni volta.

11.3 Diagramma UML



11.4 Partecipanti

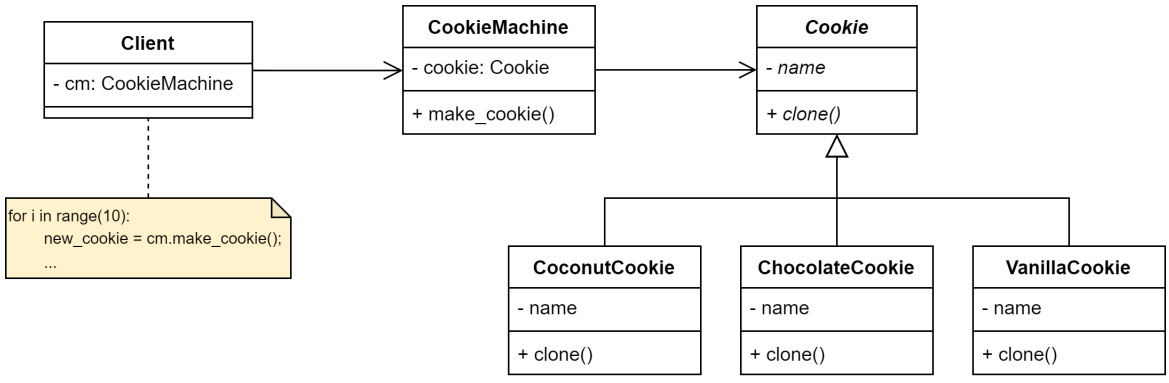
- **Prototype:** definisce un'interfaccia per clonare se stesso;
- **ConcretePrototype:** implementa `Prototype` fornendo un'operazione per clonarsi;
- **Client:** crea oggetti del tipo desiderato chiedendo a un prototipo concreto di clonarsi qualora necessario.

11.5 Esempio

Un esempio di uso del prototype pattern in python.

```
class Cookie:   # Classe Prototype
    def __init__(self, name):
        self.name = name

    def clone(self):
        return copy.deepcopy(self)


class CoconutCookie(Cookie):   # Protitpi concreti
    def __init__(self):
        Cookie.__init__(self, 'Coconut')

class ChocolateCookie(Cookie):
    def __init__(self):
        Cookie.__init__(self, 'Choco')

class VanillaCookie(Cookie):
    def __init__(self):
        Cookie.__init__(self, 'Vanilla')


class CookieMachine:   # Classe cliente
    def __init__(self, cookie):
        self.cookie = cookie

    def make_cookie(self):
        return self.cookie.clone()

if __name__ == '__main__':
    prototype = CoconutCookie()
    cm = CookieMachine(prototype)

    for i in xrange(10):
        # Uso del pattern, chiamata a clone()
        temp_cookie = cm.make_cookie()
```

Il diagramma UML relativo all'esempio:



12 Builder

12.1 Scopo

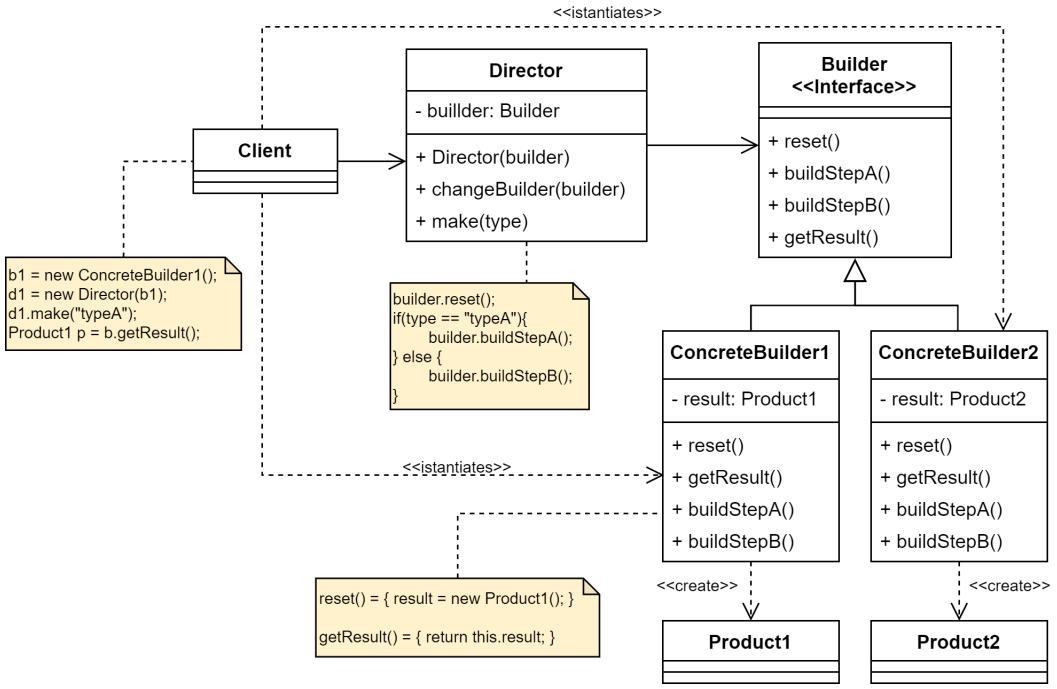
Separare la costruzione di un oggetto complesso dalla sua rappresentazione cosicchè il processo stesso possa creare diverse rappresentazioni. L'algoritmo per la creazione di un oggetto complesso diventa indipendente dalle varie parti che costituiscono l'oggetto e da come vengono assemblate. Ciò ha l'effetto di rendere più semplice la classe, permettendo a una classe builder separata di focalizzarsi sulla costruzione di un'istanza e lasciando che la classe originale si concentri sul funzionamento degli oggetti. Mentre il costruttore di una classe specifica la logica della costruzione di un'istanza, il builder delega il codice per la creazione delle istanze al di fuori della classe da istanziare, non più contenuta nel costruttore.

Il funzionamento è il seguente: il client crea l'oggetto director e lo configura per farlo operare con il builder; il director informa il builder ogni volta che una parte del product deve essere costruita; il builder riceve le richieste dal director e aggiunge le parti al product. Alla fine, il client recupera il product creato. Rispetto ad altri pattern creazionali, nel builder il focus è sul processo di costruzione; product è costruito non in una sola operazione, ma passo dopo passo sotto il controllo del director.

12.2 Casi d'uso

- Si vuole separare la logica di costruzione dalla logica di controllo di un oggetto;
- Si vuole permettere la creazione passo passo di un oggetto complesso.

12.3 Diagramma UML

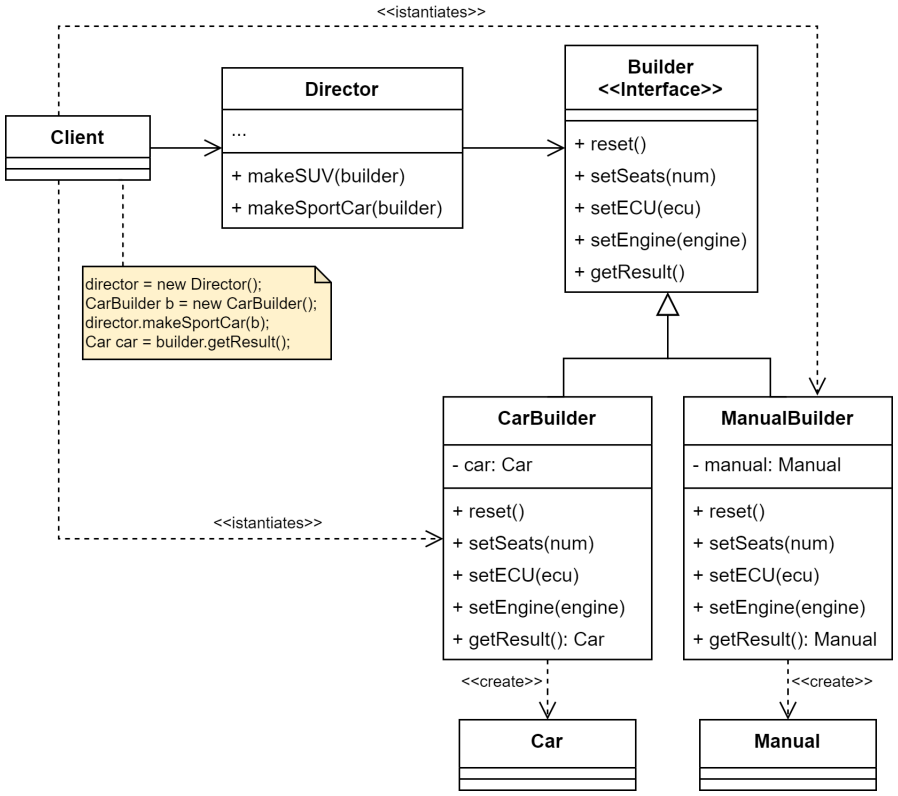


12.4 Partecipanti

- **Builder:** specifica la classe astratta che crea le parti dell'oggetto `Product` e i vari passi necessari;
- **ConcreteBuilder:** costruisce e assembla le parti del prodotto implementando l'interfaccia `Builder`; definisce e tiene traccia della rappresentazione che crea;
- **Director:** costruisce un oggetto utilizzando l'interfaccia `Builder` e definisce l'ordine dei passi di costruzione;
- **Product:** rappresenta l'oggetto complesso e include le classi che definiscono le parti che lo compongono, includendo le interfacce per assemblarle in un oggetto finale.

12.5 Esempio

Nel seguente esempio di uso del builder viene illustrato un approccio alla creazione di differenti tipologie di auto con relativo manuale.



Un oggetto di tipo `Car` è un oggetto complesso che può essere costruito (si immagini una gerarchia più estesa) in molti modi diversi. Invece di rendere complessa la classe `Car` e il suo costruttore, facciamo uso di un builder. `Client` delega la costruzione passo passo della corretta tipologia di `Car` al `Director`. Per ciascuna auto è necessario anche un manuale (`Manual`); a tal proposito si può riusare il processo di costruzione dell'automobile per creare il corretto manuale per ciascuna tipologia di `Car`.

