DESIGN PATTERN STRUTTURALI

I design pattern strutturali affrontano problemi che riguardano la composizione di classi ed oggetti; sfruttano ereditarietà ed aggregazione per consentire il riutilizzo di oggetti esistenti. Ce ne sono 7 in tutto ma qui non vengono analizzati i seguenti: Bridge, Composite e Flyweight.

- ## ADAPTER

Descrizione.
L'adapter pattern serve a convertire l'interfaccia di una classe in un'altra. Torna molto utile quando ho bisogno di poter utilizzare assieme due oggetti che sono tipi diversi e quindi non sono compatibili fra di loro. In questo caso, prendo uno dei due oggetti e lo metto dentro ad un "adattatore" che me lo rende compatibile con l'altro oggetto. Per fare ciò ho due modi:

1. Object adapter. Uso una classe adattatore che implementa l'interfaccia adapter; sfrutto quindi il principio di composizione.

2. Class adapter. Uno una classe adattatore che estende la classe "non compatibile" e la converte in una di "compatibile" secondo le mie esigenze; sfrutto quindi il principio di ereditarietà.

Generalmente è preferibile utilizzare il primo approccio perché rende il *coupling* delle classi molto più fino rispetto al secondo approccio. Comunque dipende sempre dai casi, non esiste il metodo migliore in assoluto.

Esempio.
In questo esempio si vuole creare un programma che disegna figure geometriche e fa delle azioni su di esse. La libreria è stata progettata da me e scelgo questa struttura:

```
public interface Shape {
  void draw();
  void resize();
  String description();
}

class Rectangle implements Shape {

  @Override
  public void draw() {
    System.out.println("Rect.");
  }

  @Override
  public void resize() {
    System.out.println("Resize");
  }

  @Override
  public String description() {
    return "Rectangle object";
  }

}
```

```
class Circle implements Shape {

  @Override
  public void draw() {
    System.out.println("Circle");
  }

  @Override
  public void resize() {
    System.out.println("Resize");
  }

  @Override
  public String description() {
    return "Circle object";
  }

}
```

La gerarchia è molto semplice; la struttura delle classi ha 3 metodi perché presenti nell'interfaccia `Shape`. Adesso prendiamo da esempio un'altra gerarchia di classi, sempre di figure geometriche, però fatta da un altro programmatore.

```
public interface Figure {
  double area();
  double perimeter();
  String drawShape();
}

class Triangle implements Figure {

  @Override
  public double area() {
    return 1;
  }

  @Override
  public double perimeter() {
    return 2;
  }

  @Override
  public String drawShape() {
    return "Triangle object";
  }

}
```

```
class Rhombus implements Figure {

  @Override
  public double area() {
    return 5;
  }

  @Override
  public double perimeter() {
    return 6;
  }

  @Override
  public String drawShape() {
    return "Rhombus object";
  }

}
```

Bene, adesso mettiamo che nel main voglio usare la mia libreria e quindi posso scrivere questo codice molto semplice per salvare una serie di figure.

```
List<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Rectangle());
shapes.add(new Circle());
//shapes.add(new Triangle()); non compila!
```

Non posso però usare la classe `Triangle` perché non implementa `Shape` e quindi sono fregato. Per risolvere il problema uso l'adapter pattern e ho due modi per farlo:

1. Object Adapter pattern

```
class FigureObjectAdapter implements Shape {

  private Figure figure;

  public FigureObjectAdapter(Figure figure) {
    this.figure = figure;
  }

  @Override
  public void draw() {
    figure.drawShape();
  }

  @Override
  public void resize() {
    System.out.println(description() + " cannot be resized.");
  }

  @Override
  public String description() {
```

```
    if (figure instanceof Triangle) {
      return "Triangle object";
    } else if (adaptee instanceof Rhombus) {
      Return "Rhombus"
    } else {
      Return "Unknown object";
    }
  }

}
```

Adesso posso usare tutte le figure nel mio array perché ho una classe adapter che "adatta" la struttura di una classe non mia alla struttura che mi serve e dunque fa da tramite.

```
List<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Rectangle());
shapes.add(new Circle());
shapes.add(new FigureObjectAdapter(new Triangle()));
```

2. Class Adapter pattern

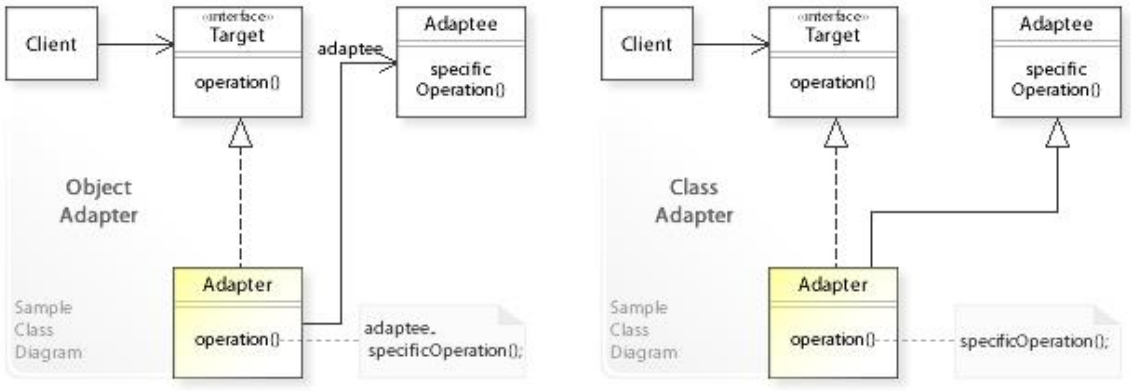
```
class TriangleAdapter extends Triangle implements Shape {

  public TriangleAdapter() {
    super();
  }

  @Override
  public void draw() {
    this.drawShape();
  }

  @Override
  public void resize() {
    System.out.println(description() + " cannot be resized.");
  }

  @Override
  public String description() {
    return "Triangle object";
  }

}
```

Adesso posso usare tutte le figure nel mio array perché ho una classe adapter che "adatta" la struttura di una classe non mia alla struttura che mi serve e dunque fa da tramite.

```
List<Shape> shapes = new ArrayList<Shape>();
shapes.add(new Rectangle());
shapes.add(new Circle());
shapes.add(new TriangleAdapter());
```

In entrambi i casi quindi ottengo lo stesso identico risultato però cambia l'approccio; il secondo non posso usarlo se la classe padre è finale e crea una forte dipendenza tra le classi. Il primo approccio invece non posso usarlo se ho bisogno di aggiungere comportamenti alla classe che voglio adattare.

Diagramma.



- DECORATOR

Descrizione.
Il decorator pattern serve ad aggiungere funzionalità ad un oggetto dinamicamente. Nel particolare, si tratta di inglobare un oggetto dentro ad un altro oggetto che offre le stesse caratteristiche più alcune extra. Note:

1. Aggiunge funzionalità in modo trasparente
2. Non fa uso di subclassing (che non sempre è possibile)
3. Evita che le classi siano agglomerati di funzioni

Il problema di questo pattern è che potrebbe avere una struttura abbastanza complessa da mettere su (ovviamente dipende da caso a caso cosa devo fare). Questo pattern si pone come "alternativa" all'ereditarietà.

Esempio.
Supponiamo di voler creare un programma che gestisce gli ordini di pizze in una pizzeria. Serve un programma che possa salvare in memoria gli ingredienti che uno vuole in una pizza (pomodoro, mozzarella, funghi, ZEOLE, bisi...). Vediamo come fare:

1. Si definisce l'interfaccia (usando le interfacce) dell'oggetto a cui verranno aggiunte funzionalità. Voglio aggiungere ingredienti ad una pizza.

```
public interface Pizza {
  //Ritorna la lista di ingredienti della pizza
  List<String> ingredients();
}
```

2. Si crea una classe concreta alla quale aggiungere le funzionalità extra. In questo caso creo la "Pizza vuota" cioè, solo la pasta che è l'ingrediente base necessario per fare una pizza. Questo non è un decoratore, è solo la *fine* della catena di innesti.

```
public class PizzaBase implements Pizza {
  public List<String> ingredients() {
    return Collections.singletonList("pasta);
  }
}
```

3. Poi bisogna definire una classe che mantiene un riferimento all'interfaccia che ci interessa e che fa da "innesto" per le altre classi decorator che poi si attaccheranno ad essa. Questa serve per poter aggiungere dinamicamente pezzi; in altre parole mi serve a fare le decorazioni, cioè a inglobare gli oggetti.

```
public abstract class PizzaIngredients implements Pizza {

  private final Pizza toDecorate;

  protected PizzaIngredients(Pizza toDecorate) {
    this.toDecorate = toDecorate;
  }
  public List<String> ingredients() {
    return addIngredients(toDecorate.ingredients());
  }

  protected abstract List<String> addIngredients(List<String> ig);
}
```

4. Ora si possono creare tutti i decorator che aggiungono funzionalità all'oggetto di base andando a "racchiuderlo". Posso quindi creare classi che aggiungono ingredienti alla pizza.

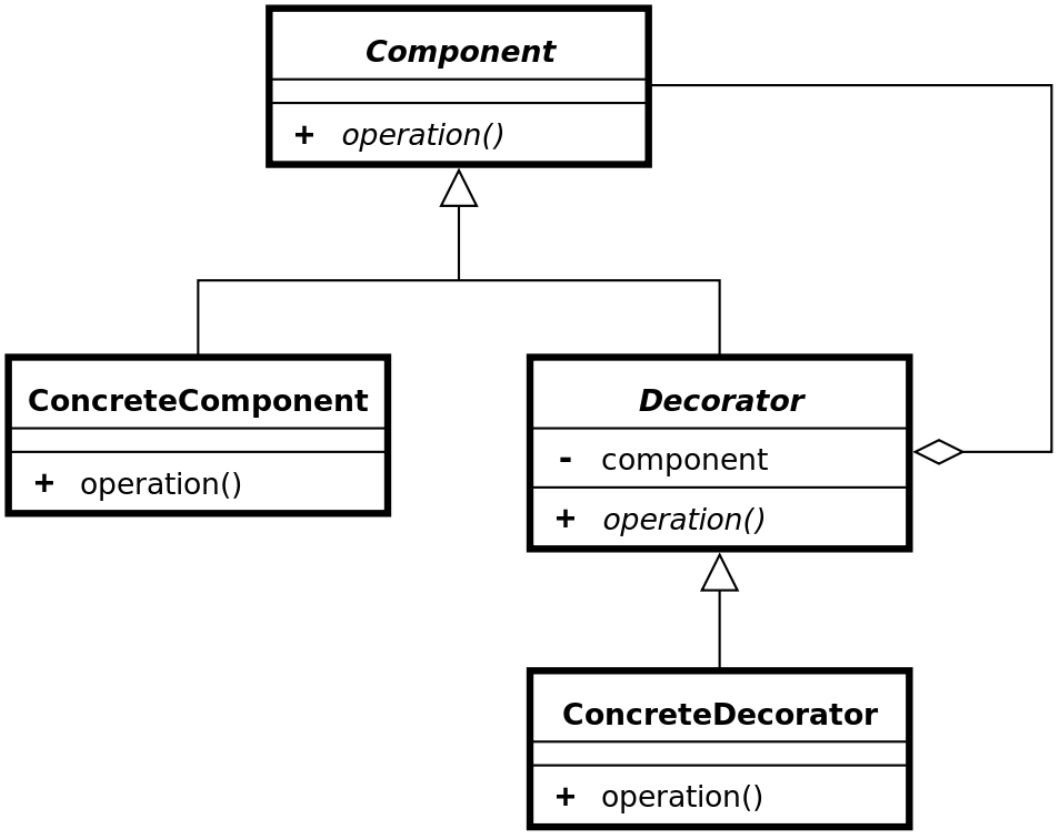
```
public class Tomato extends PizzaIngredients {

  protected Tomato (Pizza toDecorate) {
    super(toDecorate);
  }

  protected List<String> addIngredients(List<String> ig) {
    final List<String> igCopy = new ArrayList<String>(ig);
    igCopy.add("tomato");
    return igCopy;
  }
}

public class Mozzarella extends PizzaIngredients {

  protected Mozzarella (Pizza toDecorate) {
    super(toDecorate);
  }

  protected List<String> addIngredients(List<String> ig) {
    final List<String> igCopy = new ArrayList<String>(ig);
    igCopy.add("mozzarella");
    return igCopy;
  }
}

public class Zeola extends PizzaIngredients {

  protected Zeola (Pizza toDecorate) {
    super(toDecorate);
  }

  protected List<String> addIngredients(List<String> ig) {
    final List<String> igCopy = new ArrayList<String>(ig);
    igCopy.add("zeola");
    return igCopy;
  }
}
```

Tutte le classi che estendono `PizzaIngredients` sono le classi che "inglobano" gli oggetti; sono quindi i *decorators*. Ogni volta che faccio la new passo nel costruttore un oggetto di tipo `Pizza` e poi la `addIngredients` viene sempre chiamata (vedi classe base) per aggiungere nuovi ingredienti.

Bene, finito. Adesso posso creare un main, chiamandolo magari Pizzaiolo, e dentro posso creare questo codice per fare vari tipi di pizza. Ovviamente non conta l'ordine con cui faccio le `new`.

```
Pizza margherita = new Mozzarella(new Tomato(new PizzaBase()));
Pizza cipolla = new Zeola(new Tomato(new Mozzarella(new PizzaBase())));
```

Diagramma.



Il **Component** definisce l'interfaccia a cui veranno aggiunge funzionalità (`Pizza`) e **ConcreteComponent** è il pezzo "base" dal quale partire e attorno ad esso verranno aggiunte altre funzionalità (`PizzaBase`). Il **Decorator** è la classe base che fa da "innesto" per far partire la catena (`PizzaIngredients`) mentre **ConcreteDecorator** sono tutte le classi che possono avvolgere altri oggetti (`Tomato, Mozzarella, Zeola`).

- ## FACADE

Descrizione.
Il facade pattern serve a fornire un'interfaccia unica e semplice per un sottoinsieme molto complesso di oggetti. Se ci sono un sacco di classi con un sacco di metodi per

gestire qualcosa, il facade fa si che ci sia una **unica** classe che si occupa di gestire tutto questo casino di classi. Fornisce poi un'interfaccia di metodi semplici per comunicare.

Questo mi permette di interagire con una sola classe invece che con cento mila e mi fa chiamare pochi metodi invece di dovermi preoccupare di fare troppe chiamate. Il problema di questo pattern è che gestire il tutto potrebbe diventare molto difficile anche se il facade diventa un *Single Failure Point* (se stacco il facade da tutto e il programma va, allora so già al 100% che il problema sta qui).