Segue l'implementazione del diagramma UML precedente:

```
public class Car{
  public Engine engine;
  public int numSeats;
  public ECU ecu;
  ...

  public Car() {...}
}

public class Manual{
  ...

  public Manual() {...}
}

public interface Builder{
      public void reset()
      public void setSeats(...){...}
      public void setEngine(...){...}
      public void setECU(...){...}
}


public class CarBuilder implements Builder{
      private Car car;

      public CarBuilder() {
          this.reset();
      }

      public void reset(){
          this.car = new Car();
      }

      public void setSeats(...){...}

      public void setEngine(...){...}

      public void setECU(...){...}

      public Car getProduct(){
          product = this.car;
          this.reset();
          return product;
      }
}

public class CarManualBuilder implements Builder{
      private field manual:Manual

      public CarManualBuilder(){
          this.reset();
```

```
53        }
54
55        public void reset(){
56            this.manual = new Manual();
57        }
58
59        public void setSeats(...){...}
60
61        public void setEngine(...){...}
62
63        public void setECU(...){...}
64
65        public Manual getProduct(){
66          product = this.manual;
67          this.reset();
68          return product;
69        }
70  }
71
72  public class Director{
73        private Builder builder;
74
75        public void setBuilder(builder:Builder){
76            this.builder = builder;
77        }
78
79        public void constructSportsCar(builder: Builder){
80            builder.reset();
81            builder.setSeats(2);
82            builder.setEngine(new SportEngine());
83            builder.ECU(new SportECU());
84        }
85
86        public void constructSUV(builder: Builder){...}
87  }
88
89
90  public class Application{
91      Director director = new Director();
92
93      CarBuilder builder = new CarBuilder();
94      director.constructSportsCar(builder);
95      Car car = builder.getProduct();
96
97      CarManualBuilder builder = new CarManualBuilder();
98      director.constructSportsCar(builder);
99
100     Manual manual = builder.getProduct();
101 }
```

13 Template method

13.1 Scopo

Si vuole definire/incapsulare la struttura di un algoritmo all'interno di un metodo, lasciandone alcune parti non specificate.

L'implementazione di tali parti non specificate è contenuta in altri metodi, la cui implementazione è delegata alle sottoclassi, che, ridefiniscono solo alcuni passi dell'algoritmo, lasciando invariata la struttura generale.

In questo modo si può ridefinire e personalizzare parte del comportamento nelle varie sottoclassi senza dover riscrivere più volte il codice in comune.

L'idea del template method è simile a quella del factory method, ma sono diversi. In una classe, metodi astratti vengono invocati da un altro metodo e specificati nelle sottoclasi (metodi "gancio/hook" a cui agganciare un'implementazione specifica); ma:

- Nel template (comportamentale) è il metodo non astratto che invoca i metodi astratti;
- Il factory method (creazionale) è un metodo astratto che deve creare e restituire un'istanza.

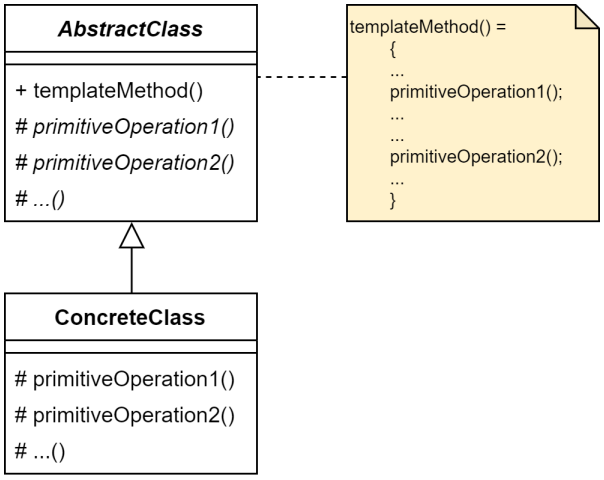
Il pattern in un certo senso ribalta il meccanismo dell'ereditarietà secondo quello che viene scherzosamente chiamato "principio di Hollywood": non chiamarci, ti chiameremo noi.

Normalmente sono le sottoclassi a chiamare i metodi delle classi genitrici; in questo pattern è il metodo template, appartenente alla classe genitrice, a chiamare i metodi specifici ridefiniti nelle sottoclassi.

13.2 Casi d'uso

- Si vuole implementare la parte invariante di un algoritmo una volta sola e lasciare che le sottoclassi implementino il comportamento che può variare;
- Il comportamento comune di più classi può essere fattorizzato in una classe a parte per evitare di scrivere più volte lo stesso codice;
- Poter controllare come le sottoclassi ereditano dalla superclasse, facendo in modo che i metodi template chiamino dei metodi "gancio"; unici metodi sovrascrivibili.

13.3 Diagramma UML



13.4 Partecipanti

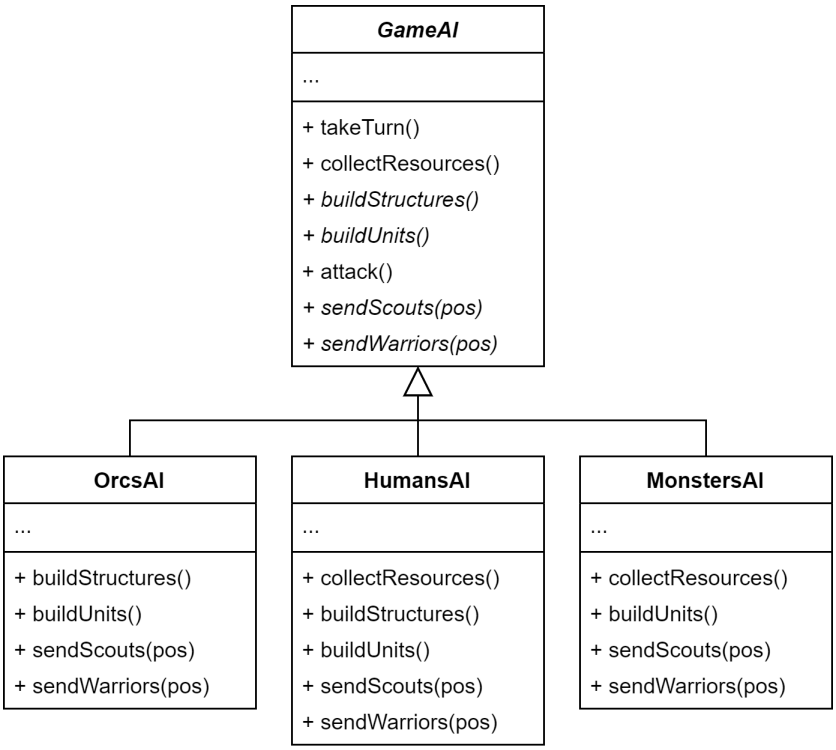
- **AbstractClass:** definisce le operazioni primitive astratte che le classi concrete sottostanti andranno ad ampliare ed implementa il metodo template (`templateMethod()`) che rappresenta la parte invariante (in comune) dell'algoritmo;
- **ConcreteClass:** implementa le operazioni primitive astratte che eredita dalla superclasse, specificando così il comportamento per i passi variabili dell'algoritmo.

13.5 Esempio

Nell'esempio sottostante, il template method fornisce uno scheletro per varie tipologie di AI in un videogioco di strategia. Tutte le razze del gioco hanno quasi gli stessi tipi di unità ed edifici. Con l'uso del pattern si può riutilizzare la stessa struttura dell'AI per varie razze, rendendo anche possibile sovrascrivere alcuni dettagli comportamentali.

Con questo approccio, si può sovrascrivere l'AI degli orchi per renderla più aggressiva, rendere gli umani più orientati alla difesa e rendere i mostri incapaci di costruire, etc.

L'aggiunta di una nuova razza al gioco richiederebbe la creazione di una nuova sottoclasse AI e l'override dei metodi predefiniti dichiarati nella classe astratta AI di base (`GameAI`), compito meno oneroso rispetto alla reimplementazione from scratch, che porterebbe anche a replicazione di codice per le parti comuni.



Il metodo `takeTurn()` è il template method, che per ciascuna razza, in ciascun turno compie n azioni, come collezionare risorse, inviare unità, etc. Ciascuna di queste azioni è implementata diversamente per ogni variante di razza/AI. Quindi, `takeTurn()` ha una struttura comune alle varie AI, ma ciascun passo di esso può essere implementato diversamente per ciascuna classe (razza).

14 Strategy

14.1 Scopo

L'obiettivo è di isolare un algoritmo all'interno di un oggetto, in maniera tale da risultare funzionale in quelle situazione in cui sia necessario modificare dinamicamente gli algoritmi utilizzati da un'applicazione (client).

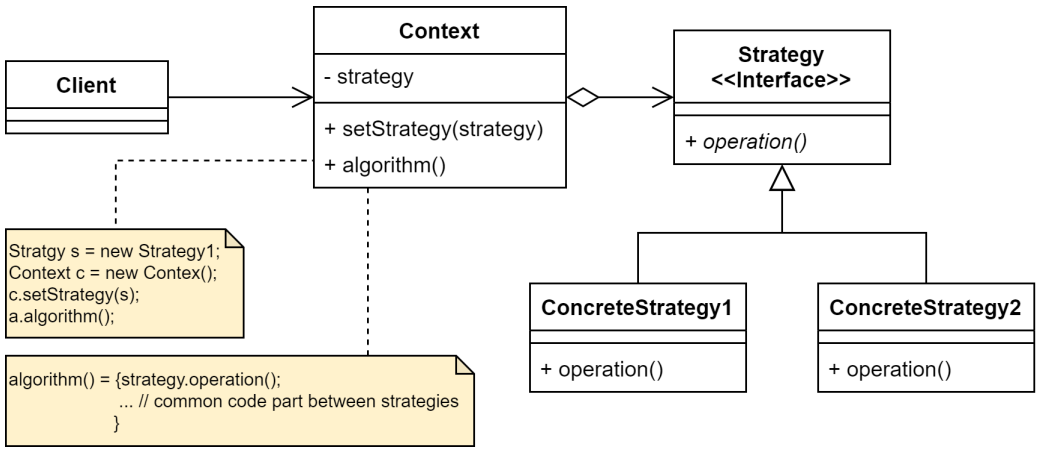
Si pensi ad esempio alle possibili visite in una struttura ad albero (visita anticipata, simmetrica, posticipata, etc.); mediante il pattern strategy è possibile selezionare a run-time una tra le tipologie di visita ed eseguirla sull'albero per ottenere il risultato voluto.

A differenza del template method che fa uso dell'ereditarietà, lo strategy pattern fa uso della composizione (permette variazioni anche al run-time).

14.2 Casi d'uso

- Si vuole poter selezionare a run-time un algoritmo (o un comportamento/sotto-algoritmo facente parte di una logica più complessa);
- Si vuole isolare i dettagli implementativi di un algoritmo dal codice (client) che ne fa utilizzo;
- Si vuole strutturare in modo più pulito (usabilità/modificabilità) un insieme di classi molto simili che differiscono solo nel modo in cui eseguono alcuni comportamenti.

14.3 Diagramma UML



14.4 Partecipanti

- **Context:** mantiene una referenza a una delle strategie concrete e comunica con tale oggetto tramite l'interfaccia `Strategy`; espone un setter per permettere al client di cambiare strategia;

- **Client:** esegue le sue operazioni, specificando una strategia qualora necessario;
- **Strategy:** definisce un'interfaccia comune a tutte le strategie concrete. Dichiara un metodo che il `Context` utilizza per eseguire la strategia;
- **ConcreteStrategy:** implementa diverse versioni dell'interfaccia `Strategy`, implementando di fatto diversi approcci risolutivi (varianti) dell'algoritmo/problema.

14.5 Esempio

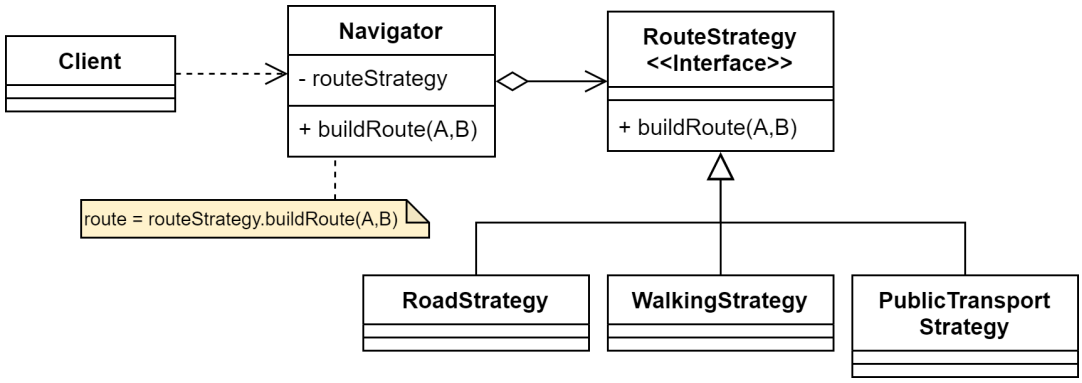
Si pensi di aver scritto un'app mobile per viaggi, focalizzata sul fornire una mappa interattiva che aiuta gli utenti a orientarsi. La feature principale consiste nella possibilità di visualizzare il percorso più veloce tra due punti ("route-planning").

Nella prima versione dell'app era possibile visualizzare solo percorsi in auto; nelle versioni successive la funzionalità è stata estesa per le bici, poi per i pedoni, poi per i trasporti pubblici, e così via.

Dal punto di vista aziendale l'app è un successo, la parte tecnica però ha accumulato del debito tecnico; presenta dei problemi: ogni estensione dell'algoritmo di routing ha reso il codice sempre più di scarsa qualità e ha esteso la classe principale del navigatore, fino a renderla ingestibile e troppo complessa.

Una possibile soluzione è l'uso del strategy pattern: prendere una classe che svolge qualcosa di specifico in molti modi diversi (la classe `Navigator`) ed estrarre tutti gli algoritmi (le varianti di routing) in classi separate chiamate strategie. La classe originale (denominata contesto) deve avere un campo per memorizzare il riferimento a una delle strategie. Il contesto delega il lavoro all'oggetto strategia selezionato.

Non è il contesto a scegliere la strategia specifica da utilizzare, ma il cliente. Nel nostro esempio, avremmo:



Nella nostra app di navigazione, ora ogni algoritmo di routing può essere estratto nella propria classe con un unico metodo `buildRoute()`. Il metodo accetta un punto di partenza A e una destinazione B e restituisce una lista di checkpoint del percorso.

Ogni classe di routing potrebbe costruire una rotta diversa. Alla classe principale del navigatore non interessa quale algoritmo sia selezionato poiché il suo compito principale è quello di eseguire il rendering di un insieme di checkpoint sulla mappa. La classe ha un metodo per cambiare la strategia di routing attiva.

15 State

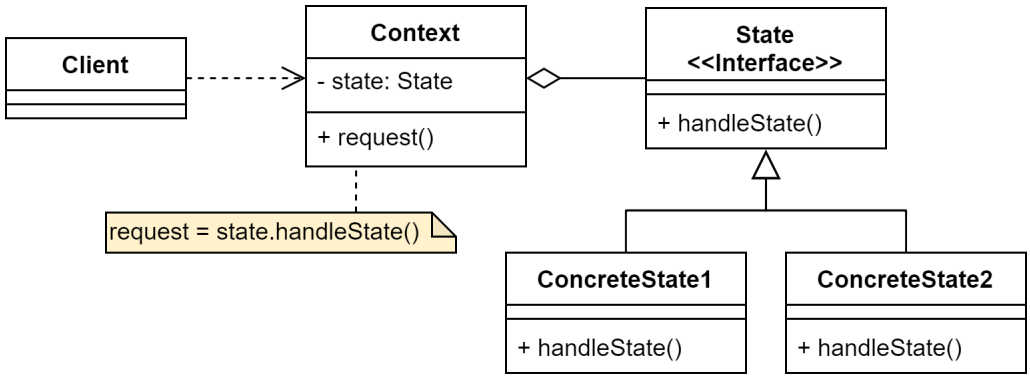
15.1 Scopo

Permettere ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello specifico stato in cui si trova.

15.2 Casi d'uso

- Un oggetto deve cambiare il suo comportamento in relazione al suo stato interno;
- I comportamenti relativi a ciascun stato devono poter essere definiti in modo indipendente;
- Una classe è inquinata da molti controlli condizionali che alterano il comportamento della classe in relazione ai valori dei parametri a run-time;
- Aggiungere un nuovo stato non deve influenzare il comportamento degli stati presenti.

15.3 Diagramma UML

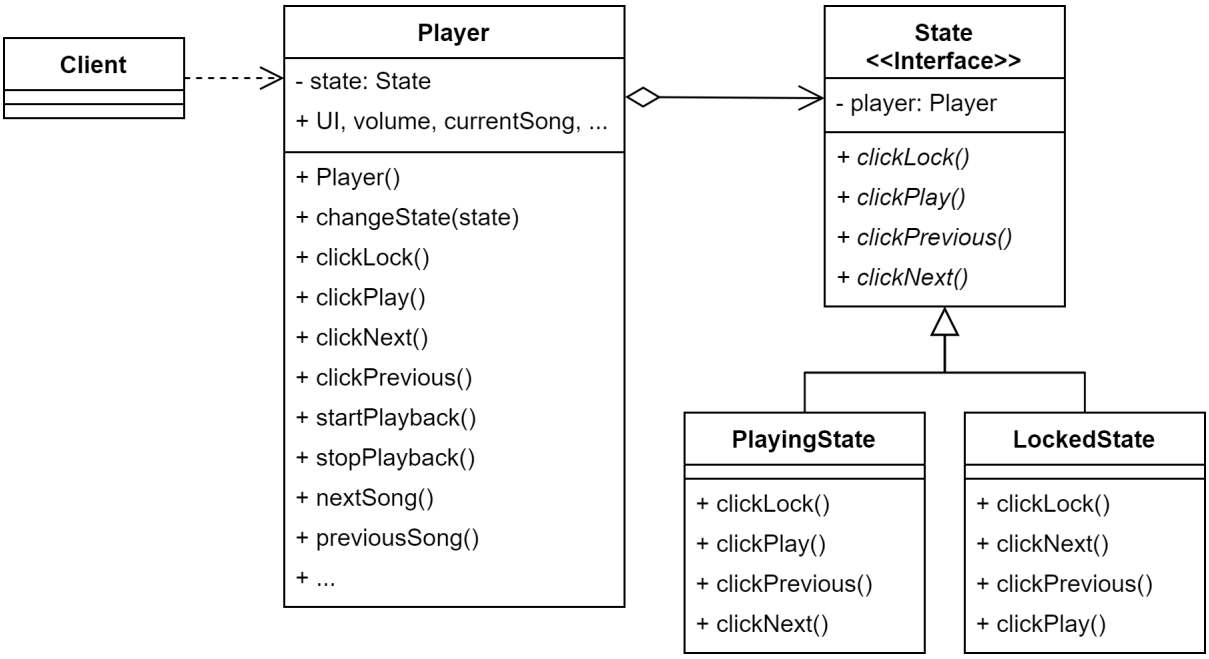


15.4 Partecipanti

- **Context:** definisce la classe utilizzata dal client e mantiene un riferimento ad un `ConcreteState`;
- **State:** definisce l'interfaccia, implementata dai `ConcreteState`, che incapsula la logica del comportamento associato ad un determinato stato;
- **ConcreteState:** implementa il comportamento associato ad un particolare stato.

15.5 Esempio

Si pensi di voler controllare un media player, che si comporta (per alcune azioni) diversamente in base allo stato corrente. Un esempio di applicazione del pattern potrebbe essere il seguente:



L'istanza di `Player` è sempre collegata a un'istanza di `State` che esegue la maggior parte del lavoro. Alcune azioni alterano lo stato del player con un altro, il che cambia il modo in cui il player reagirà ad alcune interazioni dell'utente.

16 Command

16.1 Scopo

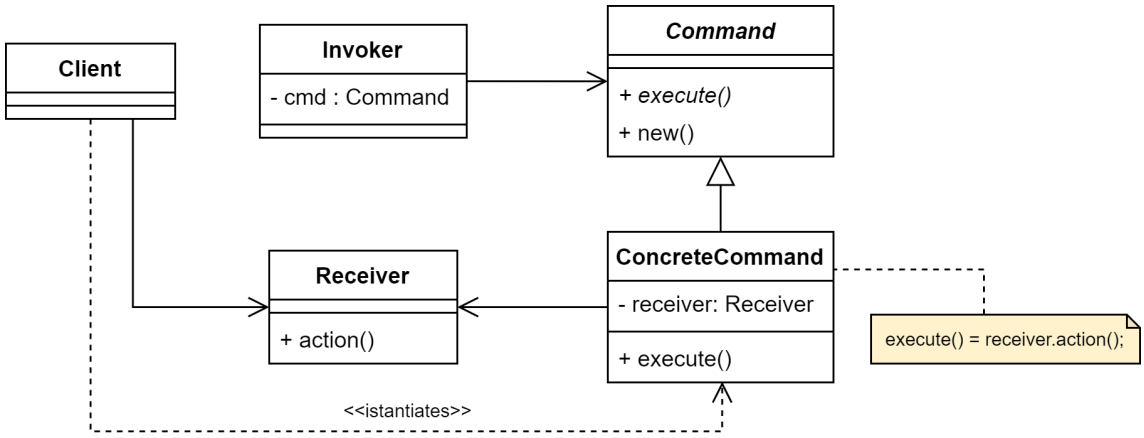
Permettere di isolare la porzione di codice che effettua un'azione (eventualmente molto complessa) dal codice che ne richiede l'esecuzione.