Esempio.
Supponiamo di dover gestire un grosso impianto nucleare e che ogni compartimento sia chiamato SubsystemX. Ci sono tante classi con tanti metodi e gestire tutto quanto a mano causa la creazione di un sacco di new, codice sparso.

```
class SubsystemA {
  public int OperationA1() { ... }
  public int OperationA2() { ... }
  public int OperationA3() { ... }
  public int OperationA4() { ... }
  public int OperationA5() { ... }
}
```

```
class SubsystemB {
  public int OperationB1() { ... }
  public int OperationB2() { ... }
  public int OperationB3() { ... }
  public int OperationB4() { ... }
  public int OperationB5() { ... }
}
```

```
class SubsystemC {
  public int OperationC1() { ... }
  public int OperationC2() { ... }
  public int OperationC3() { ... }
  public int OperationC4() { ... }
  public int OperationC5() { ... }
}
```

```
class SubsystemD {
  public int OperationD1() { ... }
  public int OperationD2() { ... }
  public int OperationD3() { ... }
  public int OperationD4() { ... }
  public int OperationD5() { ... }
}
```

Il facade pattern suggerisce di creare un singolo punto d'accesso che permetta di gestire tutte le varie istanze; come se fosse un pannello di controllo. Gestisce lui le new e chiama i metodi quando necessario.

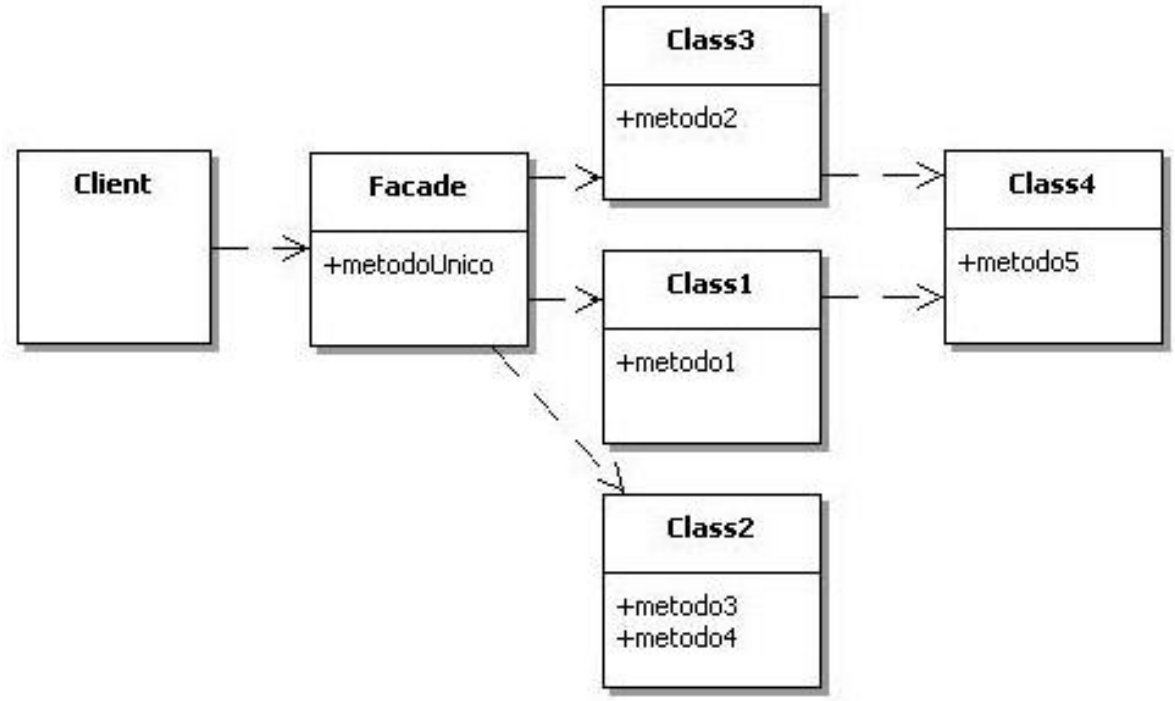
```
class SubsystemFacade {

  private SubsystemA a;

  private SubsystemB b;

  private SubsystemC c;

  private SubsystemD d;

  public SubsystemFacade() {
    a = new SubsystemA();
    b = new SubsystemB();
    c = new SubsystemC();
    d = new SubsystemD();
  }

  public int operation1() {
    return a.OperationA1() +
           b.OperationB1() +
           c.OperationC1() +
           d.OperationD1();
  }
```

```
    public int operazion2() {
      return a.OperationA2() + ...
    }

    //e via con tutti gli altri metodi

}
```

È come se i subsystem fossero classi che controllano vari hardware del computer (HDD, GPU, RAM) e il facade fosse una classe Computer che con semplici comandi mi fa partire tutti i componenti e mi dice il loro stato.

Grazie alla classe `SubsystemFacade` il client si deve interfacciare solamente con questa classe e si crea un accoppiamento molto **lasco** tra di lui e i sottoinsiemi con cui deve comunicare. Aiuta anche i tempi di compilazione e di building.

Diagramma.



La classe `Facade` di questo esempio contiene un solo metodo che serve per gestire tutta la struttura di oggetti che stanno sotto. Il client parla col facade, no col sottosistema perché ***Facade*** fa da punto di innesto.

- PROXY

Descrizione.
Il proxy pattern serve per fornire un surrogato di un altro oggetto del quale si vuole controllare l'accesso. Quando l'utente apre una galleria per esempio, il computer carica tante piccole anteprime (le *thumbnail*) e solo dopo averci cliccato sopra si vede l'immagine con tutte le informazioni.

Un proxy rinvia il costo di creazione di un oggetto all'effettivo utilizzo facendo dunque *lazy evaluation*. In altre parole questo pattern serve per mostrare una versione "lite" di

un oggetto, una anteprima poco pesante, che poi però è in grado di generare l'oggetto vero e proprio.

Esempio.
Supponiamo di voler creare un programma che mostra video a schermo tipo VLC; sarà necessaria una classe `Document` che rappresenta un documento dove dentro ci stanno i video, rappresentati dalla classe `RealVideo`. Alla creazione di `Document` non avrebbe senso caricare in memoria **tutti** i video perché pesano, mi occupano memoria e non è nemmeno detto che l'utente li guardi tutti quindi potrei averli caricati per niente.

La soluzione giusta è quella di usare il proxy pattern. Alla creazione della classe `Document` verrà creata una classe `ProxyVideo` che carica in memoria il video solo nel momento in cui questo viene aperto. In tal modo, non vengono caricati in memoria tutti i video subito ma viene caricata un'anteprima (col proxy) che poi, se selezionata, caricherà il video (l'oggetto vero e proprio).

1. Si crea l'interfaccia comune sia al proxy che all'oggetto "reale"

```
public interface Video {
    void playVideo();
}
```

2. Si crea il proxy che implementa l'interfaccia sopra e contiene un'istanza all'oggetto reale Video che andremo a creare

```
public class VideoProxy implements Video {
    private final String filePath;
    private RealVideo video;

    public VideoProxy(String filePath) {
        this.filePath = filePath;
    }

    @Override
    public void playVideo() {
        byte[] bytes = getBytesFromVideoPath(filePath);
        this.video = new RealVideo(bytes);
        this.video.playVideo ();
    }
}
```

3. Si crea ora l'oggetto vero e proprio, quello che viene inizializzato in modo lazy dal rispettivo proxy

```
public class RealVideo implements Video {
    private final byte[] file;

    public RealVideo(btye[] file) {
        this.file = file;
        System.out.println("Loading video...");
    }

    public void playVideo() {
        System.out.println("Reproducing video...");
    }

}
```

Finito. È molto semplice quindi implementare il pattern perché basta avere un'interfaccia che definisca l'oggetto di cui fare l'anteprima, definire il proxy dandogli dentro un riferimento all'oggetto vero e proprio (che creerà in modo **lazy**) e poi creare l'oggetto desiderato.

4. Ai fini dell'esempio, vediamo ora di creare la classe Document che contiene una lista di tutti i video ma attenzione, nella lista vogliamo le **anteprime** dei video perché sono leggere (dato che non caricano subito il video in memoria). Bisogna dunque fare una lista di proxy.

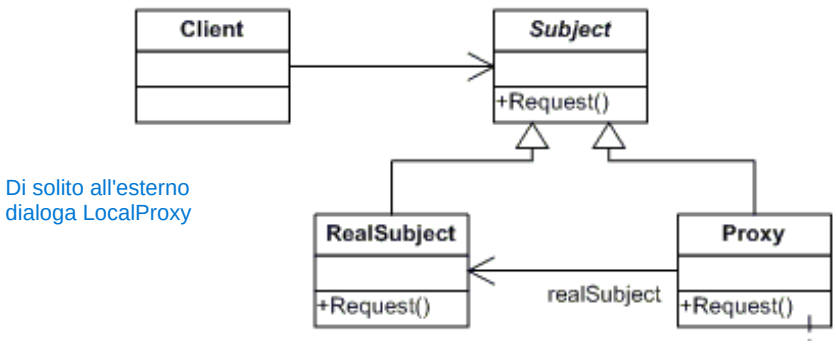
```
public class Document {

    //Video è l'interfaccia implementata dal VideoProxy, quindi
    //dentro a videos posso metterci i proxy
    private final List<Video> videos;

    public Document(List<Video> videos) {
        this.videos = videos;
    }

    public void show() {
        System.out.println("Showing videos list:");
        //Carica il video in memoria e fallo partire
        videos.stream().forEach( ... );
    }

}
```

5. Ora nel main basta creare una lista di `VideoProxy` ed assegnarla al `Document` per ottenere un uso efficiente della memoria, grazie al pattern

```
List<Video> list = new ArrayList<>();
list.add(new ProxyVideo("roberto.mp4"));
list.add(new ProxyVideo("roberta.mp4"));

//In memoria non ci sono video caricati, ci sono i proxy ma i file
//.mp4 o quello che è non sono ancora in memoria

final Document document = new Document(list);
document.show();

//Adesso dopo la show ci sono i video in memoria
```

Avendo fatto così, al momento della creazione della lista dei video ho molta efficienza perché carico in memoria i bytes solo quando strettamente richiesto.

Diagramma.



DESIGN PATTERN CREAZIONALI

I design pattern creazionali affrontano problemi che riguardano la creazione degli oggetti in modo efficace in base al problema che si pone. Si fa un ampio uso delle interfacce per rendere il sistema indipendente dall'implementazione concreta delle sue componenti. Ce ne sono 5 in tutto ma qui non vengono analizzati i seguenti: Prototype e Factory method.

- ## SINGLETON

Descrizione.
Il singleton ha lo scopo di assicurare l'esistenza di un'unica istanza di una classe e fornisce quindi un punto di accesso unico ad una risorsa. Si tratta di una classe con una sola istanza e dei metodi che permettono di accedere ad essa.

Esempio.
Supponiamo di avere un unico database e di volere avere un unico punto di accesso ad esso dall'applicazione. Il seguente codice permette di creare un singleton che fornisce l'accesso univoco al database:

```
public class SingletonDB {

    private static SingletonDB instance;

    public SingletonDB() {}

    public static SingletonDB getInstance() {
        if (instance == null) {
            instance = new SingletonDB();
        } else {
            return instance;
        }
    }

    public String query(String queryBody) { ... }

}
```

In questo modo, quando ho bisogno di interrogare il database da qualsiasi parte della mia applicazione, posso usare il seguente codice:

```
String res = SingletonDB.getInstance().query("SELECT test FROM table");
```

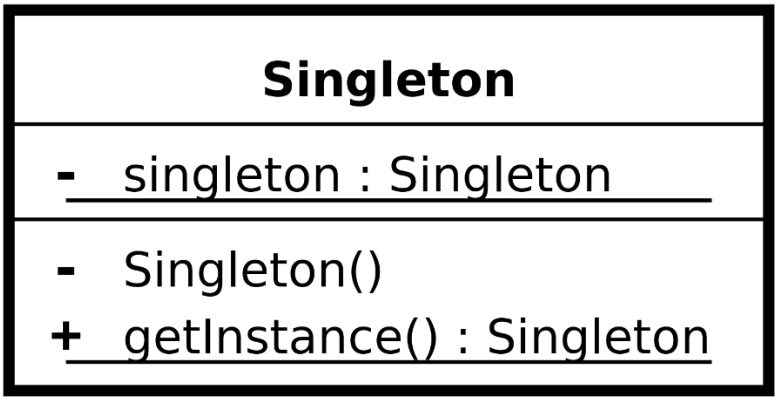
In realtà, in Java questo codice è "vecchio" e ora non va più bene. In ambienti multithread è necessaria la thread-safety che non è per nulla garantita da questo approccio. L'approccio moderno del singleton in Java è questo:

```
public enum EnumSingleton {

    INSTANCE("Info");

    private String info;

    private EnumSingleton(String info) {
        this.info = info;
    }
```

```
    public EnumSingleton getInstance() {
        return INSTANCE;
    }

}
```

Si tratta di un nuovo modo **nativo** di Java per creare singleton usando la keyword `INSTANCE`. Questo approccio è thread-safe e ha serializzazione in quanto gli `enum` sono thread-safe e serializzabili per implementazione.

Diagramma.



- BUILDER

Descrizione.
Separa la costruzione di un oggetto complesso dalla sua rappresentazione. A volte può succedere che il costruttore di un oggetto richieda tanti parametri (5 o più) e quindi ho sia un problema di leggibilità del codice sia un problema legato alle dipendenze. Con questo pattern passo al costruttore solamente pochi parametri necessari e tutti gli altri vengono aggiunti man mano (e non tutti sono obbligatori).