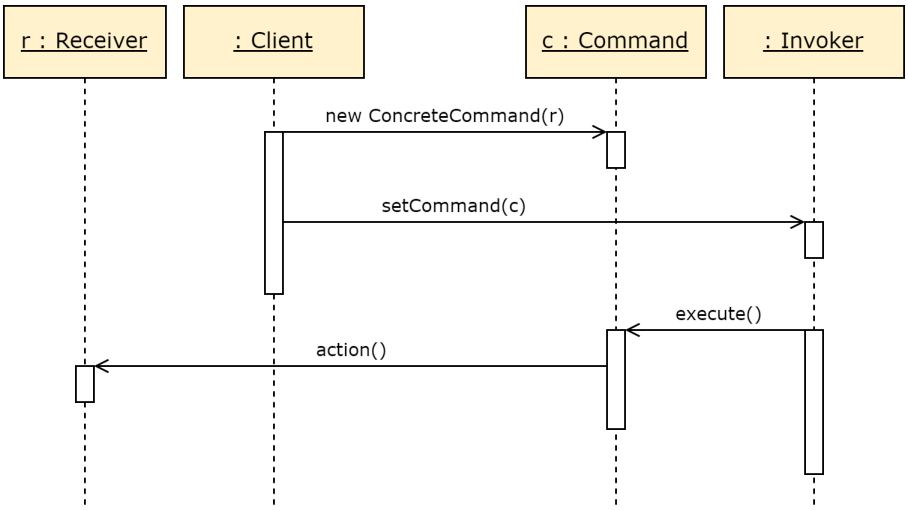
Il codice dell'azione/operazione è incapsulato all'interno di un oggetto command. Si mira a rendere variabile l'azione del client senza però conoscere i dettagli dell'operazione stessa. L'azione può non essere decisa staticamente ma bensì ricavata/scelta a run-time.

16.2 Casi d'uso

- Si vuole maggiore libertà nella scelta a run-time di una specifica operazione/azione da compiere in base al contesto;
- Si vuole incapsulare il codice al fine di nascondere l'implementazione di alcune azioni complesse;
- Le azioni diventano "gestibili"; non essendo righe di codice ma oggetti. Si ottiene maggiore libertà, esse possono essere memorizzate, passate come argomento, etc.

16.3 Diagramma UML





16.4 Partecipanti

- **Invoker:** memorizza un riferimento astratto al comando (istanza di `ConcreteCommand`, sottoclasse di `Command`) e ne invoca l'`execute()`;
- **Command:** definisce un'interfaccia per i singoli comandi "resi oggetti";
- **Receiver:** riceve l'invocazione di `action()` dal `ConcreteCommand`;
- **ConcreteCommand:** implementa l'interfaccia di `Command`, comune a tutti i comandi; invoca action di `Receiver` quando invocato dall'`Invoker`;
- **Client:** crea istanze di `ConcreteCommand` e le passa al `Receiver` per utilizzarle; svolge la logica di business tramite i singoli comandi.

16.5 Esempio

Analizziamo il codice del client con e senza command pattern:

Senza l'uso del command pattern avremmo:

```
1  Receiver r = new Receiver();
2  r.action();
```

Con l'uso del command pattern avremmo, nel client:

```
1  Receiver r = new Receiver();
2  Command c = new ConcreteCommand(r);
```

Sempre nel client (o chiunque notifichi all'invoker):

```
1  Invoker i = new Invoker();
2  i.setCommand(c);
```

Nell'Invoker:

```
1  c.execute();
```

Il command permette di disaccoppiare il client (che crea il comando/richiesta) e receiver (che esegue) tramite l'invoker (che invoca la richiesta al momento opportuno).

I comandi sono oggetti e non invocazioni, è possibile quindi comporre più comandi in un comando composto (ad esempio tramite il composite pattern). È più semplice aggiungere nuovi comandi, è possibile memorizzare uno storico dei comandi per l'esecuzione dell'undo, etc.

17 Observer

17.1 Scopo

Permettere di osservare lo stato di un oggetto tramite degli oggetti che vengono notificati ed aggiornati ad ogni variazione.

Un osservatore, in realtà, non "osserva" ma rimane in attesa e aspetta che gli venga notificato di controllare ("guardare") l'oggetto di riferimento. Viene tipicamente utilizzato nel paradigma di programmazione ad eventi.

Il funzionamento è il seguente: gli osservatori si "registrano/abbonano" presso l'oggetto osservato che li notificherà ogni qual volta cambierà stato, tramite l'invocazione di un metodo ("meccanismo di callback").

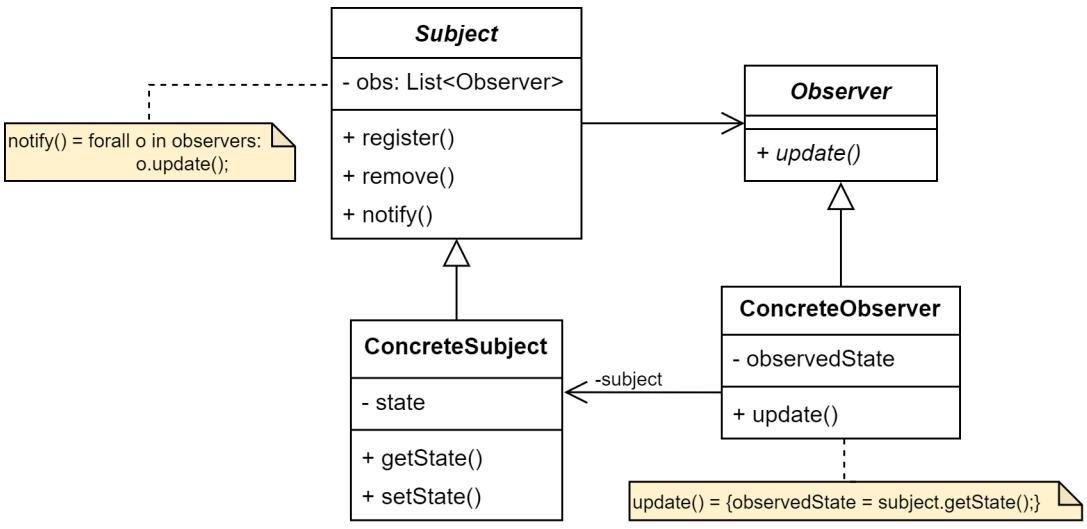
Quando un osservatore viene notificato, decide cosa fare (niente; richiedere all'osservato informazioni sul nuovo stato, etc.).

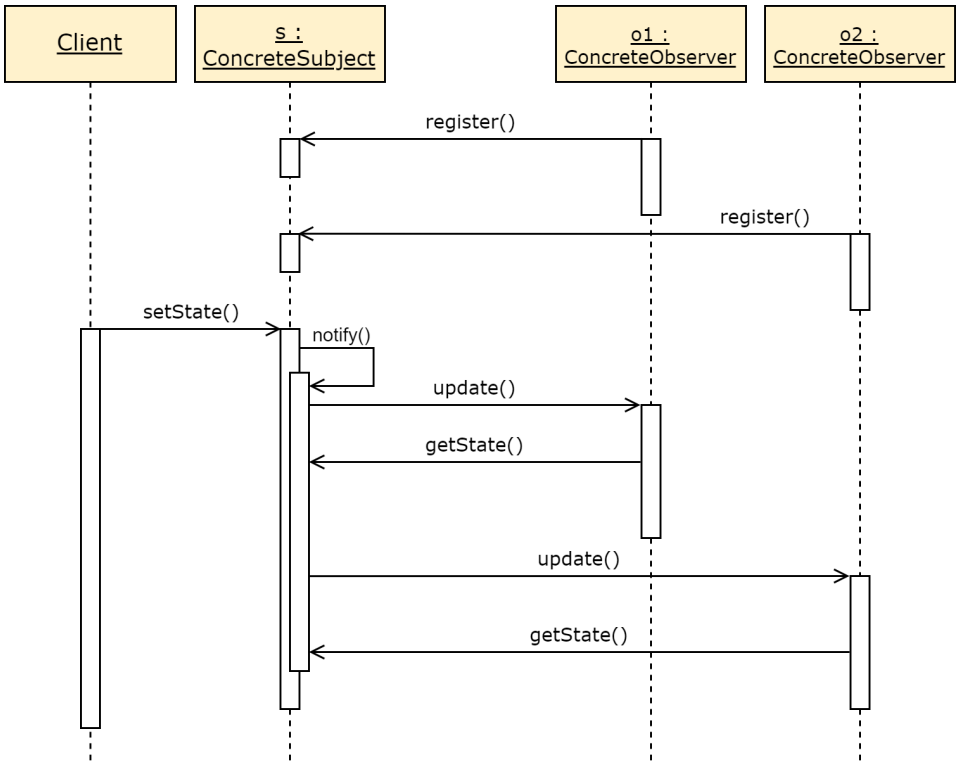
Gli osservatori possono aggiungersi/rimuoversi dalla lista degli "abbonati" a run-time. L'oggetto osservato non conosce il tipo concreto degli osservatori ma gli basta conoscere una superclasse/interfaccia da essi condivisa.

17.2 Casi d'uso

- Si vuole monitorare lo stato di un oggetto e reagire ai cambiamenti;
- Si vuole definire una dipendenza uno-a-molti senza rendere gli oggetti strettamente accoppiati;
- Si vuole aggiornare in modo automatico una serie di oggetti al cambiamento di stato di un'altro oggetto;

17.3 Diagramma UML





17.4 Partecipanti

- **Subject:** fornisce un'interfaccia per registrare, rimuovere o notificare gli observer;
- **ConcreteSubject:** fornisce lo stato dei soggetti concreti agli observer e si occupa di notitificare gli observer registrati invocando la funzione `update()`;
- **Observer:** definisce un'interfaccia per tutti gli observers, per ricevere le notifiche dal soggetto osservato. È utilizzata come classe astratta per implementare i veri `Observer`, ossia i `ConcreteObserver`;
- **ConcreteObserver:** mantiene un riferimento al soggetto concreto (osservato) per riceverne lo stato quando notificato. Il `ConcreteObserver` implementa la funzione astratta `update()`: quando viene chiamata dal soggetto concreto, il `ConcreteObserver` chiama la funzione `getState()` sul soggetto concreto per aggiornare il suo nuovo stato.

17.5 Esempio

Un esempio di implementazione generica in python:

```
class Observable:
    def __init__(self):
        self._observers = []

    def register_observer(self, observer):
        self._observers.append(observer)

    def notify_observers(self, *args, **kwargs):
        for obs in self._observers:
            obs.notify(self, *args, **kwargs)


class Observer:
    def __init__(self, observable):
        observable.register_observer(self)

    def notify(self, observable, *args, **kwargs):
        print("Received", args, kwargs, "From", observable)


subject = Observable()
observer = Observer(subject1)
subject.notify_observers("test", kw="python")

# -- Output:
# prints: Got ('test',) {'kw': 'python'} From <__main__.Observable object
```

18 Mediator

18.1 Scopo

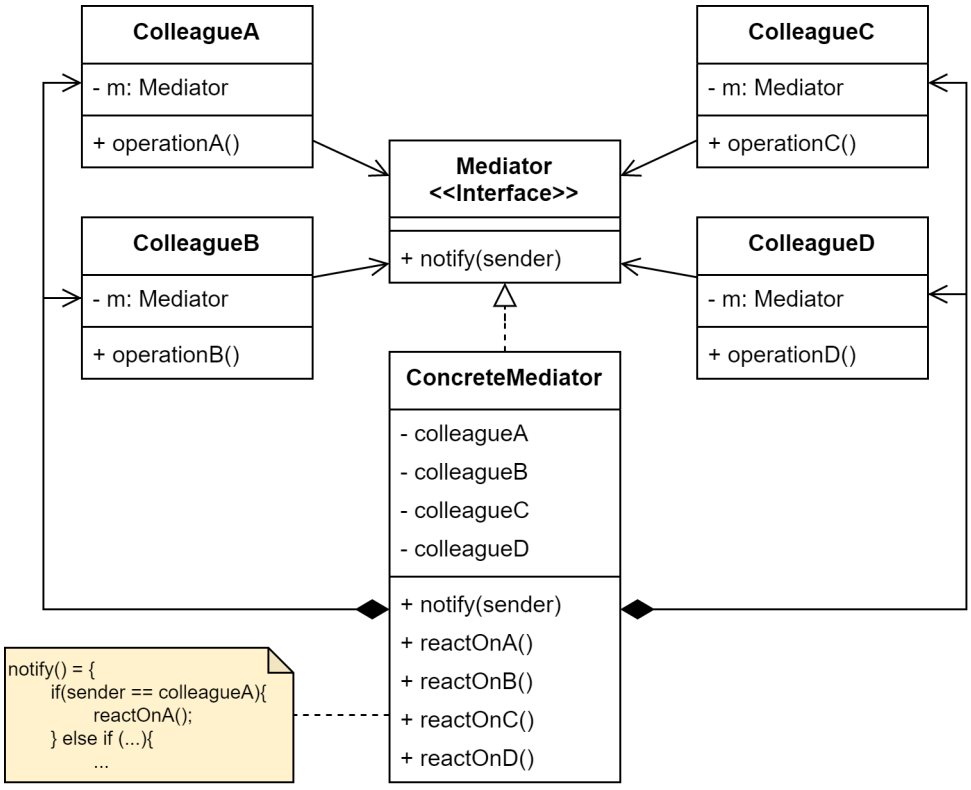
Lo scopo del mediator è di permettere di modificare agilmente le politiche di interazioni tra oggetti, poiché le entità coinvolte devono far riferimento a un solo oggetto (il mediatore).

Senza mediator, tante classi hanno alto accoppiamento, in quanto ciascuna di esse deve conoscere i metodi di molte altre classi. Cala la comprensibilità e il riuso del codice. Con il mediator si abbassa l'accoppiamento e si facilitano i cambiamenti nella logica di interazione (in quanto localizzata in una singola classe). Le classi singole si semplificano e migliora il riuso e la chiarezza del codice.

18.2 Casi d'uso

- Si vuole maggior controllo e un punto specifico dove specificare le modalità di interazioni tra oggetti molteplici;
- Si vuole nasconodere ai singoli oggetti i "fratelli" con cui interagiscono.

18.3 Diagramma UML



Si noti come i singoli `Colleague` potrebbero essere riorganizzati gerarchicamente in una classe astratta `Collegue`.

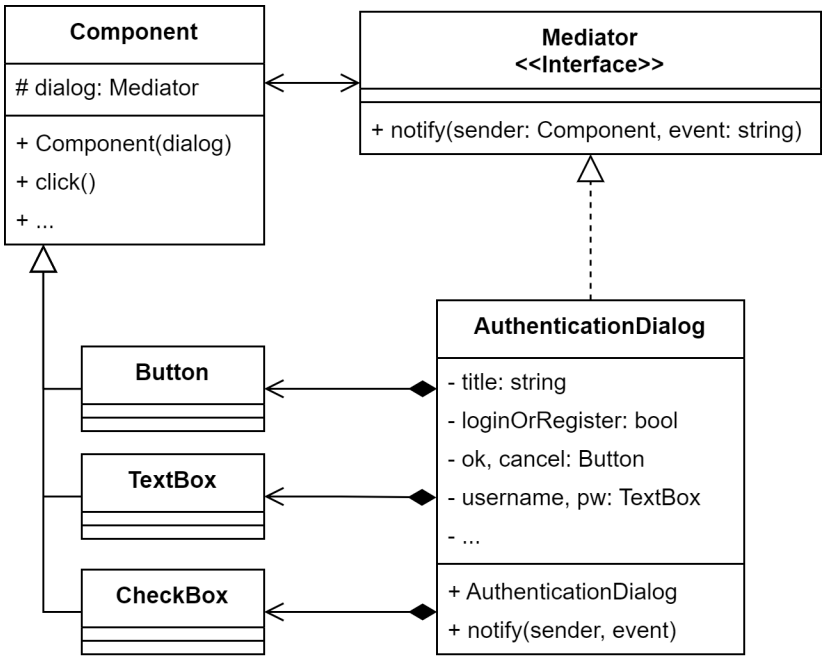
18.4 Partecipanti

- **Mediator:** funge da intermediario fra i vari "colleagues", oggetti dei quali mantiene un riferimento interno. Riceve gli eventi da essi e modifica lo stato del sistema di conseguenza;
- **Colleague:** gli oggetti che interagiscono fra loro mediante il mediator. I vari colleagues possono anche essere sottoclassi di una classe astratta comune.

18.5 Esempio

in questo esempio vediamo come è possibile eliminare le mutue dipendenze tra singoli elementi di una API grafica: UI, bottoni, checkboxes, text labels, etc.

L'applicazione del pattern porterebbe alla seguente gerarchia:



Un elemento attivato/triggerato dall'utente, non comunica con gli altri elementi direttamente, anche se a livello logico dovrebbe. Piuttosto, fa sapere al `Mediator` dell'evento, fornendo informazioni contestuali.

Il mediator (`AuthenticationDialog`) conosce come gli elementi concreti debbano collaborare e facilita la loro comunicazione e cooperazione. Alla ricezione di una notifica, il mediator esegue le operazioni relative e aziona/altera gli altri elementi in modo appropriato.

La logica di interazione è quindi completamente delegata al mediator.

19 Memento

19.1 Scopo

Lo scopo è quello di catturare/estrarre lo stato interno di un oggetto, e di memorizzarlo, in modo da poterlo ripristinare in seguito, senza violare l'incapsulamento.

Tipico esempio è l'operazione di undo, che consente di ripristinare lo stato di uno o più oggetti a come era/erano prima dell'esecuzione di una data operazione.

Il punto chiave di questo pattern è la definizione di un oggetto di tipo memento nel quale verrà immagazzinato lo stato di un oggetto, l'originator. Tale oggetto memento disporrà di una doppia interfaccia:

- Quella verso l'originator, più ampia, che consentirà a questo di salvare il suo stato interno e di ripristinarlo;
- Quella verso gli altri, che esporrà solamente l'eventuale distruttore.

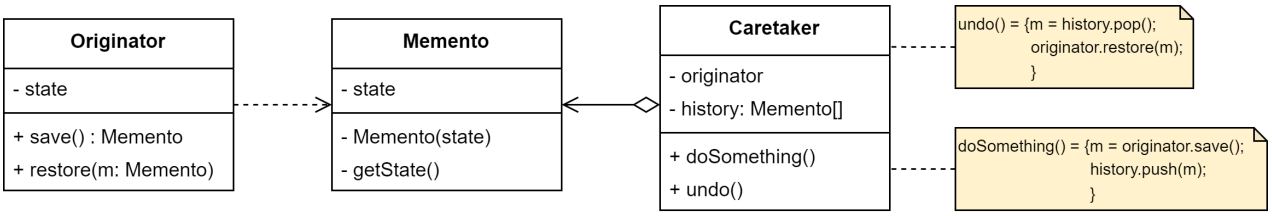
Solo l'originator conoscerà quindi la reale interfaccia del memento, e solo esso sarà in grado di istanziarlo.

19.2 Casi d'uso

- Si vuole salvare e ripristinare uno snapshot dello stato interno di un oggetto;

19.3 Diagramma UML

La classica implementazione del memento si basa sulle classi nidificate (nested-class):



In questa implementazione, la classe `Memento` è nidificata all'interno dell'originator. Ciò consente alla classe `Originator` di accedere ai campi e ai metodi del memento, anche se privati. Il `Caretaker` ha un accesso limitato ai campi e metodi del memento; questo gli consente di archiviare i singoli memento (stati/snapshots/ricordi) senza poterli alterare in alcun modo.

Implementazioni alternative si basano su un'interfaccia intermedia: in assenza di classi nidificate, si può limitare l'accesso ai campi del memento stabilendo una convenzione secondo cui i caretakers possono lavorare con un ricordo solo attraverso un'interfaccia intermedia dichiarata esplicitamente, che dichiarerebbe solo metodi relativi ai campi del memento.