Esempio.
Supponiamo di voler creare una classe che rappresenti un'email. Sarà necessario ricevere in input, quindi tramite costruttore, tutte le seguenti cose:

1. L'email del destinatario
2. Il testo dell'email
3. Il saluto alla persona alla quale inviamo l'email
4. La frase di chiusura dell'email
5. Un titolo all'email

Per fare ciò vediamo un esempio di builder pattern che evita la creazione di un costruttore a 5 parametri. Vale la pena usarlo comunque perchè se in futuro fossero necessari altri parametri di input il costruttore diventerebbe un casino con 8/9 parametri. Col builder è anche più chiaro capire cosa succede.

```
public class Email {

    private final String title;
    private final String recipients;
    private final String message;
```

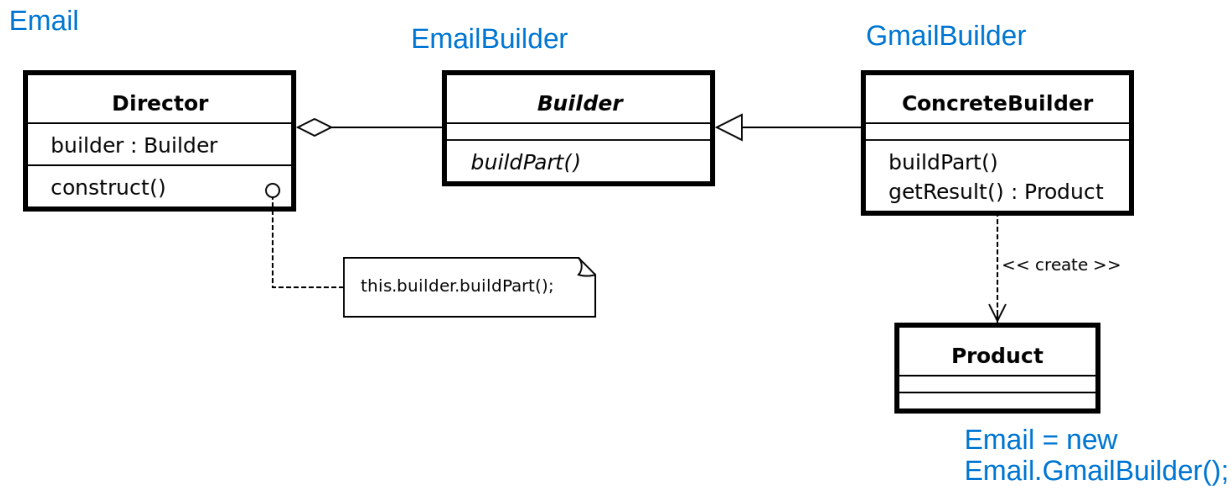
```
    private Email(String title, String recipients, String message) {
        this.title = title;
        this.recipients = recipients;
        this.message = message;
    }

     public String getMessage() {
         return message;
     }

     public String getRecipients() {
         return recipients;
     }

     public String getTitle() {
         return title;
     }

     public static class EmailBuilder {

         private String recipients;
         private String title;
         private String greeting;
         private String mainText;
         private String closing;

         public EmailBuilder setTitle(String title) {
             this.title = title;
             return this;
         }

         public EmailBuilder setRecipients(String recipients) {
             this.recipients = recipients;
             return this;
         }

         public EmailBuilder setGreeting(String greeting) {
             this.greeting = greeting;
             return this;
         }

         public EmailBuilder setMainText(String mainText) {
             this.mainText = mainText;
             return this;
         }

         public EmailBuilder setClosing(String closing) {
             this.closing = closing;
             return this;
         }

         public Email build() {
             String message = greeting + "\n" + mainText + "\n"
                              + closing;
             return new Email(title, recipients, message);
        }

    }
}
```

Si crea la classe `Email` con un costruttore **privato** perché la nuova istanza di questa classe non verrà fornita direttamente dall'operatore new ma dalla classe builder interna. Il builder, cioè la classe che ritorna l'istanza dell'oggetto da creare, contiene

tanti setter quanti sono i parametri da impostare ed infine col metodo `build()`, che va chiamato alla fine, costruisce l'oggetto.

```
Email email = new Email.EmailBuilder()
                .setRecipients("roberto@gmail.com")
                .setMainText("Hello, this is the builder pattern!")
                .setGreeting("Hi Robert!")
                .setClosing("Regards")
                .setTitle("Builder pattern test")
                .build();
```

Diagramma.



Questo diagramma prevede l'uso di una interfaccia `Builder` che non è strettamente necessaria, però comunque male non fa. Il `ConcreteBuilder` è il builder, ovvero la classe che crea l'istanza come `EmailBuilder` dell'esempio. Director è la classe che costruisce l'oggetto e può essere interna (come nel nostro esempio) oppure esterna che riceve il builder in ingresso (come mostra il diagramma).

- ## ABSTRACT FACTORY

Descrizione.
L'abstract factory fornisce un'interfaccia per creare famiglie di prodotti senza dovere specificare le classi concrete. Vengono definite le interfacce degli oggetti da creare e le classi che concretizzano il factory vengono create una sola volta. Si usa quando si vuole fornire una libreria di classi senza rivelare l'implementazione oppure quando si vuole configurare un sistema con più famiglie di prodotti. Ad esempio vendere una libreria per creare GUI con bottoni e finestre sia per Windows che per Mac OS.

Esempio.
Supponiamo di voler creare una serie di componenti visuali da voler visualizzare su Windows 10 e Mac OS Mojave, come per esempio un bottone ed uno slider. In questo caso l'abstract factory è perfetto perché definisce la famiglia di componenti (bottoni e slider) ma poi l'implementazione specifica (per Win10 e Mojave) viene affidata ad altre classi. Vanno create le seguenti interfacce:

```
//A. factory che definisce
la famiglia di componenti
public interface UIFactory {
    Button buildButton();
    Slider buildSlider();
```

```
//Bottone generico
public interface Button { void show(); }
```

```
//Slider generico
public interface Slider { void slide(); }
```

```
}
```

Ora si crea l'implementazione della famiglia di componenti. Partiamo da Windows 10:

```
public class WindowsButton implements Button {
    @Override
    public void show() {
        System.out.println("Windows 10 button");
    }
}

public class MojaveButton implements Button {
    @Override
    public void show() {
        System.out.println("Mac OS Mojave button");
    }
}

public class WindowsSlider implements Slider {
    @Override
    public void slide() {
        System.out.println("Windows 10 slider");
    }
}

public class MojaveSlider implements Slider {
    @Override
    public void slide() {
        System.out.println("Mac OS Mojave slider");
    }
}
```

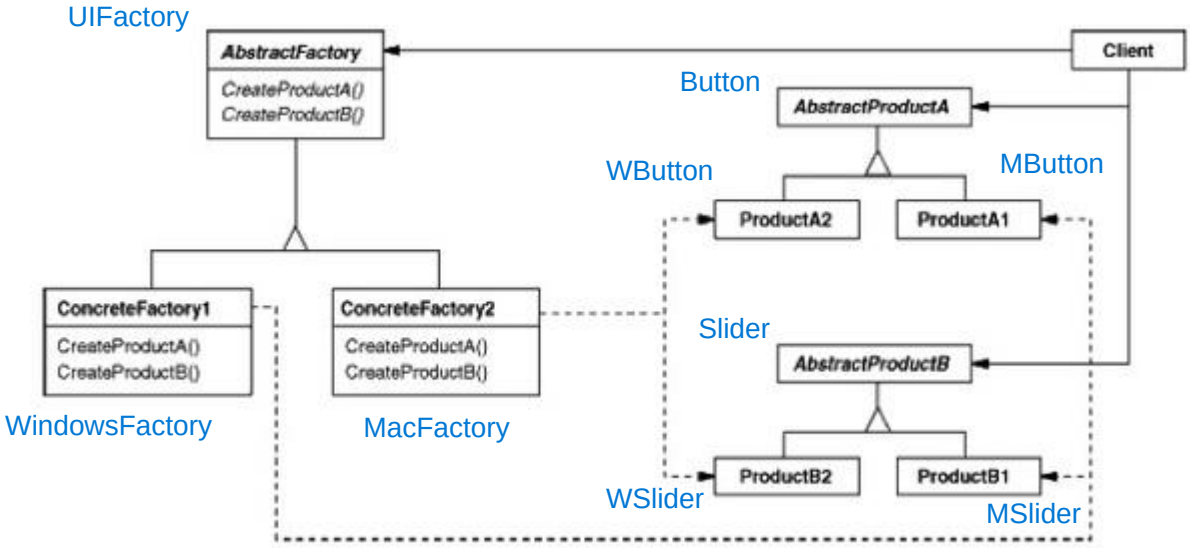
Ora che ho definito sia la famiglia di componenti (`UIFactory`) e le implementazioni (vedi classi sopra) posso costruire il la classe che crea questi componenti e quindi fornisce un'implementazione specifica della famiglia.

```
class Win implements UIFactory {

    @Override
    public Button buildButton() {
       return new WindowsButton();
    }

    @Override
    public Slider buildSlider() {
       return new WindowsSlider();
    }
}
```

```
class Mac implements UIFactory {

    @Override
    public Button buildButton() {
       return new MojaveButton();
    }

    @Override
    public Slider buildSlider() {
       return new MojaveSlider();
    }
}
```

Vediamo dunque che l'interfaccia definita non è dipendete dall'implementazione perche quest'ultima la fornisce la classe specifica per quel componente. In questo modo, ogni famiglia di componenti avrà la sua specifica implementazione.

Diagramma.



DESIGN PATTERN COMPORTAMENTALI

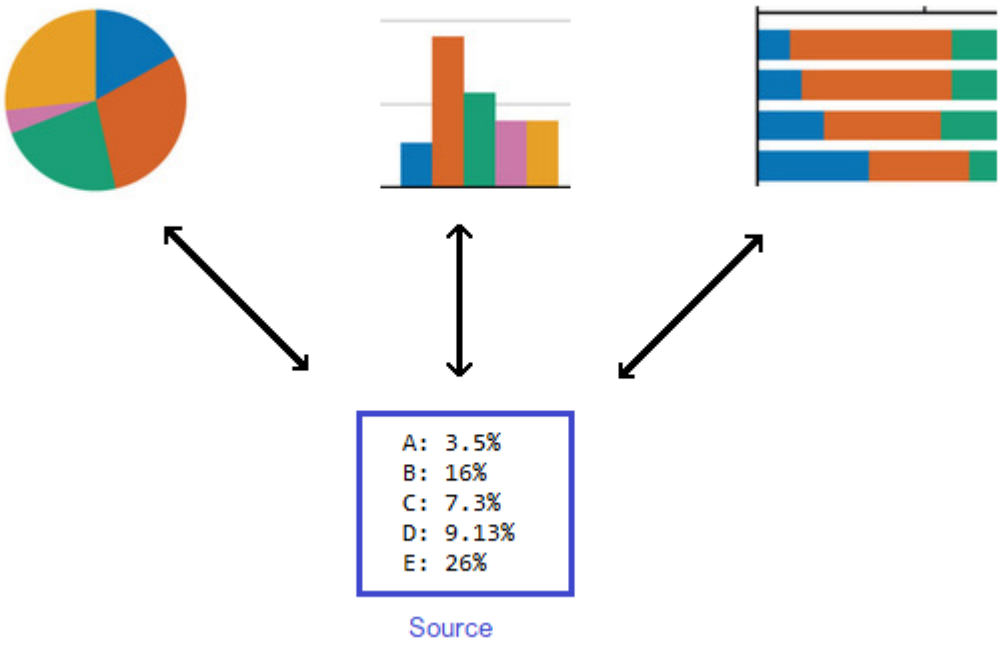
I design pattern comportamentali definiscono come un oggetto svolge la sua funzione ed in che modo i vari oggetti comunicano fra di loro. Si fa un ampio uso di questi design pattern nei framework dei vari linguaggi come ad esempio Spring (per Java, C#...). Ce ne sono 11 in totale ma non verranno analizzati i seguenti: Interpreter, Chain of Responsibility, Mediator, Memento, State e Visitor.

- ## OBSERVER PATTERN

Descrizione.
L'observer pattern definisce una dipendenza di tipo 1..n fra oggetti riflettendo la modifica di un oggetto su di quelli che sono a lui relazionati. Questo serve a mantenere consistenza nel sistema facendo in modo che in seguito ad una segnalazione (ad un **evento**) avvengano degli aggiornamenti. Ci sono due componenti principali:

- *Subject*: una classe che effettua delle modifiche
- *Observer*: una classe che è "in ascolto" sul subject

Quindi la subject è la classe "da guardare" perché compie delle azioni; observer è quella classe che fa qualcosa ogni volta che la subject ha fatto un'azione specifica. Dunque la subject è guardata da tutti gli osservatori (observer) che fanno qualcosa in risposta ad uno specifico evento.



In questo esempio ci sono una fonte di dati (Source) e dei grafici. Ogni volta che la fonte di dati viene aggiornata cambiando delle percentuali, i grafici si devono aggiornare cambiando immagine. Stessa cosa con le classi: ad ogni modifica della classe `Source` avverrà una chiamata ad `update()` degli obsevers (le classi dei grafici) che si aggiorneranno.

Esempio.
Supponiamo di avere un'agenzia di notizie che sta a capo di 2 giornali e li deve notificare quando ci sono delle news. Quando arrivano notizie dal mondo, l'agenzia di notizie le riceve e deve inviare a tutti gli ascoltatori (i suoi giornali associati) queste news. Quindi servirà una classe `AgenziaNotizie` che al suo interno tiene una lista di observers (i giornali) che vengono notificati ogni volta che lo stato della classe cambia (cioè quando arriva una news dal mondo).

1. Si crea un'interfaccia che verrà implementata dagli observers (i giornali)

```
public interface Giornale {
    public void update(Object o);
}
```

2. Poi si creano gli observers, ovvero tutte quelle classi che devono stare in ascolto di modifiche e cambiare il loro stato.

```
class GiornaleUno implements Giornale {
    private String news;

    @Override
    public void update(Object o) {
        this.news = (String)o;
    }
}

class GiornaleDue implements Giornale {
    private String news;
```

```
    @Override
    public void update(Object o) {
        this.news = (String)o;
    }
}
```

3. Ora bisogna creare la classe che è osservata dagli osservatori (cioè GiornaleUno e GiornaleDue) ovvero `AgenziaNotizie`. Bisogna che questa aggiunga ad una lista interna tuti gli osservatori e, usando il metodo notify(), li aggornerà.

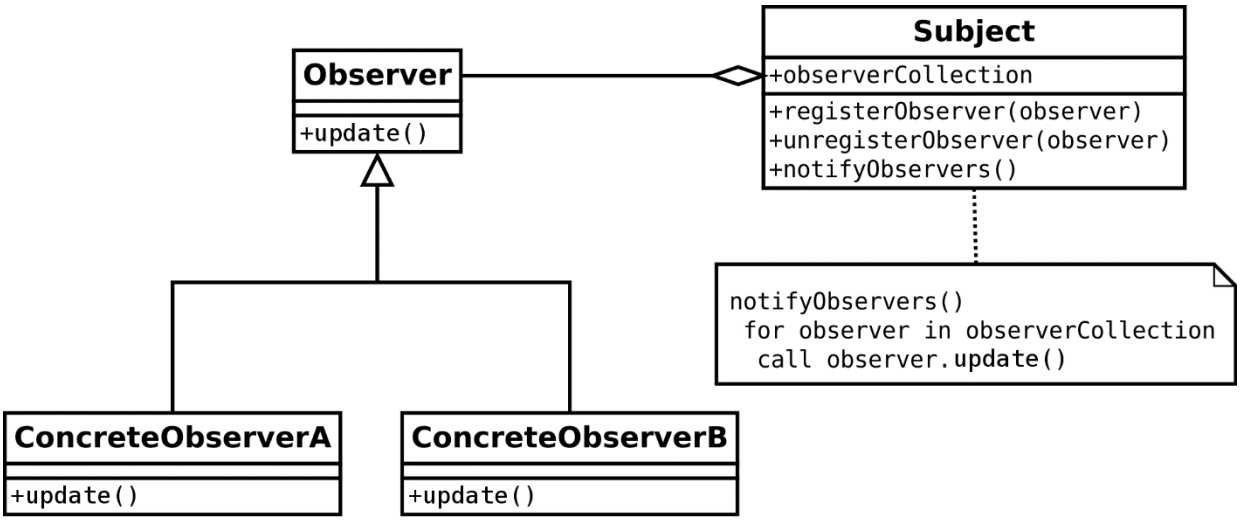
```
public class AgenziaNotizie {

    private String news;

    private List<Giornale> giornali = new ArrayList<>();

    //Aggiungo un giornale all'agenzia, cioè aggiungo un observer
    //che sarà notificato in caso di nuova notizia
    public void addObserver(Giornale giornale) {
        this.giornali.add(giornale);
    }

    //Tolgo osservatori
    public void removeObserver(Giornale giornale) {
        this.giornali.remove(giornale);
    }

    //Viene aggiunta una nuova notizia e dunque lo stato della
    //classe osservata (subject) cambia; bisogna notificare tutti
    //gli observer dell'aggiornamento
    public void setNews(String news) {
        this.news = news;

        for (Giornale giornale : this.giornali)
            giornale.update(news);
    }

}
```

In questo modo ogni volta che viene aggiunta una nuova notizia alla classe che fa da subject (l'osservata) tutti gli observer (gli osservatori) vengono notificati e si aggiornano di conseguenza.

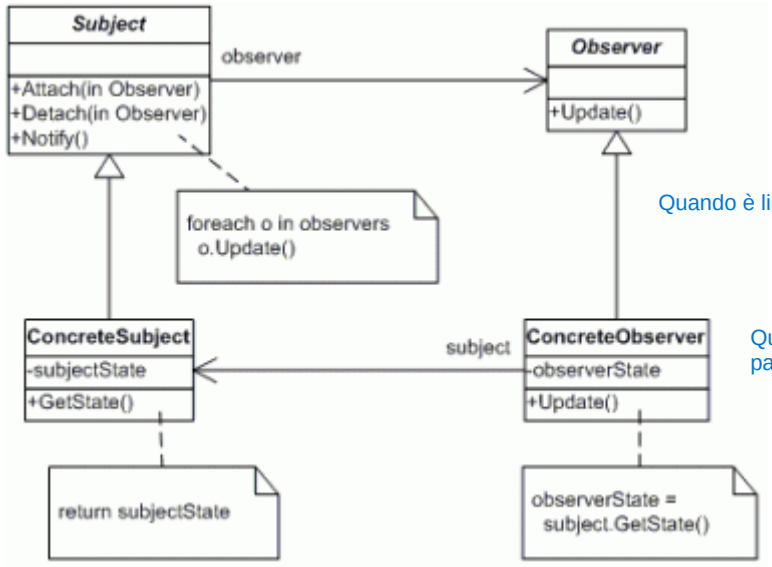
Questo approccio è molto semplificato e in Java non si fa così perché l'API mette a disposizione l'interfaccia Observable; non si fa tutto da zero. Dall'esempio sopra si potrebbe mettere `AgenziaNotizie` come classe astratta, mettere il `notify()` protetto e poi in una sottoclasse mettere il `setNews()` con tutti gli altri metodi. Sarebbe più ordinato e prende il nome di *Concrete Observer pattern*.

Questo pattern si usa molto come base architetturale di molti sistemi che hanno necessità di gestire gli eventi perché appunto questa cosa dell'update/notify serve a mantenere il sistema aggiornato in caso di eventi.

Diagramma.



Observer pattern



Concrete Observer pattern

- COMMAND PATTERN

Descrizione.
Il command pattern incapsula una richiesta in un oggetto cosicché i client siano indipendenti dalle richieste. In altre parole, questo pattern incapsula in un oggetto tutti i dati e i metodi necessari per fare una certa azione (un comando) creando un accoppiamento molto lasco con i client che usano il command. In una classica implementazione del pattern, queste sono le 4 componenti che servono:

1. Il *Command*
2. Il *Receiver*
3. L'*Invoker*
4. Il *Client* che usa il pattern

Esempio.
Facciamo finta di voler creare un semplice editor di testo; ci serviranno le funzionalità

di apertura, scrittura e salvataggio di un *.txt (e molte altre anche ma vabbè). Per fare ciò servono i 4 componenti che abbiamo menzionato sopra:

1. **Command**
   Un *command* è un oggetto che ha il compito di salvare tutte le informazioni necessarie per poter compiere un'azione. Quindi deve contenere tutti i campi e i metodi che permettano di fare qualcosa a pieno.

```
public interface TextFileOperation {
    String execute();
}

class OpenTextFileOperation implements TextFileOperation {

    //La classe TextFile è il receiver, vedi punto 2
    private TextFile textfile;

    //Costruttori, metodi, parametri...

    @Override
    public String execute() {
        return textfile.open():
    }

}

class SaveTextFileOperation implements TextFileOperation {

    //La classe TextFile è il receiver, vedi punto 2
    private TextFile textfile;

    //Costruttori, metodi, parametri...

    @Override
    public String execute() {
        return textfile.save():
    }

}
```

   L'interfaccia definisce l'API per i comandi e le due classi che la implementano fanno le azioni **concrete** e contengono tutti i metodi e dati per fare il loro lavoro (salvataggio e lettura).

2. **Receiver**
   Un *receiver* è un oggetto che fa una serie di azioni coese; è il componente che fa le azioni quando viene chiamato il comando `execute()`. La classe *receiver* la chiameremo `TextFile` e verrà quindi usata per fare le azioni di salvataggio e apertura di file di testo.

```
class TextFile {

    private String name;

    //Costruttori, metodi, parametri...

    public String open() {
        return "Opening the file " + name;
    }
```

```
    public String save() {
        return "Saving the file " + name;
    }

    //Altri metodi per fare altre cose da chiamare in execute()

}
```

3. **Invoker**
   L'invoker è quell'oggetto che sa come eseguire un determinato comando ma non sa come questo comando è stato implementato. Conosce solo l'interfaccia di questo comando.

```
class TextFileOpExecutor {

    private final List<TextFileOperation> textFileOperations =
                  new ArrayList<>();

    public String executeOperation(TextFileOperation txtOp) {
        textFileOperations.add(txtOp);
        return txtOp.execute();
    }
}
```

   Non è necessario che ci sia una lista delle operazioni fatte; è stata messa perché nel caso in cui si voglia fare l'undo di qualcosa oppure un rollback di azioni, si ha uno storico. Il pattern non lo prevede. È necessario però che ci sia il metodo `executeOperation` che esegue l'operazione senza sapere come questa avviene.

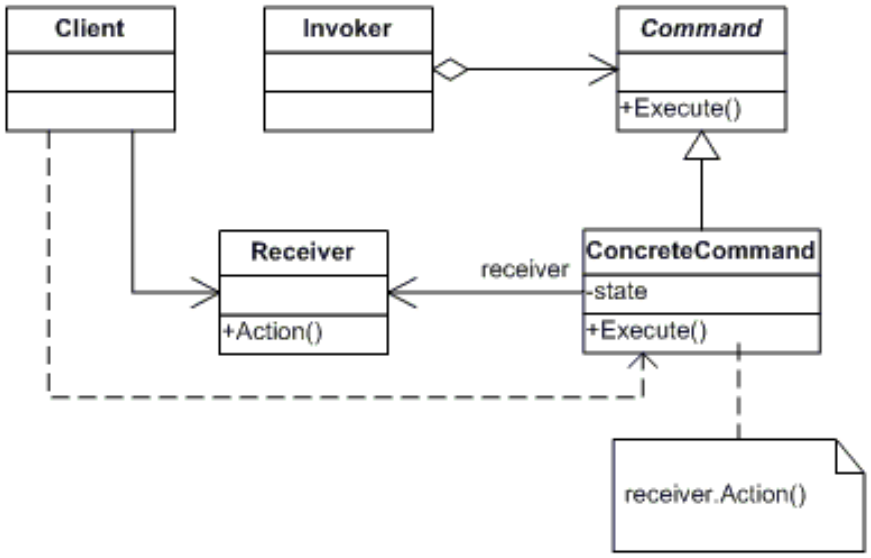
4. **Client**
   Il client è la classe che utilizza i comandi interfacciandosi con l'*invoker* e creando istanze del *receiver*. Dall'esempio si capisce a cosa serve:

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

    TextFileOpExecutor exec = new TextFileOpExecutor();

    exec.executeOperation(
        new OpenTextFileOperation(new TextFile("file.txt"))
    );

    exec.executeOperation(
        new SaveTextFileOperation(new TextFile("file.txt"))
    );

}
```

   Vediamo dunque che i *command* sono le classi che hanno tutto il materiale (campi e metodi) per svolgere le operazioni e si servono dei *receiver* per attuare le operazioni. Poi nel *client* (chi usa i comandi) serve utilizzare l'*invoker* che richiama i *command* per fare le operazioni.

Importante vedere che l'*invoker* prende in input l'**interfaccia** e che quindi non conosce l'implementazione specifica e nemmeno le classi che andrà a richiamare. Questa classe è uno strato molto sottile che distacca i comandi da chi li usa e "centralizza" l'uso. Si usa questo pattern ad esempio se serve uno storico di azioni o se servono callback.

Diagramma.



- ## STRATEGY PATTERN

Descrizione.
Lo strategy pattern definisce una famiglia di algoritmi che possono essere fra di loro interscambiabili. Nei videogiochi ad esempio posso scegliere la difficoltà (facile o difficile) e cambiano dei **comportamenti** ma la **struttura** resta sempre uguale.
Si usa dunque lo strategy quando ho tipi diversi che differiscono tra di loro per il comportamento ma non per l'interfaccia. Mi permette di cambiare il comportamento di un algoritmo a runtime.

Esempio.
Facciamo finta di avere un negozio di dolci e vogliamo applicare diversi tipi di sconti sui pezzi in base al periodo dell'anno, cioè se siamo a Natale, Pasqua o capodanno.
Prima di tutto serve l'interfaccia con i metodi che verranno usati dall'algoritmo:

```
public interface Discounter {
    double applyDiscount(double amount);
}
```

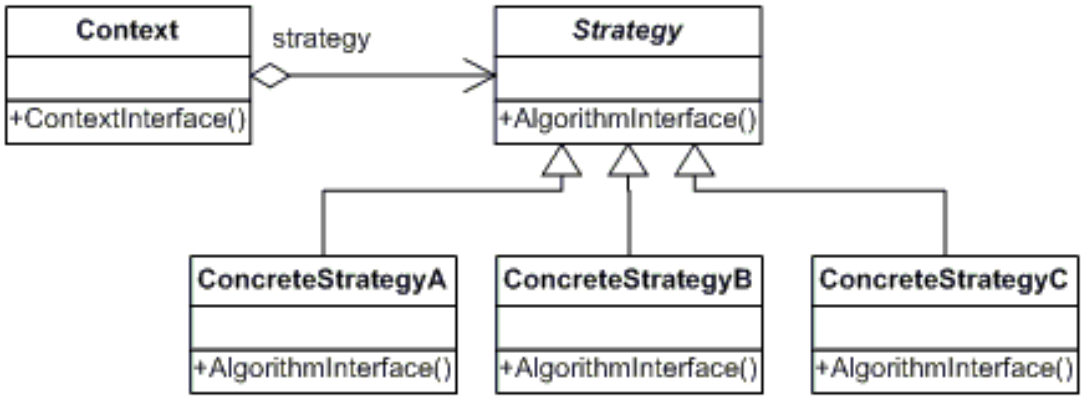
Supponiamo di voler applicare lo sconto del 40% a Pasqua e del 15% a Natale; serve creare gli algoritmi che implementano l'interfaccia.

```
class EasterDiscounter implements Discounter {
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.4;
    }
}

class ChristmasDiscounter implements Discounter {
    @Override
    public double applyDiscount(double amount) {
        return amount * 0.15;
    }
}
```

Lo strategy è finito e basta soltanto nel main creare le istanze che ci interessano degli sconti. La cosa importante da capire è che l'interfaccia definisce la struttura dell'algoritmo che è comune a **tutte** le sottoclassi ma tale algoritmo verrà creato in modo diverso.

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

    Discounter discounter;
    double discount;

    //Calcola lo sconto su della roba da 13 euro nel periodo in cui sono
    if (isEaster)
        discount = (new EasterDiscounter()).applyDiscount(13);
    else
        discount = (new ChristmasDiscounter()).applyDiscount(13);
}
```

Diagramma.



- ## TEMPLATE METHOD PATTERN

Descrizione.
Il template method pattern definisce lo scheletro di un algoritmo e lascia l'implementazione di alcuni passi alle sottoclassi. Ho degli step da seguire di un algoritmo e ci sono delle varianti in alcuni passi; quindi ho una classe astratta con lo scheletro dell'algoritmo, poi le sottoclassi lo raffinano.

Esempio.
Ho a disposizione un account di facebook ed un database remoto; devo gestire il login a queste due risorse. Per effettuare il login (indipendentemente dal fatto che sia su facebook o nel database) devo fare i seguenti passi:

1. Verificare la connessione a internet
2. Validare l'input vedendo se username e password sono conformi
3. Ottenere l'autorizzazione (vedere se il login è valido o no)
4. Ritornare un oggetto User coi dati dell'utente

Questi 4 step sono comuni sia all'azione di login su facebook che all'azione di login sul database; l'unica cosa che cambia è **come** fare il login. Creo quindi una classe astratta che da lo scheletro (cioè "imposta" i 4 step) e poi nelle sottoclassi ridefinisco i metodi di modo che facciano il loro dovere. Esempio:

```
public abstract class LoginManager {

    //Questo è il template method; definisce lo scheletro degli "step"
    //da seguire per l'algoritmo. I metodi astratti sono quelli che
    //vanno implementati nelle sottoclassi perché azioni specifiche
    public User login(String username, String password) {
        //Step 1
        checkConnection();
        //Step 2
        validateInput(username, password);
        //Step 3
        authenticate(username, password);
        //Step 4
        return getUserData(username);
    }

    //Questo metodo dello Step 1 viene implementato qui perché ci dovrà
    //sempre essere in tutte le sotto classi e dato che deve verificare
    //la connessione, lo fa qua perché tanto sempre quella è da fare
    private void checkConnection() {
        if(internetIsNotActive()) {
            throw new RuntimeException("No internet connection");
        }
    }

    //Metodi degli step 2, 3 e 4 che vanno definiti dalle sottoclassi
    protected abstract void validateInput(String username, String pass);
    protected abstract void authenticate(String username, String pass);
    protected abstract User getUserData(String username);

}
```

Ora bisogna creare le sottoclassi che vanno a implementare i metodi astratti di modo da poter definire l'algoritmo con i passi specifici per ogni caso.

```
public abstract class FacebookLogin extends LoginManager {

    private FacekookAPI api;

    protected void validateInput(String username, String pass) {
        if ((username.equals(pass)) {
            throw new Qualcosa("Errore");
        }
    }

    protected void authenticate(String username, String pass) {
        api = new FacebookAPi(username, pass);
        if (!api.login()) {
            throw new Qualcosa("Errore connesione Facebook");
        }
    }

    protected User getUserData(String username) {
        if (api.isAuthenticated())
            return new User(api.getUserData(username));
        else
            return new User();
    }
```