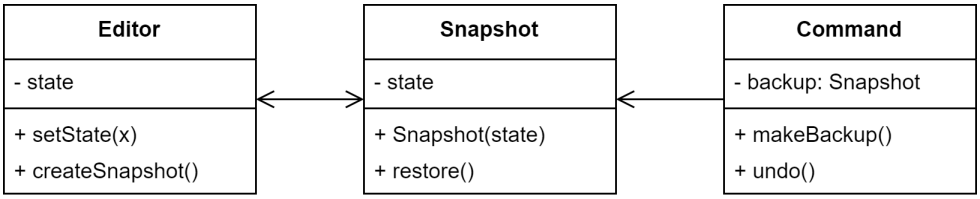
D'altra parte, gli originator possono lavorare direttamente con un oggetto memento (concreto, che implementa l'interfaccia), accedendo ai campi e ai metodi dichiarati nell'interfaccia `Memento`. Lo svantaggio di questo approccio è che è necessario dichiarare pubblici tutti i membri del memento.

19.4 Partecipanti

- **Originator:** produce uno snapshot del suo stesso stato e ripristina il suo stato a partire da uno snapshot; il client fa uso dell'originator;
- **Memento:** è di fatto lo snapshot prodotto dall'originator. Tipicamente è un oggetto immutabile dopo la creazione;
- **Caretaker:** conosce quando e perché catturare lo stato dell'originator e quando questo deve essere ripristinato. Il caretaker può tenere traccia della storia dell'originator tramite (ad esempio) uno stack.

19.5 Esempio

Il seguente esempio di utilizzo del pattern è accoppiato al command pattern. Si ipotizzi di avere un text-editor e di volerne salvare e poter successivamente ripristinare lo stato.



Gli oggetti `Command` fungono da caretakers. Essi prelevano il memento dall'editor prima di eseguire operazioni relative ai comandi. Quando il cliente chiede di annullare il comando più recente, l'editor può utilizzare il memento memorizzato in quel comando per tornare allo stato precedente.

La classe `Memento` non dichiara alcun campo pubblico, ne tantomento getter o setter. Di conseguenza nessun oggetto può alterarne il contenuto. I memento sono collegati all'oggetto editor che li ha creati, consentendo a un memento di ripristinare lo stato dell'editor collegato passando i dati tramite un setter sull'oggetto `Editor`.

Segue lo pseudocodice dell'esempio:

```
class Editor is
  private field text, curX, curY, selectionWidth

  method setText(text) is
      this.text = text

  method setCursor(x, y) is
      this.curX = x
      this.curY = y

  method setSelectionWidth(width) is
      this.selectionWidth = width

  method createSnapshot():Snapshot is
      return new Snapshot(this, text, curX, curY, selectionWidth)
```

```
class Snapshot is
  private field editor: Editor
  private field text, curX, curY, selectionWidth

  constructor Snapshot(editor, text, curX, curY, selectionWidth) is
      this.editor = editor
      this.text = text
      this.curX = x
      this.curY = y
      this.selectionWidth = selectionWidth

  method restore() is
      editor.setText(text)
      editor.setCursor(curX, curY)
      editor.setSelectionWidth(selectionWidth)


class Command is
  private field backup: Snapshot

  method makeBackup() is
      backup = editor.createSnapshot()

  method undo() is
      if (backup != null)
          backup.restore()
  // ...
```

20 Iterator

20.1 Scopo

Fornire un metodo di accesso sequenziale agli elementi di un oggetto composto senza esporre la struttura di quest'ultimo, come ad esempio scorrere una lista o visitare un albero.

Senza iterator si necessita di mantenere la responsabilità dell'accesso nella classe dell'oggetto composto, con la conseguenza che si potrebbe aver bisogno di molteplici attraversamenti. Questo causerebbe un sovraccarico dell'interfaccia dell'oggetto composto.

Con l'iterator la responsabilità di accesso e attraversamento viene separata in una classe separata, detta iteratore, che ha la responsabilità di:

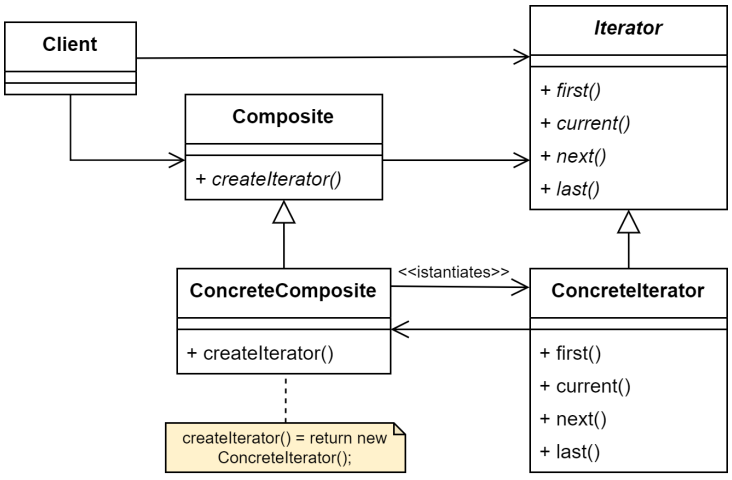
- Rappresentare un metodo/algoritmo di visita;
- Restituire l'elemento "successivo" (secondo la visita);
- Mantenere traccia dell'elemento corrente e degli elementi visitati.

A questo punto alla classe dell'oggetto composto rimane solo la responsabilità di creare l'iteratore opportuno.

20.2 Casi d'uso

- Si vuole evitare di sovraccaricare l'interfaccia della classe di un oggetto "visitabile/attraversabile";
- Si vuole evitare di centralizzare le operazioni di visita/accesso nella classe principale (il chè non consente di effettuare contemporaneamente più visite/accessi indipendenti);
- Si vuole più controllo e indipendenza tra vari metodi di visita alternativi di un oggetto molto complesso; senza sovraccaricare la classe principale.

20.3 Diagramma UML



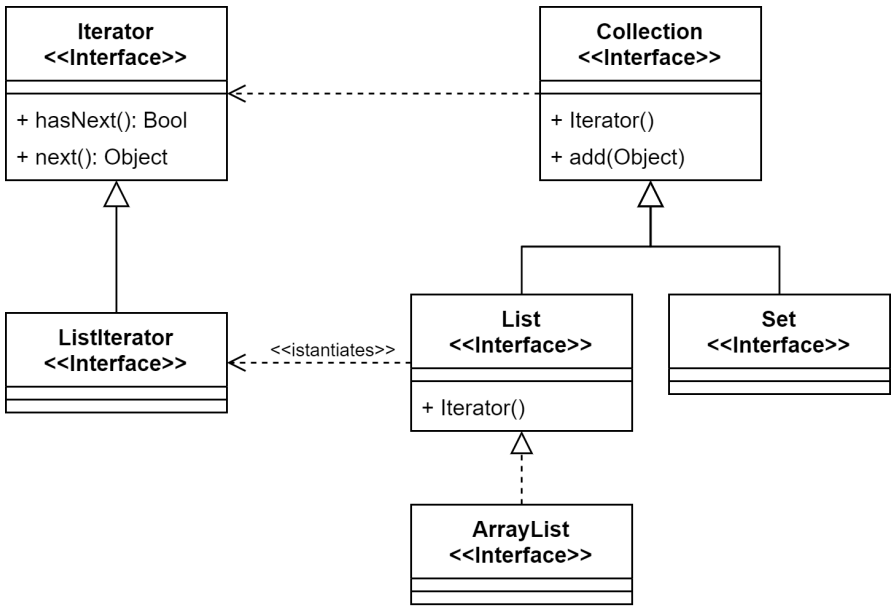
20.4 Partecipanti

- **Iterator:** definisce un'interfaccia per attraversare l'insieme degli elementi di un composite/oggetto e accedere ai singoli elementi;
- **ConcreteIterator:** implementa l'interfaccia `Iterator` tenendo traccia della posizione corrente nel composite e calcolando quale sia l'elemento successivo nell'attraversamento corrente;
- **Composite:** definisce un'interfaccia per creare un oggetto `Iterator`;
- **ConcreteComposite:** implementa l'interfaccia di creazione dell'`Iterator` e ritorna un'istanza appropriata di `ConcreteIterator`.

20.5 Esempio

Un esempio di uso del pattern può essere trovato in linguaggi di programmazione come Java; come ad esempio le interfacce `Collection`, `List`, `Set`. Il linguaggio espone l'interfaccia `java.util.Iterator`. Ad esempio, `iterator()` di `Collection` è un factory method che restituisce l'iteratore opportuno.

Il diagramma UML di `java.util.Iterator`:




```
List list = Arrays.asList(new String[] {"have","a","nice","day",":)"});
Iterator i = list.iterator();

while (i.hasNext()){
  System.out.println(i.next());
}
```

21 Visitor

21.1 Scopo

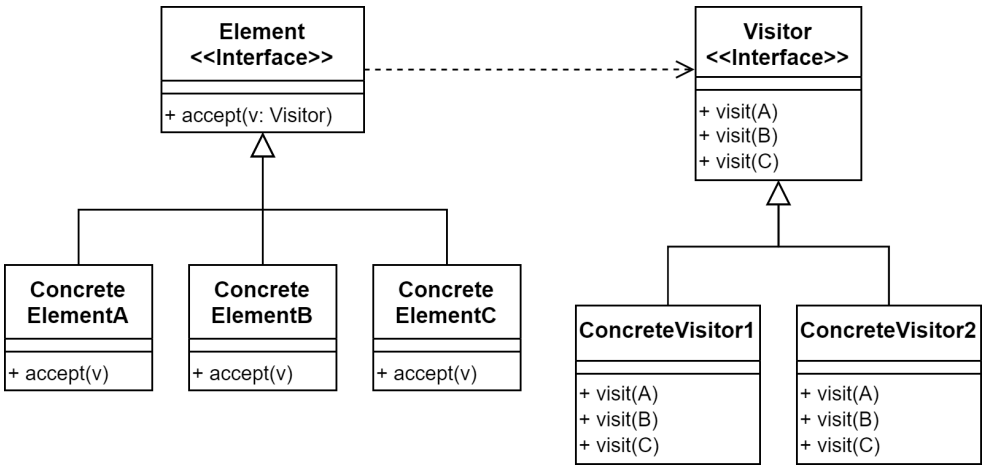
Permette di separare un algoritmo dalla struttura di oggetti composti a cui è applicato, in modo da poter rendere possibile l'aggiunta di nuove operazioni e comportamenti senza dover modificare la struttura stessa dell'oggetto su cui opera.