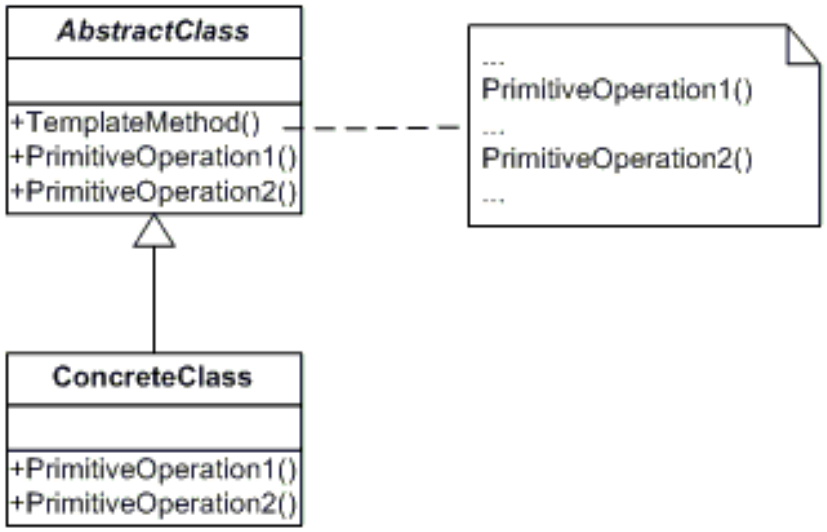
```
}

public abstract class DatabaseLogin extends LoginManager {

    private DatabaseAccess db;

    protected void validateInput(String username, String pass) {
        if (!db.validData(username, pass)) {
            throw new Qualcosa("Errore");
        }
    }

    protected void authenticate(String username, String pass) {
        if (!db.authenticate(username, pass)
            throw new Qualcosa("Errore connessione db");
    }

    protected User getUserData(String username) {
        return new User(db.getData(username));
    }

}
```

Importante dunque capire che il template method è in sostanza una classe astratta che definisce un algoritmo ben preciso con dei passi ordinati da seguire ma alcuni (o anche tutti) di questi passi possono variare da caso a caso. Qui per esempio il passo 1 è necessario a tutti quindi è stato messo diretto nella classe base; gli altri passi sono comuni a tutti ma hanno implementazioni diverse.

Da non confondere con lo Strategy pattern comunque. Nel template method ho passi **uguali** e mi serve uno scheletro **costante**, mentre nello Strategy ho la stessa **famiglia** di algoritmi (che quindi non hanno gli stessi passi). In altre parole:

- Nel template method definisco un algoritmo UNICO che è sempre quello ma cambiano delle cose specifiche dentro a questo, alcuni passi di esecuzione
- Nello strategy definisco una famiglia di algoritmi che hanno lo stesso scopo ma dentro sono fatti in modo diverso

Diagramma.



- ## ITERATOR PATTERN

Descrizione.
L'iterator pattern fornisce l'accesso sequenziale ad elementi di un aggregato senza esporre l'implementazione di quest'ultimo. Si basa sul principio che "per scoprire non è necessario conoscere" e la responsabilità è spostata su un iteratore; stesso concetto del C++.

Esempio.
È abbastanza facile capire cos'è un iteratore perché il concetto è lo stesso del C++ quando si itera su vettori, liste, mappe etc. Prima di tutto si crea l'interfaccia per l'iteratore (che avanza di un posto e ritorna l'oggetto puntato).

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

Poi si crea un'altra interfaccia che va implementata dalla classe che vuole avere un iteratore per ritornare un'istanza dell'oggetto iteratore.

```
public interface Container {
    Iterator getIterator();
}
```

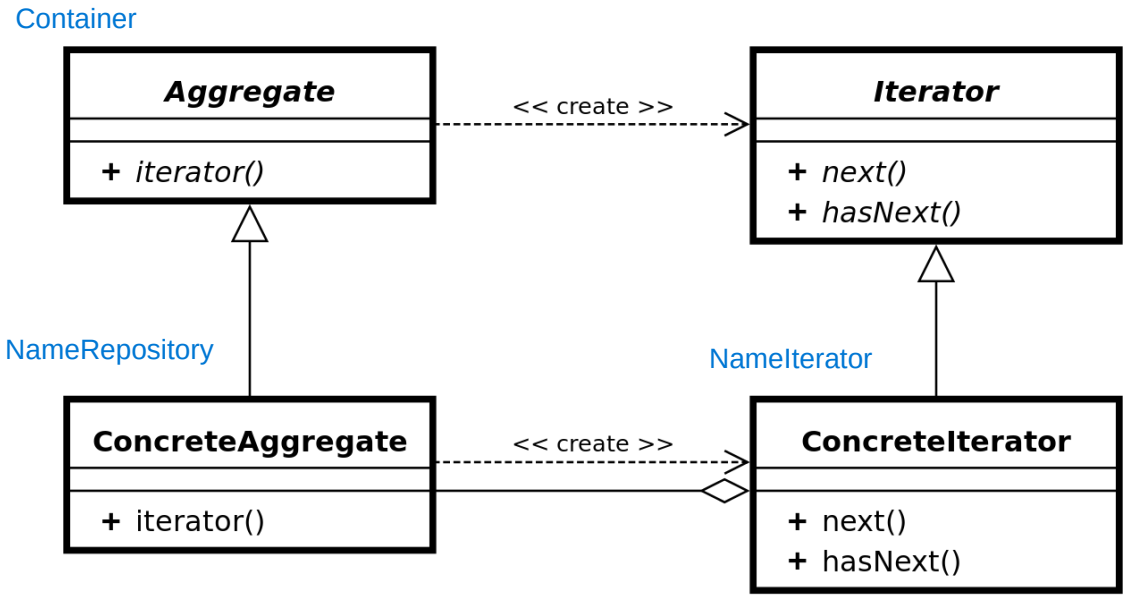
Ora basterà implementare l'interfaccia `Container` per dare il supporto "iteratore" alla classe e sarà necessaria una classe interna che implementa `Iterator`.

```
public class NameRepository implements Container {
    public String names[] = {"Roberto", "Franco", "Patrizio", "Davide"};

    @Override
    public Iterator getIterator() {
        return new NameIterator();
    }

    private class NameIterator implements Iterator {

        int index;

        @Override
        public boolean hasNext() {
            if (index < names.length)
                return true;

            return false;
        }

        @Override
        public Object next() {
            if(this.hasNext()
                return names[index++];

            return null;
        }
    }

}
```

L'utilizzo dell'iteratore può essere ad esempio una cosa di questo genere:

```
NameRepository names = new NameRepository();

for(Iterator it = names.getIterator(); iter.hasNext(); ) {
    System.out.println("Name: " + (String)iter.next());
}
```

Diagramma.



DESIGN PATTERN ARCHITETTURALI

I design pattern architetturali hanno servono a separare il comportamento di una componente dalla risoluzione delle sue dipendenze. Collegare due componenti in modo esplicito ne aumenta l'accoppiamento e dunque diventano difficili gli unit-test ed il mantenimento. I due design pattern di questo gruppo sono l'MVC ed il DI.

- ## DEPENDENCY INJECTION

Descrizione.
L'obiettivo è quello di minimizzare il grado di dipendenza fra le parti di un sistema e, quando c'è necessità di dipendenza, questa va trattata con molta attenzione. La DI mostra delle tecniche per poter "passare" delle dipendenze ad una classe cercando di far sì che l'accoppiamento fra le due parti sia il più lasco (o "leggero") possibile.

Esempio.
Supponiamo di avere una classe che ritorna una lista di film presenti in una libreria con la seguente interfaccia (molto semplificata ovviamente).

```
public class MovieLister {

    private MovieFinder finder;
```

```
    public MovieLister() {
        finder = new MovieFinder();
    }

    public List<String> trovaFilm(String autore) {
        return finder.searchByAuthor(nome);
    }
}
```

Ho la classe `MovieLister` che serve a trovare i film ma in realtà ho una dipendenza da `MovieFinder` perché è lei che fa tutto il lavoro; devo togliere questa dipendenza così forte. In questo esempio inoltre la classe `MovieLister` deve **sapere** come fare ad inizializzare `MovieFinder` perché fa la new.

Il problema è che se mai dovessi cambiare il costruttore di `MovieFinder` per qualche motivo, allora dovrei cambiare anche la segnatura del costruttore di `MovieLister` perché il modo con cui chiamo la `new` sul `finder` devo cambiarlo. In un grosso progetto con tante più classi, questo tipo di dipendenze fa un sacco di casini. Le soluzioni sono:

1. Constructor Injection

   Dichiaro come parametri del costruttore le dipendenze e le assegno alle variabili interne. Questa soluzione è molto valida perché **toglie** l'onere alla classe "esterna" di sapere come dover costruire le sue dipendenze.

   ```
   public MovieLister(MovieFinder finder) {
       this.finder = finder;
   }
   ```

   In questo modo, per creare un oggetto di tipo `MovieLister`, sarò costretto a fare una cosa come la seguente:

   ```
   MovieFinder finder = new MovieFinder( ... );
   MovieLister lister = new MovieLister(finder);
   ```

   Questo approccio è il migliore e va scelto sempre quando possibile; la dipendenza viene passata "da fuori" e così la classe che la contiene (`MovieLister`) non si deve preoccupare di come gestire la sua dipendenza (cioè di come crearla).

2. Setter injection

   Valgono tutti i discorsi di prima sul fatto che le dipendenze è meglio passarle da fuori invece che gestirle internamente però cambia il modo con cui passarle.

   ```
   public void setMovie(MovieFinder finder) {
       this.finder = finder;
   }
   ```

   Invece di passare la dipendenza tramite costruttore, la passo tramite un metodo setter della classe. Questo è indice del fatto che la dipendenza verso `MovieFinder` non è così forte. Prima passavo la dipendenza via costruttore, quindi ero **obbligato** a fornire l'istanza di `MovieFinder` perché indispensabile

per creare l'oggetto `MovieLister`, ora invece passo la dipendenza via metodo rendendo più "leggero" il legame. Il metodo lo posso chiamare quando voglio, oppure non chiamarlo proprio, e quindi questo indica che si c'è una dipendenza ma non è essenziale per la vita dell'oggetto che la contiene.

Potrebbe accadere che ci si dimentica di chiamare il setter e quindi avere da qualche parte la famosa `NullPointerException`. È preferibile usare la constructor injection perché mi imposta subito tutte le dipendenze (anche se non sono proprio necessarie al 100%) e non mi dimentico di niente.

• MODEL VIEW CONTROLLER (MVC)

Descrizione.
Si tratta di un pattern architetturale da utilizzare nelle applicazioni nelle quali è necessaria la presentazione di informazioni tramite l'uso di una UI (User Interface). Per tenere tutto organizzato e specialmente per ridurre le dipendenze si cerca di separare bene i ruoli di ciascuna componente. In un sistema quindi che ha sia dei dati da gestire che una UI sulla quale mostrare cose si individuano

- **Model**: i dati di *business* e le regole di accesso a questi
- **View**: la rappresentazione grafica dei dati del modello
- **Controler**: collegamento tra view e model

Nel particolare, ciascuna componente del pattern MVC serve a:

- **Model:** *business* del programma con tutte le classi realizzate tramite l'uso di design pattern. Questa parte si occupa di processare le informazioni ed elaborare i dati; grosso modo è la parte che "fai i conti". La caratteristica del model è che deve essere **indipendente** dal sistema. Deve essere un blocco che non va a toccare la view, ha tutto dentro a sé. Il model notifica alla view gli aggiornamenti avvenuti sui dati utilizzando l'observer pattern.

- **View:** è la parte visiva del programma, cioè la GUI che interagisce con l'utente. Cattura gli input passati e delega al controller l'elaborazione di questi; essa viene aggiornata costantemente grazie all'utilizzo dell'observer pattern. Mantiene un riferimento al model ma non comunica direttamente con lui.

- **Controller:** è la parte che realizza la logica dell'applicazione perché trasforma le interazioni dell'utente in azioni. In pratica, prende i dati dalla GUI, li elabora e poi li passa al model.

Il controller è quello che comunica sia con la view che con il model perché è quello che deve fare da tramite. Importante notare che esiste un controller per ogni view quindi se ci fossero 1 model (1 fonte di dati) e 2 views (2 GUI) allora servirebbero anche 2 controller.

Quasi sempre la relazione che c'è fra view e model avviene attraverso l'utilizzo dell'Observer pattern; quanto è necesario cambiare il comportamento dell'applicazione a seconda delle circostanze, il controller usa anche lo Strategy.

Esempio.
Vediamo ora un esempio basilare di MVC senza l'uso di Observer o Strategy, proprio facile per avere l'idea di come funziona. L'esempio mostra a schermo i dati degli studenti universitari inseriti da un docente. Questa classe è il **model:**

```
public class Student {

    private String studentId;

    private String name;

    public String getStudentId() {
        return studentId;
    }

    public void setStudentId(String studentId) {
        this.studentId = studentId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Vengono qui gestiti tutti i dati e forniti i metodi per agire su di essi; si tratta della parte di "business" ovvero quella che gestisce le informazioni. Adesso si crea la view:

```
public class StudentView {
    public void printStudentDetails(String studentName, String
studentId){
        System.out.println("Student: ");
        System.out.println("Name: " + studentName);
        System.out.println("Id Number: " + studentId);
    }
}
```

Infine basta creare il controller, ovvero quella classe che si occupa di aggiornare il model e la view in base alle richieste inviate dal client tramite la GUI.

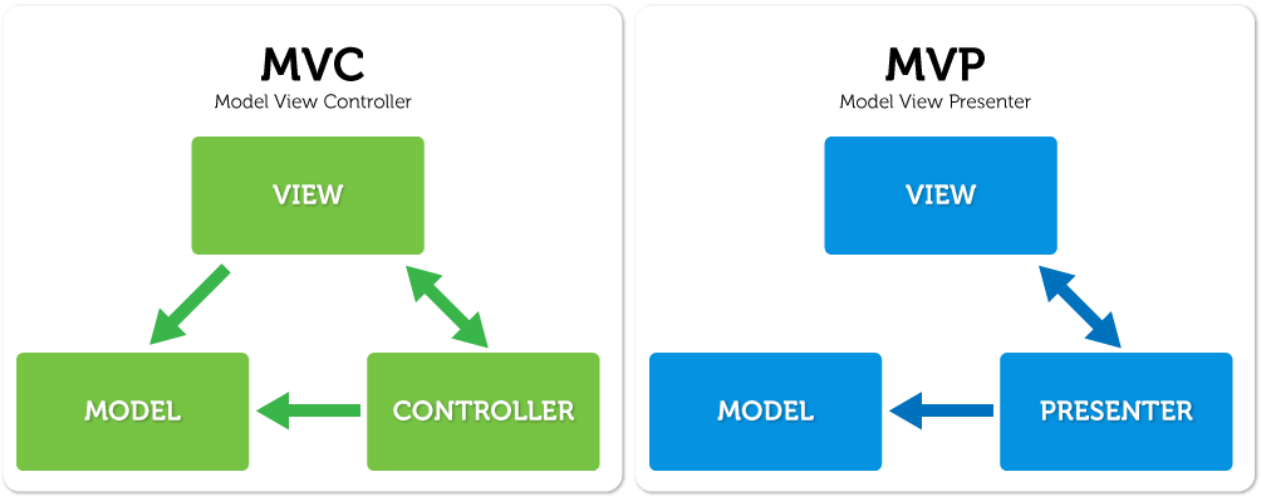
```
public class StudentController {

   private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view){
        this.model = model;
        this.view = view;
    }

    public void setStudentName(String name){
        model.setName(name);
    }

    public String getStudentName(){
        return model.getName();
    }
```

```
    public void setStudentRollNo(String rollNo){
        model.setStudentId(rollNo);
    }

    public String getStudentRollNo(){
        return model.getStudentId();
    }

    public void updateView(){
        view.printStudentDetails(model.getName(), model.getStudentId());
    }
}
```

Ora nel main posso usare il pattern MVC creando le tre istanze e, per lavorare sui dati, utilizzo il controller per fare quello che mi serve. Lui infatti prende un riferimento ai dati e alla grafica (il controller che elabora e la GUI che prende l'input) e li sincronizza.

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

        //Creo il model caricando i dati
        Student md = loadStudentsFromDatabase();
        //Creo la view
        StudentView sv = new StudentView();
        //Creo il controller
        StudentController sc = new StudentController(md, sv);

        //Mostra: Roberto - 01
        sc.updateView();

        //Mostra: Roberta - 01
        sc.setStudentName("Roberta");
        sc.updateView();
}

    public static Student loadStudentsFromDatabase() {
        Student s = new Student();
        s.setStudentId("01");
        s.setName("Roberto");
        return s;
}
```

C'è tutta una famiglia di pattern che si chiamano MV* (MV*star*) che prevedono sempre il model, quindi i dati, e la view, cioè la presentazione, ma hanno modi diversi per far comunicare le due.



Questo è un esempio classico. L'MVC è il pattern classico più conosciuto che è stato visto fino ad adesso. L'MVP è una variante (che si usa quando si fanno app Android ad esempio) nella quale la view **non** ha il riferimento al model. Dire *presenter* è un altro modo di dire *controller*.

Mentre nel pattern MVC la view è "intelligente" perché ha un riferimento al model per poter controllare lo stato, nel pattern MVP la view è "stupida" perché è il controller che deve fare proprio tutto. Di solto nell'MVC la view è una classe che ha uno stato, nell'MVP la view è qualcosa tipo un file XML che quindi non fa cose (ecco perché non c'è il riferimento al model).

RIASSUNTO DEI DESIGN PATTERN

Strutturali

- **Adapter**: serve a convertire l'interfaccia di una classe in un'altra; le varianti sono object adapter (con interfacce) oppure class adapter (con ereditarietà).
- **Decorator:** serve ad aggiungere funzionalità ad un oggetto dinamicamente; si ingloba un oggetto in un altro che offre più funzionalità
- **Facade:** serve a fornire un'interfaccia unica e semplice per un sottoinsieme complesso di oggetti; fa da *single failure point*
- **Proxy:** serve a fornire un surrogato di un altro oggetto del quale si vuole controllare l'accesso; rinvia il costo di creazione di un oggetto all'effettivo utilizzo

Creazionali

- **Singleton:** ha lo scopo di assicurare l'esistenza di un'unica istanza di una classe; fornisce un punto di accesso unico ad una risorsa
- **Builder:** separa la costruzione di un oggetto complesso dalla sua rappresentazione; serve ad evitare l'effetto di *telescoping* nei costruttori delle classi
- **Abstract factory:** fornisce un'interfaccia per creare famiglie di prodotti senza dovere specificare le classi concrete

Comportamentali

- **Observer**: definisce una dipendenza di tipo *1 . . n* fra le componenti riflettendo la modifica di un oggetto su di quelli che sono a lui relazionati; crea un modello ad eventi
- **Command**: incapsula una richiesta di un oggetto in modo che i client siano indipendenti dalle richieste; c'è molto disaccoppiamento fra client ed implementazione
- **Strategy**: definisce una famiglia di algoritmi che sono fra loro interscambiabili; di tali algoritmi cambiamo i comportamenti ma la struttura resta sempre uguale
- **Template method**: definisce lo scheletro di un algoritmo e lascia l'implementazione di alcuni passi alle sottoclassi; differisce dallo strategy perché qua c'è un flusso di esecuzione comune a tutti gli algoritmi mentre prima no
- **Iterator**: fornisce l'accesso sequenziale ad elementi di un aggregato senza esporre l'implementazione di quest'ultimo; stesso concetto degli iteratori del C++

Architetturali

- **Dependency injection:** minimizza il grado di dipendenza fra le parti di un sistema cercando di "iniettare" (passare) le dipendenze dall'esterno, di modo che la classe abbia il minor numero di responsabilità possibile. Ci sono *constructor injection* per le dipendenze forti e la *setter injection* per le dipendenze deboli, ma sarebbe da preferire sempre la prima
- **Model View Controller:** organizza la gestione di un sistema che utilizza un'interfaccia grafica per interagire con il client (l'utente) ed un modello di dati alla base. Si individuano tre componenti principali: il *model*, la parte di business del programma, la *view*, la parte visiva del programma ed il *controller*, la logica dell'applicazione che comunica con *view* e *model*.

S.O.L.I.D. PRINCIPLES

Il nome SOLID indica le iniziali di 5 principi che aiutano il programmatore a gestire le dipendenze all'interno di un sistema. Si parte sempre dal principio che tutte le componenti devono avere un accoppiamento che sia il più lasco possibile. I primi 3 che formano le lettere S.O.L. sono fondamentali e il rispetto di questi garantisce molto probabilmente anche la conformità agli altri due I e D.

- ## **S**ingle Responsibility Principle

  Il SRP dice che una classe deve avere una ed una sola singola responsabilità; **una classe deve avere un solo motivo per cambiare**. In altre parole, ogni oggetto deve fare una sola cosa (come si capisce anche dal nome del principio). Bisogna analizzare tutto il contesto e non la singola classe per capire se si aderisce al principio SRP.

  Supponendo di dover creare un programma che disegna figure geometriche e fa conti, non conviene avere una classe unica che fa calcoli aritmetici, disegna cose nella GUI e magari si connette anche ad un database per salvare uno storico di figure. Maggiori sono le responsabilità di una classe, maggiore sarà la probabilità che ci sarà bisogno di doverla cambiare. Si punta ad avere oggetti piccoli e concisi, facili anche da testare.

- ## **O**pen Closed Principle

  L'OCP dice che in una buona architettura software bisognerebbe essere in grado di estendere il comportamento delle classi senza modificarle. Questo concetto dà il nome al principio perché lo scopo è quello di essere aperti al cambiamento ma chiusi alle modifiche. In pratica vuol dire che non si cambia il codice vecchio per fare cambiamenti, ma se ne aggiunge di nuovo mantenendo il vecchio.

  La chiave di tutto è l'**astrazione** che a livello di codice si ottiene con l'uso delle interfacce o delle classi astratte. Solitamente si fa uso di interfacce che permettono di definire l'interfaccia ma non l'implementazione e quindi l'aggiunta di nuove componenti non implica la modifica di ciò che già esiste. Vediamo un esempio semplice riguardante una classe che calcola l'area di figure geometriche.

```
class Rectangle {
    private double height;
    private double width;

    public Rectangle(double h, double w) {
        this.height = h;
        this.width = w;
    }

    public double getHeight() {
        return height;
    }

    public double getWidth() {
        return width;
    }
}

class Circle {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
}

class ShapeManager {
    public double calculate(Object shape) {
        if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.getHeight() * r.getWidth();
        } else {
            Circle c = (Circle)shape;
            return c.getRadius() * c.getRadius() * Math.PI;
        }
    }
}
```

Questo codice non va bene perché per come è fatto, se dovessi aggiungere una classe `Square`, allora dovrei modificare il metodo `calculate` (perché non c'è un controllo sul tipo `Square`). Per risolvere si fa così:

```
interface Area {
    double getArea();
}

class Rectangle implements Area {
    private double height;
    private double width;

    public Rectangle(double h, double w) {
        this.height = h;
        this.width = w;
    }

    @Override
    public double getArea() {
        return height * width;
    }
}
```

```
class Circle implements Area {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return radius * radius * Math.PI;
    }
}

class ShapeManager {
    public double calculate(Area shape) {
        return shape.getArea();
    }
}
```

Grazie all'uso dell'interfaccia, ora posso aggiungere quante classi voglio (a patto che implementino `Area` ovviamente). In questo modo il codice è chiuso alle modifiche, perché non devo toccare il metodo `calculate` come prima, ed aperto alle estensioni, perché basta implementare `Area` per aggiungere tipi.

- ## Liskov Substitution Principle

Il LSP dice che le classi derivate devono essere sostituibili alle loro classi base. In una classe derivata le precondizioni devono essere più forti di quelle della classe base; diciamo che la derivazione deve avere le precondizioni della classe padre più altre che le servono.

Tipico esempio: si supponga di avere la classe `Rectangle`, con i setter ed i getter per l'altezza e la larghezza, ed una classe `Square` che è sottoclasse di `Rectangle`. Ci si aspetta ovviamente che i lati del quadrato abbiano dimensione uguale. Se si usa tuttavia una cosa tipo `Rectangle x = new Square()` si possono usare ancora i setter ereditati dal padre per modificare i lati e dunque si potrebbe avere un quadrato con i lati diversi. Questo è un chiaro esempio di rottura del principio LSP anche perché se modificassi i setter di `Square`, allora violerei le post condizioni della classe `Rectangle` dato che questa **non** ha il vincolo di avere i lati uguali.

- ## Interface Segregation Principle

Design by Contract

L'ISP dice che un client non deve essere forzato ad implementare un'interfaccia che non deve usare. Il succo del discorso è: creare delle interfacce con pochi metodi e molto specifiche. Mettere tanti metodi in una interfaccia può essere controproducente perché si rischia di avere classi che non dispongono di tutte quelle funzionalità.

Facciamo un esempio molto semplice nel quale vogliamo modulare le attività lavorative di un uomo e di una macchina:

```
interface Worker {
    void work();
    void sleep();
}
```

```
class Human implements Worker {

    public void work() {
        System.out.println("I do a lot of work");
    }

    public void sleep() {
        System.out.println("I need to sleep or I'll die ☹");
    }

}

class Robot implements Worker {

    public void work() {
        System.out.println("I work a lot... beep...");
    }

    public void sleep() {
        //I robot non vanno a letto a dormire!
    }

}
```

Il problema che c'è nel codice sopra è che ho un'interfaccia con troppi metodi dentro per il contesto in cui mi serve. L'essere umano lavora e va a letto a dormire per riposare ma il robot non ha bisogno del sonno. Dunque mi trovo con un metodo da lasciare vuoto e non va bene. La soluzione è:

```
interface Worker {
    void work();
}

interface Sleeper {
    void sleep();
}

class Human implements Worker, Sleeper {

    public void work() {
        System.out.println("I do a lot of work");
    }

    public void sleep() {
        System.out.println("I need to sleep or I'll die ☹");
    }

}

class Robot implements Worker {

    public void work() {
        System.out.println("I work a lot... beep...");
    }

}
```

Questo ovviamente non vuol dire che le interfacce debbano avere un metodo solo ma mostra il fatto che esse devono avere il minimo indispensabile di metodi. Averne troppi può causare problemi e addirittura in alcuni casi portare alle stesse rogne che si possono avere con l'ereditarietà da classi concrete.

- **D**ependency Inversion Principle

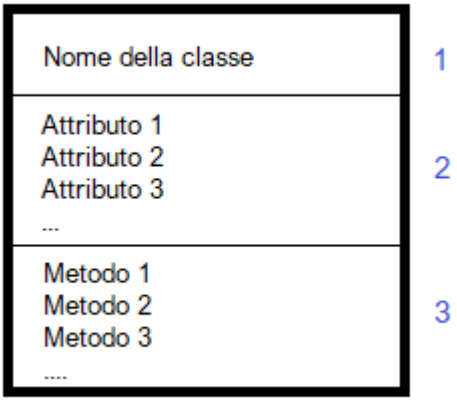
  Il DIP dice che bisogna dipendere dalle astrazioni e non dalle concretizzazioni. Molto semplicemente indica il fatto che conviene usare spesso interfacce e classi astratte per definire la forma (e poi implementare o ereditare da queste) invece che usare subito una classe concreta che definisce il comportamento.

DIAGRAMMI UML

L'UML, *Unified Modeling Language,* è un linguaggio di modellazione basato sul paradigma OOP che utilizza una serie di notazione grafiche per descrivere il progetto dei sistemi software. Si tratta di una astrazione che non scrive il codice ma lo descrive attraverso figure e segni. Ci sono diversi tipi di diagrammi che verranno analizzati di seguito.

Diagramma delle classi

Questo tipo di diagramma serve a descrivere come è fatta una classe senza specificarne l'implementazione. Si usa un rettangolo diviso da delle linee che individuano tre zone:



1. Parte obbligatoria che contiene il nome della classe. Ci può essere anche solo questo primo rettangolo e basta
2. Parte opzionale che contiene i campi/tipi della classe, cioè le variabili con il loro tipo (int, double, String...)
3. Parte opzionale che contiene i metodi della classe col rispettivo tipo di ritorno

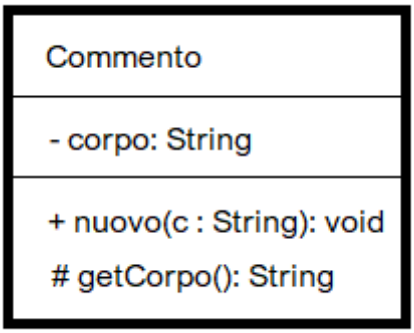
Nelle parti 2 e 3 devo usare una certa sintassi per indicare sia la visibilità dell'item che il tipo dell'attributo o il tipo di ritorno.

ATTRIBUTI	**METODI**
`visibilità NOME: tipo`	`visibilità NOME(parametri): tipo-ritorno`

Dove `tipo` e `tipo-ritorno` sono delle stringhe che indicano appunto il tipo come int, double, char etc., mentre `visibilità` può essere uno di questi quattro simboli:

- + per indicare *public*
- # per indicare *protected*
- ~ per indicare *package*
- - per indicare *private*

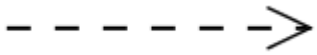
Quindi per esempio `private int test(int a)` diventa in UML - *test(a: int): int* e le stesse regole valgono anche per gli attributi. Esempio:




```
public class Commento {
    private String corpo;

    public Commento() { ... }

    protected String getCorpo() {
        return corpo;
    }

    public void nuovo(String c) {
        corpo = c;
    }
}
```

Notare che l'ordine in cui sono scritti gli item nei rettangoli non è importante quindi non è necessario che sia rispettato anche nella classe scritta in codice. Si possono anche descrivere le dipendenze che ci sono tra le varie classi e per farlo si usano dei particolari tipi di frecce.