Ciò significa che il visitor permette di aggiungere nuove funzioni "virtuali" a una famiglia di classi, senza ncessità di modificare le classi stesse.

21.2 Casi d'uso

- Una gerarchia di oggetti è costituita da molte classi con interfacce diverse ed è necessario che l'algoritmo esegua su ogni oggetto un'operazione differente a seconda della classe concreta dell'oggetto stesso;
- È necessario eseguire molteplici operazioni indipendenti sugli oggetti di una gerarchia composta, ma non si vuole sovraccaricare le interfacce delle loro classi. Riunendo le operazioni correlate in ogni visitor è possibile inserirle nei programmi solo dove necessario;

21.3 Diagramma UML



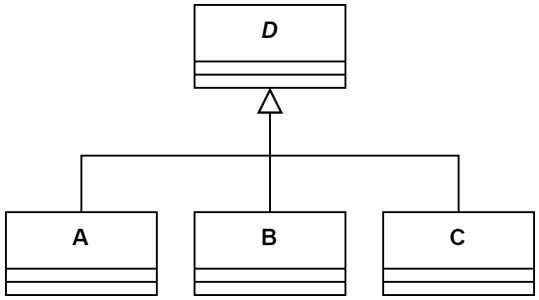
21.4 Partecipanti

- **Visitor:** definisce un'interfaccia e dichiara un metodo `visit()` per ciascun elemento concreto appartenente alla gerarchia di oggetti in modo tale che ciascuno di essi possa invocare il metodo appropriato passando un riferimento a sé (`this`) come parametro;
- **ConcreteVisitor:** implementa l'operazione `visit()` perché agisca come desiderato per la rispettiva classe;

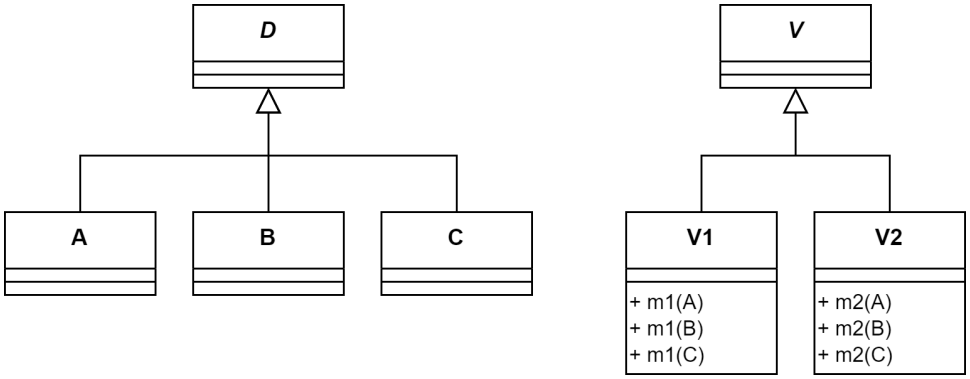
- **Element:** fornisce un'interfaccia che dichiara l'operazione `accept()` utilizzata per accettare un `Visitor` passato come parametro;
- **ConcreteElement:** implementa `accept()` per la rispettiva classe concreta.

21.5 Esempio

Si pensi di avere la seguente gerarchia. È comodo aggiungere nuove classi, basta estendere la classe `D`. Aggiungere operazioni a tutte le classi è più complesso e richiede la modifica di tutte le sottoclassi. Il visitor propone un'approccio alternativo.

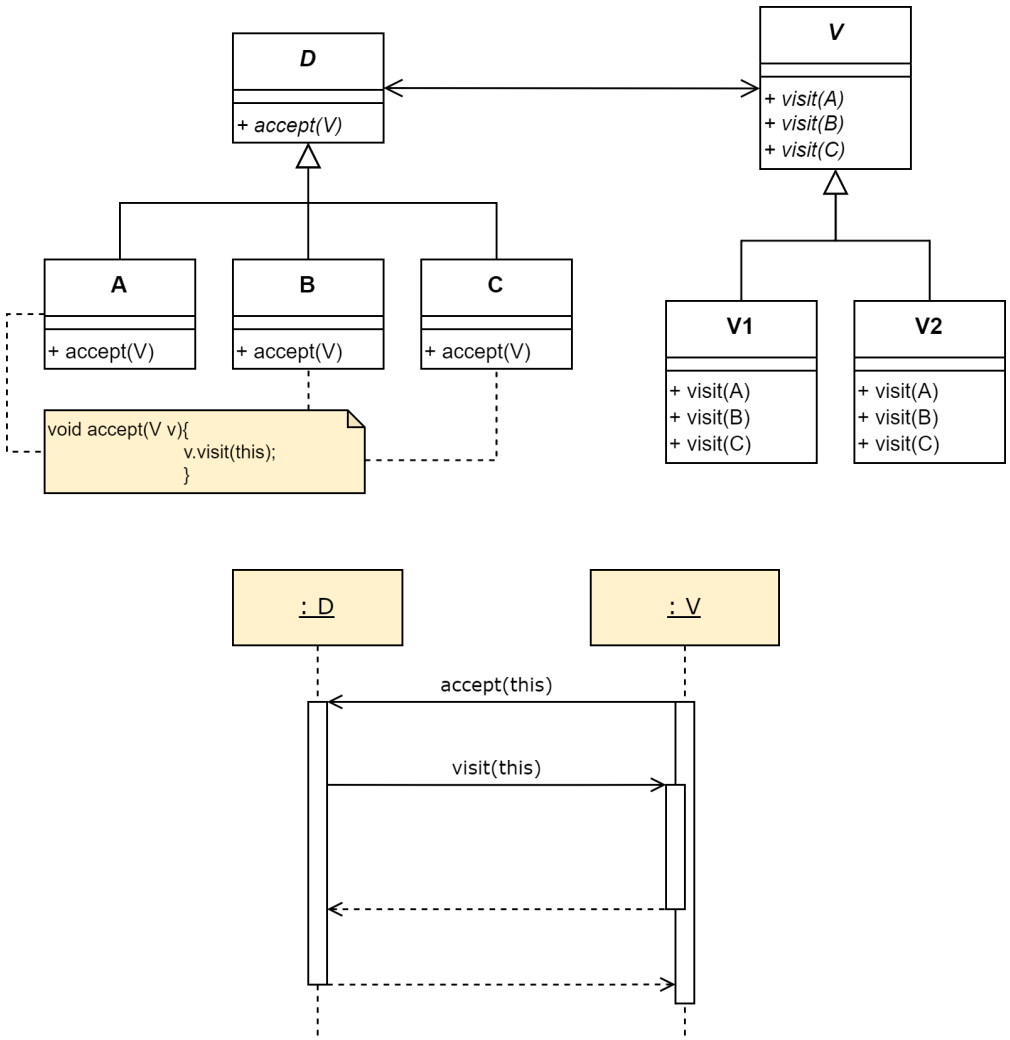


Si crea una sottoclasse di `V` per ciascuna operazione.
Ogni sottoclasse contiene l'implementazione dell'operazione per tutte le sottoclassi della gerarchia indotta da `D`. Tali funzioni vengono sovraccaricate grazie al diverso tipo di oggetto passato come argomento (un metodo per ciascuna sottoclasse di `D`).



Il visitor si basa sulla metafora del visitatore: il visitatore chiede di essere ospitato (invocando `accept()`), mentre il "padrone di casa" accetta (invocando il metodo `visit()`).

Applicando il pattern avremmo il seguente diagramma UML:



Il pattern fa uso di una tecnica chiamata "double-dispatching", in italiano "doppio inoltro", ovvero:

- `V` invoca `accept()` di `D` (in realtà il metodo sovraccaricato della sottoclasse);
- `D` esegue `accept()`, cioè invoca `visit()` di `V` (in realtà il metodo sovraccaricato della sottoclasse).

22 Chain of Responsability

22.1 Scopo

Lo scopo è di evitare l'accoppiamento fra mittente di una richiesta e il destinatario. Il pattern presenta più oggetti concatenati (handler) che possono soddisfare la richiesta.

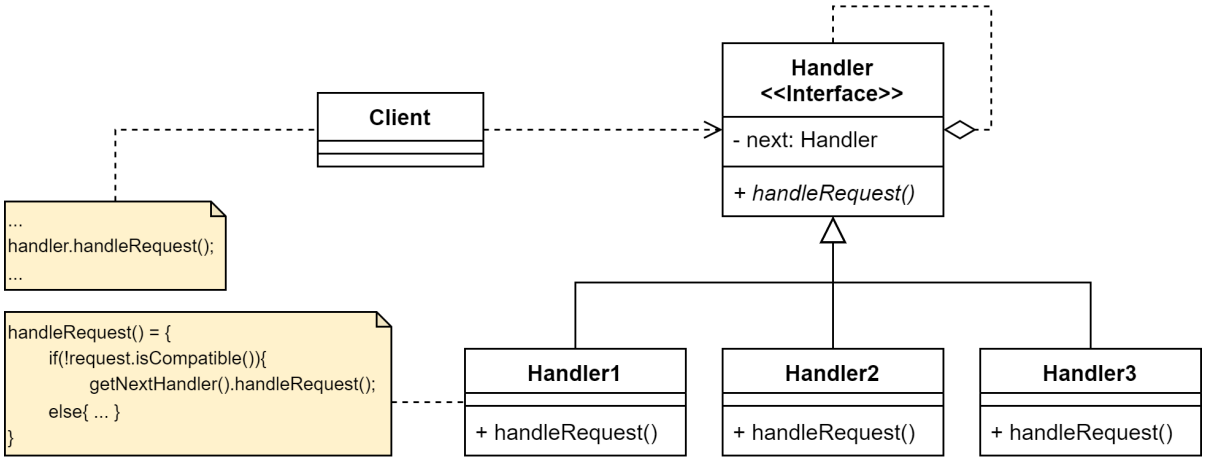
Ciascun oggetto contiene la logica che definisce le tipologie di comandi che può gestire; i comandi non gestiti sono passati all'oggetto successivo facente parte della catena.