- Dipendenza

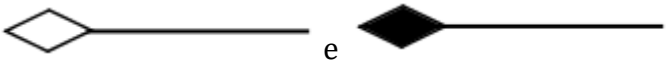
  Quando gli oggetti di una classe usano brevemente gli oggetti di un'altra classe; ad esempio quando una classe A crea o usa brevemente un oggetto che sta in B. Posso aggiungere l'attributo *<<create>>* sulla freccia per specificare che fa una `new`.

- Associazione "ha un"

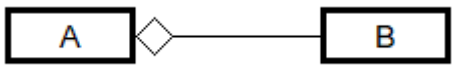
  

  La uso per specificare se ci sono dei tipi non primitivi che lavorano con la classe ma non è obbligatorio fare così. In pratica è come se avessi anziché `int a;` un `Nummero a;` dove `Numero` è per esempio un tipo `enum` definito da parte.

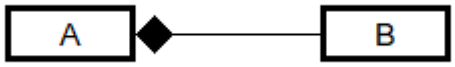
- Aggregazione e Composizione "has a"

  Nel caso dell'**aggregazione** si ha che una classe A contiene un oggetto B ma condivide la reference ad esso con altre classi. Vuol dire che A contiene dentro un campo di tipo B che viene assegnato da fuori (cioè A non fa la new di B, ma l'istanza di B viene passata da fuori con constructor injection o setter injection per esempio).

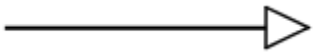
  

  Nel caso della **composizione** si ha che una classe A contiene un oggetto di tipo B, nel senso che dentro ad A c'è un campo di tipo B e ne viene fatta la new. A crea B e lo gestisce dentro.



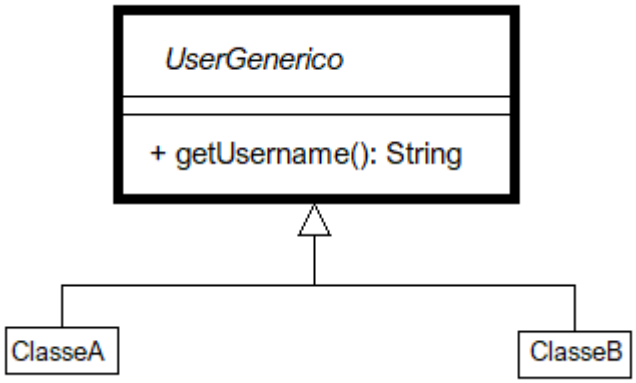
In entrambi i casi comunque si parla di references ad oggetti; la differenza sta nel fatto che col rombo vuoto (aggregazione) l'oggetto non possiede la reference (non fa la new e l'oggetto è gestito da altri) mentre nel caso del rompo pieno (composizione) l'oggetto possiede la reference (fa la new e gestisce l'oggetto).

- Ereditarietà



Quando una classe è sottoclasse di un'altra, è il tipico caso di ereditarietà.

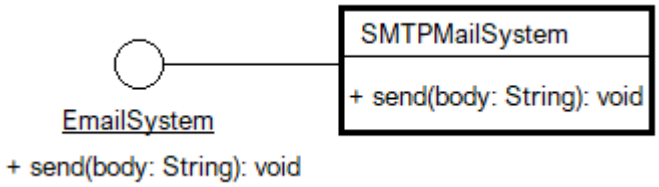
Fra le caratteristiche principali di UML c'è ovviamente anche la possibilità di descrivere le classi astratte e le interfacce. In particolare:



Questo è un esempio di classe astratta di nome `UserGenerico`.

La caratteristica delle classi astratte è la presenza della doppia linea sotto al nome della classe e il fatto che il nome va scritto in corsivo.

Ovviamente ci possono essere attributi e metodi come in una classe normale e valgono le stesse regole.



Questo è un esempio di interfaccia di nome `EmailSystem`.

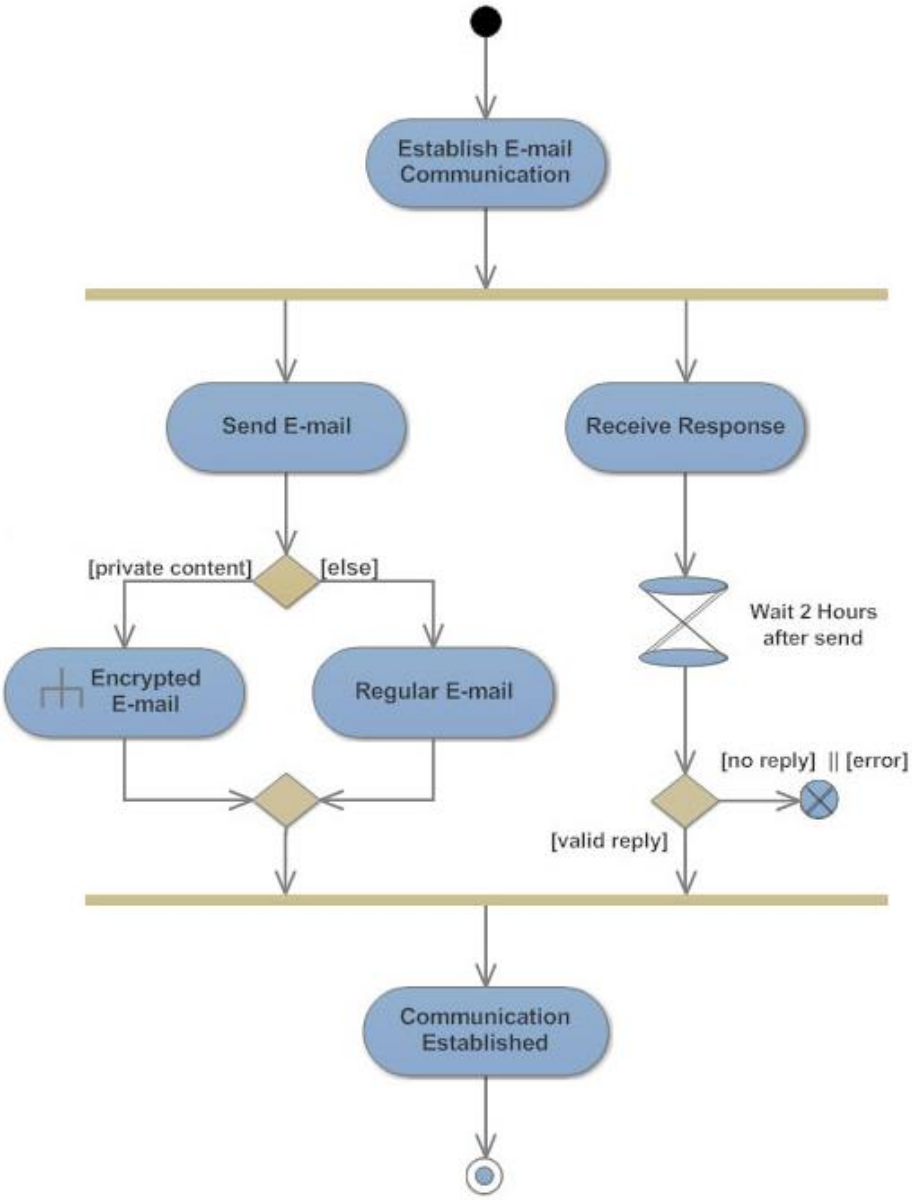
Le interfacce si rappresentano con la *ball notation* che prevede un pallino con sotto il nome sottolineato. I metodi seguono le convenzioni solite che si applicano alle classi.

Un'ultima cosa da ricordare per i diagrammi delle classi è che se sottolineo il nome di un attributo, di un metodo e/o di una classe allora vuol dire che quell'item è **static**, quindi *statico* nel senso che c'è una sola istanza nel programma.

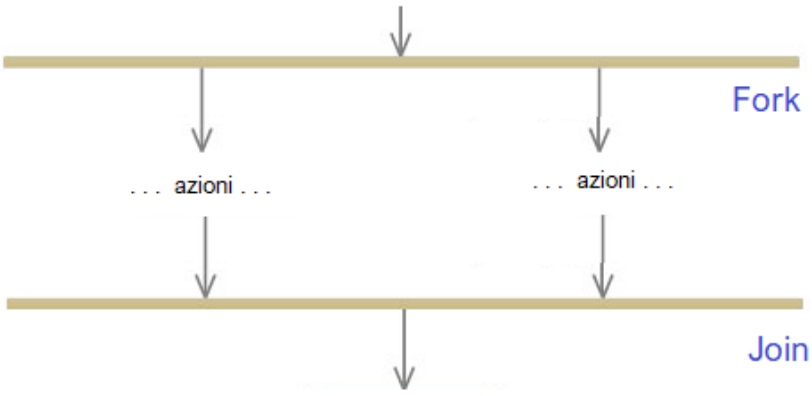
Diagramma di attività

Con UML si può anche descrivere **come** le caratteristiche di una architettura interagiscono nel tempo e descrivono algoritmi. Un diagramma di attività è un insieme di più azioni che partono da un nodo iniziale (pallino nero) che deve seguire dei percorsi per arrivare fino alla fine. Si

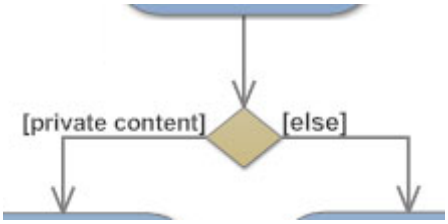
tratta della “descrizione visuale” di un algoritmo. In pratica si fa un disegno e bisogna far sì che il pallino nero possa raggiungere la fine (◉) passando per tutti gli “ostacoli” che possono essere degli `if`, delle esecuzioni asincrone, dei timer o degli errori.



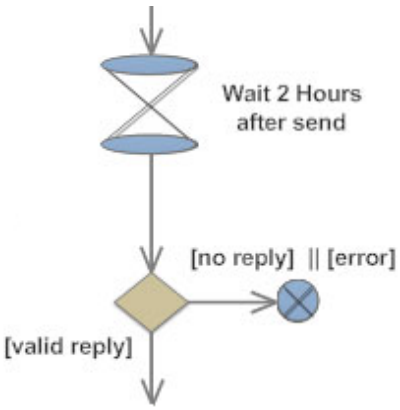
Si comincia sempre col pallino nero iniziale che segue le frecce, passa i vari “ostacoli” e deve **sempre** arrivare al pallino finale. I rettangoli azzurri raffigurano le azioni da compiere e le frecce collegano fra di loro le varie azioni; il pallino segue le frecce ed entra/esce dai rettangoli. Il pallino nero indica il *flow* di esecuzione.



La prima sbarra indica un **fork** ovvero l'inizio di un'esecuzione in parallelo; la seconda barra indica il **join** ovvero il punto di sincronizzazione, la fine di un'esecuzione parallela. Ci sono due pallini che escono dalla sbarra di fork e poi non ha importanza il primo che arriva alla sbarra di join perché si aspetta che tutti arrivino.



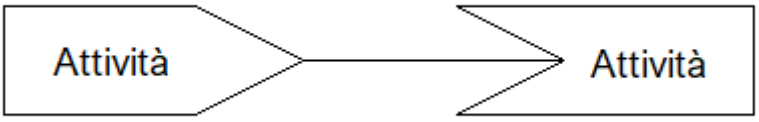
Il diamante indica un **branch**, ovvero un *if* e le condizioni sono specificate tra parentesi quadre; instrada il pallino in uno dei due rami e dopo c'è il rombo finale, il **merge**, che riunisce i rami e fa riprendere il flow.



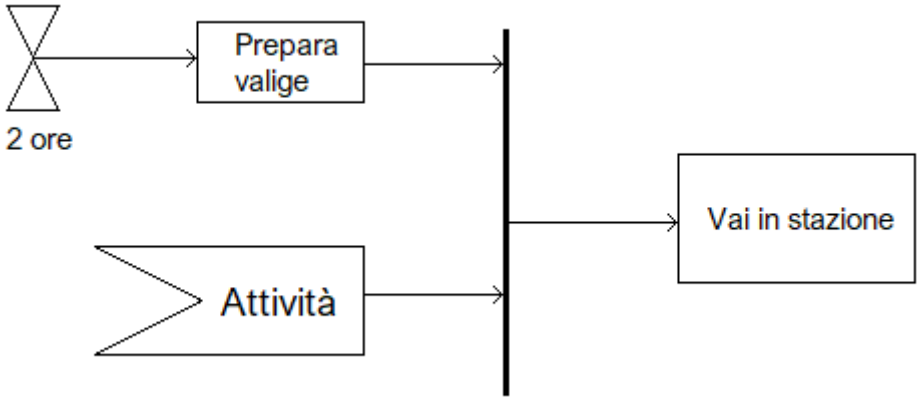
Nella figura c'è una clessidra che è un **time signal**; si tratta di un timer che genera un token nuovo dopo un certo intervallo temporale oppure ne tiene uno fermo. Nel branch in questione si vede che si può andare a sinistra in caso di errore o non risposta e si finisce in un nodo **killer** (⊗). Questo nodo semplicemente mangia il pallino nero senza bloccare l'esecuzione.

Il primo token (pallino) che raggiunge il nodo terminale ◉ blocca tutta l'esecuzione e l'algoritmo quindi termina

Se nel flusso ci sono due pallini che corrono, il primo che arriva alla fine blocca tutto anche se l'altro è ancora in giro. Quando si finisce in un ⊗ il token muore ma non blocca tutto, l'arrivo ad un ◉ invece ferma tutto.

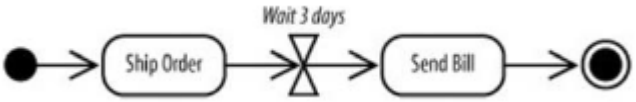


Può succedere che ci sia un rettangolo fatto così ad "incastro" e sta ad indicare che una attività esce dal diagramma per fare dell'altro e poi rientra riprendendo il suo cammino. Il secondo pezzo di destra è bloccante, cioè blocca il flusso fino a che no rientra il token che è uscito. Vediamo ora un esempio che include queste ultime azioni:

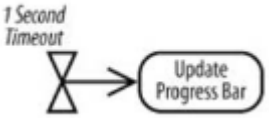


C'è un segnale di tempo che fa aspettare 2 ore, poi fa preparare le valige per la stazione e dopo si arriva ad una **join**, dunque bisogna attendere che anche l'altro ramo arrivi. Sotto c'è una azione "ad incastro" che attende il verificarsi di un segnale proveniente dall'esterno. Una volta che tutte e due le azioni hanno portato il token al join, si va alla *Vai in stazione*.