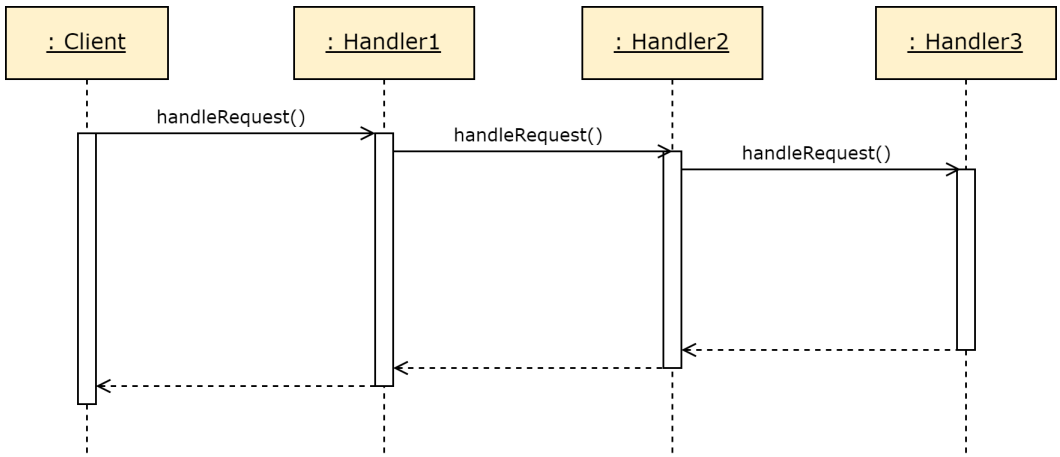
Se la richiesta arriva a fine catena, non è gestita. La catena è modificabile a run-time.

22.2 Casi d'uso

- Si vuole evitare l'accoppiamento tra sender e receiver di una richiesta;
- Si vuole permettere l'esistenza di più di un receiver capace di gestire una richiesta.

22.3 Diagramma UML





22.4 Partecipanti

- **Handler:** rappresenta la classe astratta/interfaccia che offre il metodo `handleRequest()` che verrà utilizzato dal codice del client (mittente) per inoltrare le richieste;
- **ConcreteHandler:** rappresenta l'effettiva implementazione della gestione dei comandi/richieste per un oggetto.

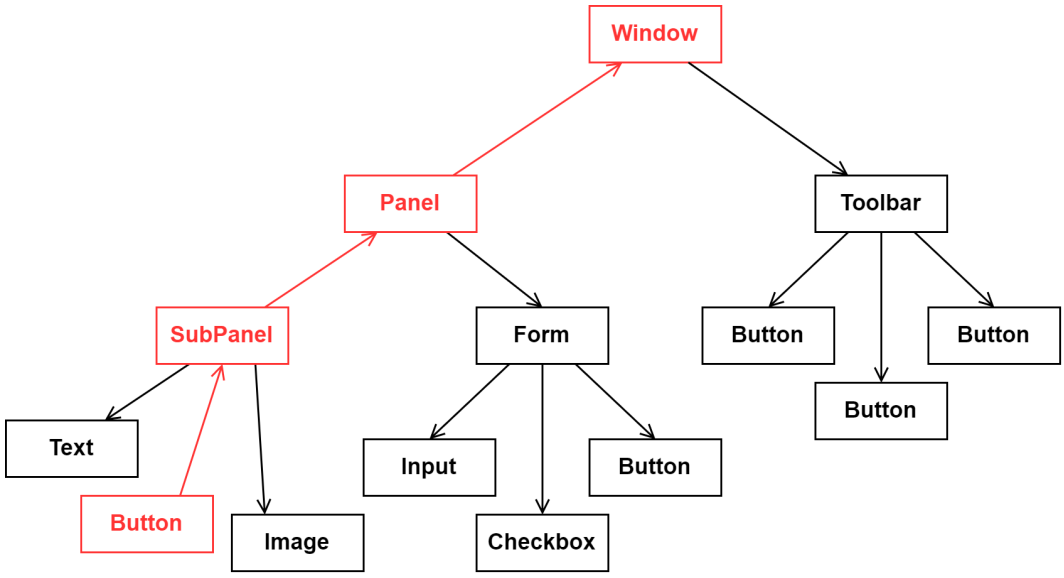
22.5 Esempio

L'uso del pattern è molto comune nella gestione di eventi in uno stack di componenti grafiche per interfacce utenti.

Per esempio, quando un utente clicca un pulsante, l'evento viene propagato attraverso la catena di responsabilità di elementi della gerarchia della GUI, iniziando in questo caso dalla classe `Button`.

Tale richiesta (evento) viene propagata attraverso i suoi container (ad esempio dei form o dei panel), fino ad arrivare "in cima" alla finestra principale dell'applicazione.

L'evento è processato dal primo elemento della catena capace di gestirlo.



È cruciale che tutte le classi `Handler` implementino la stessa interfaccia.

23 Interpreter

23.1 Scopo

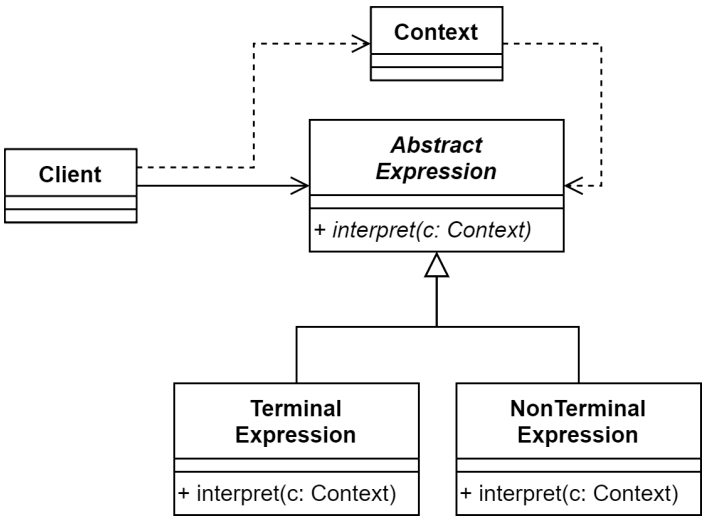
Definire una rappresentazione della grammatica di un linguaggio insieme ad un componente interprete che utilizza tale rappresentazione per l'interpretazione delle espressioni la cui sintassi rispetta il linguaggio.

L'interprete specifica come valutare sentenze in un linguaggio. L'idea base è quella di avere una classe per ciascun simbolo (terminale o non terminale). L'albero di sintassi di una sentenza nel linguaggio trattato è un'istanza di un composite pattern ed è usata per valutare (interpretare) la sentenza per il client.

23.2 Casi d'uso

- Implementazione di linguaggi d'interrogazione di database;
- Implementazione di linguaggi informatici specializzati; ad esempio per descrivere protocolli di comunicazione;
- Implementazione di linguaggi di programmazione.

23.3 Diagramma UML



23.4 Partecipanti

- **AbstractExpression:** definisce un'interfaccia per interpretare un'espressione tramite il metodo `interpret(context)`;
- **TerminalExpression:** non ha figli ed interpreta una espressione direttamente;
- **NonTerminalExpression:** contiene una lista di espressioni figlie a cui inoltra le sotto-richieste di interpretazione.

23.5 Esempio

Si pensi alla seguente grammatica per espressioni aritmetiche in Backus-Normal form (BNF):

```
expression ::= plus | minus | variable | number
plus ::= expression expression '+'
minus ::= expression expression '-'
variable ::= 'a' | 'b' | 'c' | ... | 'z'
digit = '0' | '1' | ... | '9'
number ::= digit | digit number
```

Produce espressioni in notazione polacca inversa, come:

```
a b +
a b c + -
a b + c a - -
```

Tale grammatica può essere implementata in C# con l'uso dell'interpreter-pattern nel seguente modo:

```
class Program
{
    static void Main()
    {
        var context = new Context();
        var input = new MyExpression();

        var expression = new OrExpression
        {
            Left = new EqualsExpression
            {
                Left = input,
                Right = new MyExpression { Value = "4" }
            },
            Right = new EqualsExpression
            {
                Left = input,
                Right = new MyExpression { Value = "four" }
            }
        };

        input.Value = "four";
        expression.Interpret(context);
        // Output: "true"
        Console.WriteLine(context.Result.Pop());

        input.Value = "44";
        expression.Interpret(context);
        // Output: "false"
        Console.WriteLine(context.Result.Pop());
    }
}
```

```
class Context
{
    public Stack<string> Result = new Stack<string>();
}

interface Expression
{
    void Interpret(Context context);
}

abstract class OperatorExpression : Expression
{
    public Expression Left { private get; set; }
    public Expression Right { private get; set; }

    public void Interpret(Context context){
        Left.Interpret(context);
        string leftValue = context.Result.Pop();

        Right.Interpret(context);
        string rightValue = context.Result.Pop();

        DoInterpret(context, leftValue, rightValue);
    }

    protected abstract void DoInterpret(Context context, string leftValue,
    string rightValue);
}

class EqualsExpression : OperatorExpression
{
    protected override void DoInterpret(Context context, string leftValue,
    string rightValue){
        context.Result.Push(leftValue == rightValue ? "true" : "false");
    }
}

class OrExpression : OperatorExpression
{
    protected override void DoInterpret(Context context, string leftValue,
    string rightValue){
        context.Result.Push(leftValue == "true" || rightValue == "true" ?
   "true" : "false");
    }
}

class MyExpression : Expression
{
    public string Value { private get; set; }

    public void Interpret(Context context){
        context.Result.Push(Value);
    }
}
```

Per completezza si fornisce anche il codice del parser:

```
  private static Expr parseToken(String token, ArrayDeque<Expr> stack) {
    Expr left, right;
    switch(token) {
    case "+":
        // It's necessary to remove first the right operand from the stack
        right = stack.pop();
        // ...and then the left one
        left = stack.pop();
        return Expr.plus(left, right);
    case "-":
        right = stack.pop();
        left = stack.pop();
        return Expr.minus(left, right);
    default:
        return Expr.variable(token);
    }
}
public static Expr parse(String expression) {
    ArrayDeque<Expr> stack = new ArrayDeque<Expr>();
    for (String token : expression.split(" ")) {
        stack.push(parseToken(token, stack));
    }
    return stack.pop();
}
```

Un esempio di utilizzo:

```
  public static void main(final String[] args) {
    Expr expr = parse("w x z - +");
    Map<String, Integer> context = Map.of("w", 5, "x", 10, "z", 42);
    int result = expr.interpret(context);
    System.out.println(result);         // -27
}
```