TIMEOUT



EVENTO RIPETUTO

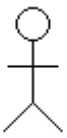


Se una clessidra ha un arco entrante, allora è un *timeout* che tiene fermo il token; se invece ha solo un arco uscente allora è un *evento ripetuto* che genera token.

Diagramma dei casi d'uso

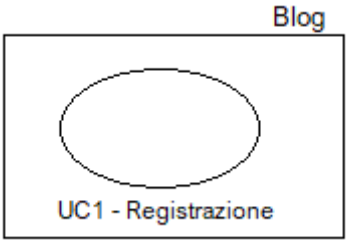
I diagrammi dei casi d'uso servono a far capire che cosa è richiesto dal cliente e lo traducono in un linguaggio comprensibile al programmatore. Servono in altre parole ad arrivare ai requisiti funzionali del software, cioè che cosa il programma fa. Qui non interessa del codice e l'implementazione tecnica; si tratta di identificare gli **scenari** per raggiungere un obiettivo e l'attore è quello che fa tali step. Le figure in gioco sono:

- **Attore**



La figura dell'omino indica un attore, cioè una persona esterna che interagisce con il mio sistema per fare azioni; può essere una persona o un sistema esterno. Quindi è tutto ciò di **esterno** che interagisce con l'**interno**. Un attore rappresenta una classe e quello che questa può fare.

- **Caso d'uso**

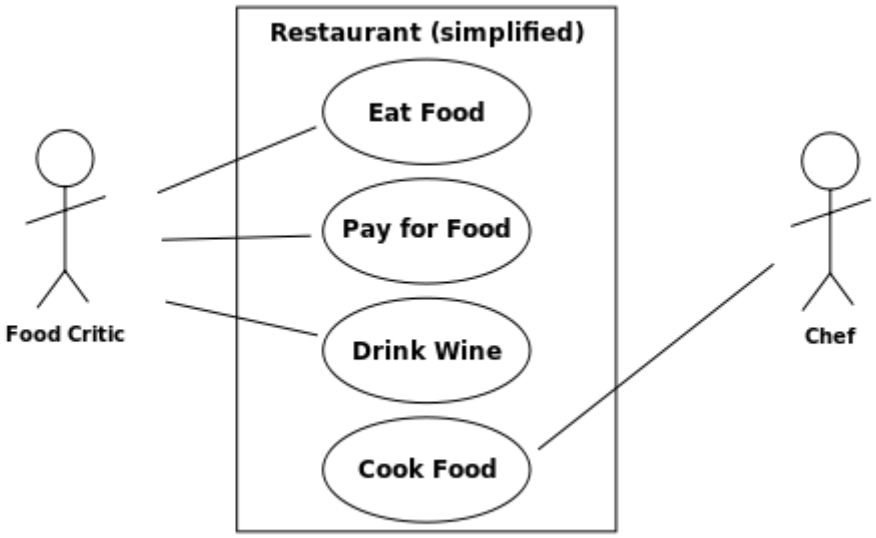


Si usa un rettangolo grane per indicare lo “scope” dei casi d’uso e ci si dà un nome; in questo caso è **blog** perché si vogliono rappresentare i casi d’uso per un blog. Il pallino indica l’azione da compiere e spesso viene identificato da dei codici univoci progressivi di tipo UC*x*. Questi casi d’uso rappresentano i metodi di una classe

- **Connessione**

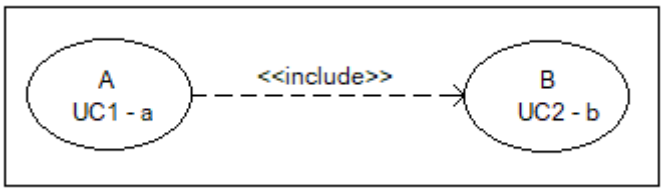
  Si fa con una linea dritta **senza** freccia alla fine ed indica la connessione tra l’attore e l’azione oppure le relazioni fra azioni.

Vediamo ora un esempio che riporta un diagramma per un ristorante con un critico di cibo (l’attore, cioè una classe) e dall’altra parte lo chef. Ogni UC è un’azione che questi possono fare cioè un “metodo della classe”.



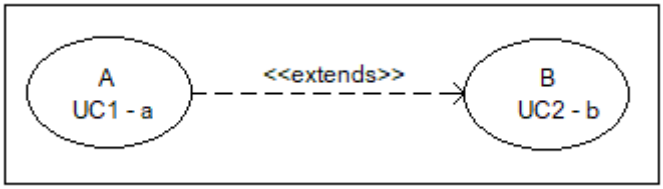
Fra casi d’uso ci possono essere delle connessioni, cioè avere delle sfere dentro al rettangolo attaccate da una linea dritta **con freccia** in questo caso. Queste connessioni indicano una relazione che può essere di questi tipi:

- **Inclusione**



Significa “eseguo **tutto** A e **tutto** B”. Si usa questa relazione quando un attore deve eseguire A e poi, una volta terminato, esegue incondizionatamente (cioè sempre) anche B. Vengono eseguiti prima A e poi B in sequenza sempre.

- **Estensione**



Significa “eseguo **un pezzo** di A e **tutto B”**. Si usa questa relazione quando un attore deve eseguire A e poi (mentre sta eseguendo A) si verifica una condizione che mi porta fuori da A andando in B, che viene eseguito del tutto. Importante capire che A **NON** viene **MAI** terminato.

L’inclusione la uso per esempio quando in un blog creo un nuovo post e verifico che abbia almeno 5000 caratteri; dopo aver terminato la fase di creazione del post (A) eseguo sempre anche (B) perché devo verificare che sia valido il testo.

L’estensione la uso per esempio quando ho da fare il login di un utente ma questo fallisce a causa di credenziali errate; mentre sto facendo la fase di login (A) ad un certo punto mi accorgo che la condizione di username e password esatti non è verificata e quindi vado in una pagina di errore (B). Eseguo una parte di A e poi tutto B.

Un attore esegue sempre tutte le inclusioni ma non esegue sempre tutte le estensioni; l’inclusione dice una funzionalità fissa che si ripete ogni volta, l’estensione dice una variazione della funzionalità standard.

Generalizzazione

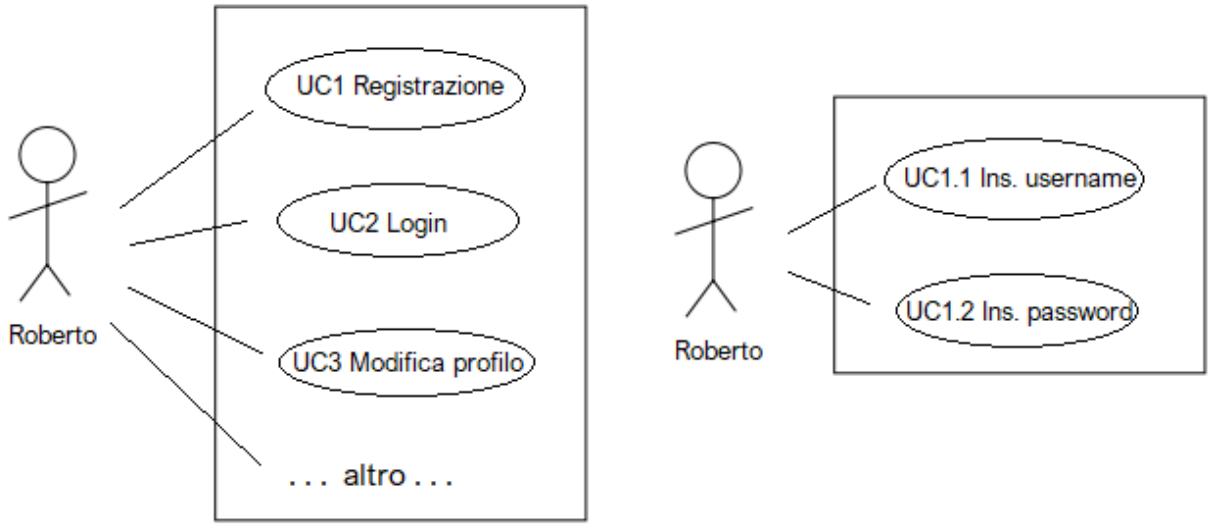
Questa roba riguarda anche gli attori. L’omino che sta sotto accede a **tutti** i casi d’uso dell’omino che sta sopra; si tratta di ereditarietà fra attori dove quello sotto eredita da quello sopra.

QUESTA COSA **NON** INDICA ANCHE L’EREDITARIETA’ FRA CLASSI

In questo caso B ha tutte le funzionalità di A più altre che lui può avere. Le stesse regole di generalizzazione valgono anche per gli UC e si rappresentano come con gli attori, cioè una freccia che va dal cerchio figlio a quello padre.

È importante si cercare di fare un diagramma delle classi che sia completo ma questo non vuol dire doverlo fare “troppo completo”. I diagrammi UC descrivono a far capire cosa deve fare il programma ma senza specificare il codice; andare troppo nello specifico nel disegno potrebbe portare a indicare anche il modo di scrivere il software e non va bene. È giusto dunque avere si un buon grado di dettaglio ma non troppo elevato.

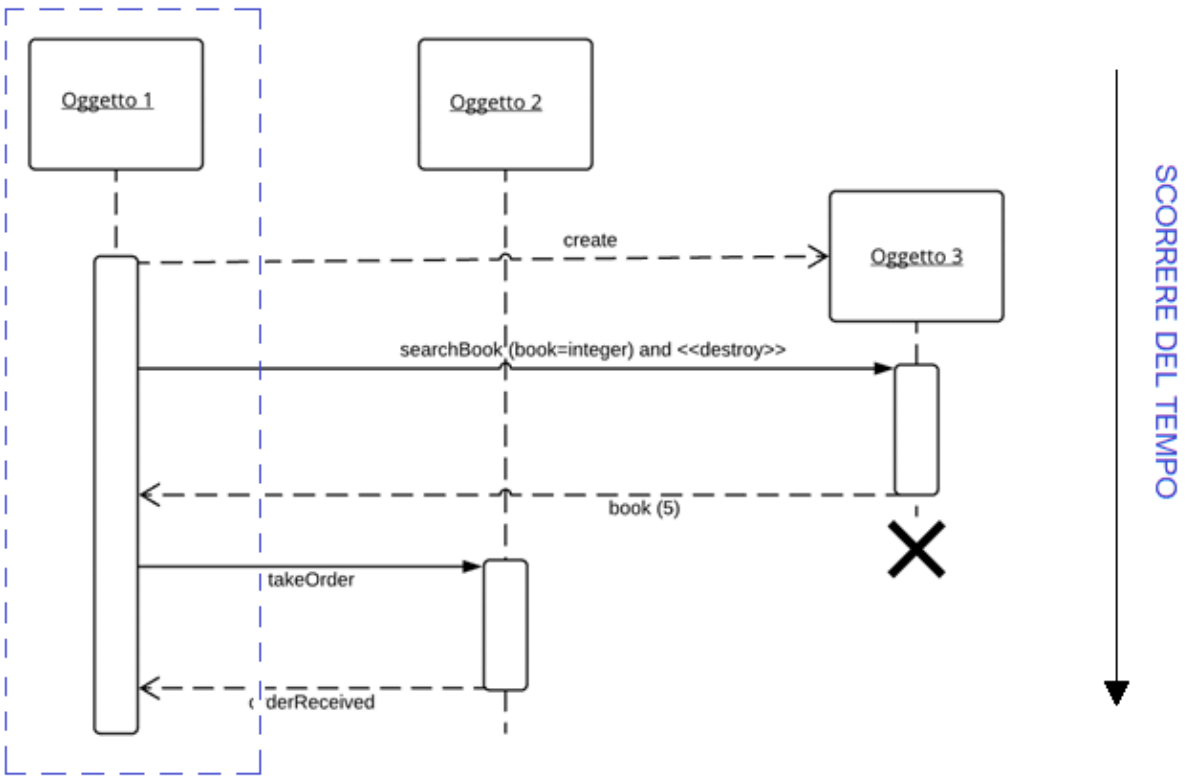
Da ricordare anche che si possono fare dei **sottocasi d’uso** che sono dei rettangoli che contengono delle specifiche di alcune azioni che possono essere fatte da un UC. Esempio:



In questo esempio si vede che c'è il caso d'uso registrazione e poi ci sono i due sottocasi d'uso di *Registrazione* mostrati sulla destra. Ovviamente dipende dal testo e da cosa bisogna fare ma se è richiesto, si può fare anche così. Attenzione perché *include* ed *esclude* sono "pericolosi" perché traggono in inganno; dipende sì dal testo ma solitamente è probabile che la soluzione più giusta sia fare un altro UC oppure fare dei sottocasi.

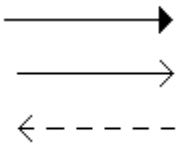
Diagramma di sequenza

Descrivono la collaborazione di un gruppo di oggetti che devono implementare collettivamente un comportamento. Tradotto, servono a mostrare le chiamate a metodi e le interazioni fra oggetti che avvengono durante il periodo di esecuzione del software.



Questi diagrammi vanno letti dall'alto verso il basso. Nei quadratini ci sono i nomi delle classi che sono coinvolte all'interno del programma o dello scenario che ci interessa; la barra bianca che sta sotto indica il tempo di vita dell'oggetto di quella classe. Quindi il rettangolo col nome è l'oggetto, la sbarra grossa sotto è il *object lifetime* cioè quanto vive quell'oggetto.

Quello tratteggiato in blu è il **partecipante** cioè l'oggetto che detiene il flusso del caso d'uso e ha le references agli altri oggetti. Detto semplice, il partecipante sarebbe l'oggetto principale "creato nel main" e poi lui fa partire il flusso del programma, cioè crea oggetti e chiama metodi. Non è proprio così perché non necessariamente sta nel main il partecipante, però è per avere un'idea. Le frecce indicano l'invocazione di metodi e possono essere:

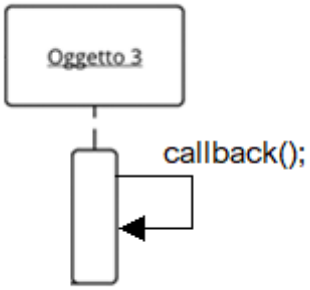


Messaggio sincrono

Messaggio asincrono

Messaggio di ritorno

Sopra alla freccia di solito si scrive il nome del metodo chiamato (con i parametri) oppure si aggiungono *<<create>>* o *<<destroy>>* per indicare la creazione o distruzione di oggetti. La croce (✕) alla fine indica la distruzione dell'oggetto durante il *flow* del programma.



Scenari come questi indicano che *Oggetto 3* fa una chiamata interna al metodo `callback` (magari se `callback` è `private`). Da ricordare che nei messaggi sincroni il chiamante resta in attesa del completamento del metodo, mentre in quelli asincroni no.