Ingegneria del Software

***Disclaimer**: sono sicuramente presenti errori. In caso di dubbi o incongruenze approfondire autonomamente. Appunti di Simone Brancato, lezioni e slide dei prof. Tramontana e Calvagna.*

L'ingegneria del software e Java

I metodi non devono fare troppe cose. Per ogni metodo devo implementare solo un piccolo compito altrimenti renderemmo il codice un monolite difficilmente riutilizzabile. Inoltre, potrebbe implementare un antipattern detto **spaghetti code**. Dobbiamo inoltre minimizzare le linee di codice, quindi puntare al massimo al riutilizzo del codice già esistente e alla produzione di codice il più riutilizzabile possibile.

Spaghetti code è un antipattern che consiste in metodi lunghi che utilizzano solo variabili globali, il flusso di esecuzione viene determinato dall'implementazione e non dal chiamante. Vi sono poche interazioni tra oggetti e la OOP si traduce in procedurale poichè non viene sfruttata. In generale possiamo dire che il naming dei metodi deve sempre essere parlante e deve spiegare cosa effettivamente fa quella funzione.

Come capire se si sta usando bene il **paradigma OOP**?

- Ogni metodo ha una **singola responsabilità**
- Posso **riutilizzare** i metodi senza troppi problemi
- I metodi si dividono in due categorie:
    - **Command:** cambiano lo stato del sistema non restituendo valori
    - **Query:** restituiscono lo stato del sistema ma non ne modificano lo stato.

  Questo ci permette di mantenere anche consistenza dei dati, separando i momenti di lettura e scrittura dei dati. Da notare il fatto che i metodi Query rendono l'oggetto osservabili, i Command richiedono più potenza computazionale.

L'estensione di funzionalità mediante sottoclassi si chiama **riuso a white-box** e richiede profonda comprensione della superclasse. Quindi è meglio limitare i livelli di una gerarchia di classi usando la composizione anzichè l'ereditarietà facendo quindi un **riuso black-box**.

Testing del codice

Più test facciamo e più siamo sicuri che il codice è corretto. I test si compongono di **valori in input**, **valore atteso** per quei valori in input e **risultato effettivo** a run-time. Per ogni metodo di classe esisterà almeno un metodo di test.

Ogni test è in un **ambiente totalmente controllato**, altrimenti i test non sono ben scritti e perdono di validità. Ogni test ovviamente deve autovalutarsi, e deve dare responso sia quando i test riescono sia quando falliscono, altrimenti avremo il dubbio se e quando sono stati eseguiti i test. Nel caso di fallimento dobbiamo includere cosa è andato storto.

Nel seguente esempio vediamo presente anche il main che chiama i test per eseguirli, in java abbiamo **jUnit** per omettere questa parte dato che li renderà eseguibili automaticamente.

```java
public class TestPagamenti {
    private Pagamenti pgm = new Pagamenti();

    private void initLista() {  //Inizializziamo i dati in input
        pgm.svuota();           //Svuotiamo la lista per rendere i test indipen.
        pgm.inserisci("321.01");
        pgm.inserisci("531.7");
        pgm.inserisci("1234.5");
    }

    public void testSommaValori() {
        initLista();
        pgm.calcolaSomma();
        if (pgm.getSomma() == 2087.21f)     //Testiamo il risultato effettivo
            System.out.println("OK test somma val");
        else System.out.println("FAILED test somma val");
    }

    public void testLeggiFile() {
        try {
            pgm.leggiFile("csvfiles", "Importi.csv");
            System.out.println("OK test leggi file");
        } catch (IOException e) {
            System.out.println("FAILED test leggi file");
        }
    }

    public static void main(String[] args) {
        TestPagamenti tl = new TestPagamenti();
        tl.testLeggiFile();
        tl.testSommaValori();
    }
}
```

Un metodo di sviluppo molto importante è il **Test Driven Development (TDD)** che consiste nello scrivere prima il test e poi il codice che risolve il requisito. Consiste nello scrivere codice solo se introducendo nuovi test vi sono errori, quindi *vanno lanciati anche se ci aspettiamo degli errori*. Scrivere prima i test ci consente di **riflettere su cosa vogliamo fare davvero**.

Esistono due tipi di test:

- **Test white-box**: chi scrive i test conosce gli algoritmi che sta testando, immaginando come si comporterà. In genere li scrive il programmatore che sta sviluppando il componente.
- **Test black-box**: chi scrive i test non conosce l'algoritmo, ha solo il documento dei requisiti potendo immaginare tante modalità di esecuzione e i casi particolari. Li scrive un programmatore diverso da chi ha scritto l'algoritmo, potendo quindi essere ancor più oggettivo e ragionare fuori dagli schemi di chi ha scritto il programma. Saper scrivere test black-box significa essere esperti sulla scrittura del codice, perchè immaginerà quali sono i punti critici del codice.

Refactoring del codice

Le **tecniche di refactoring** servono a passare da un codice che soddisfa i requisiti a una versione che continua a soddisfarle ma con una struttura migliore. Il refactoring consiste nella manutenzione del codice già esistente, adottando degli aggiustamenti nei casi in cui il codice evolva, deteriorandosi naturalmente. Il refactoring quindi non migliora il comportamento stesso del codice.

In generale il refactoring **riduce la dimensione del codice** e il **debito tecnico del progetto**, che consiste in un costo che un giorno dovrò pagare con gli interessi dato che **mi servirà più tempo fare le correzioni rispetto a programmare subito seguendo le best practices**. Inoltre, il refactoring aiuta nella pratica di ricerca di bug. Sarà necessario fare test costantemente, quindi prima scrivere i test, poi fare refactoring e verificare nuovamente i test così da scoprire subito se il refactoring ha alterato la correttezza del codice.

Esistono vari modi di fare refactoring:

1. **Estrai Metodo**: pratica dove si raggruppano frammenti di codice per creare un metodo il cui nome spiega il funzionamento di quel frammento. Per farlo si cerca laddove vi sono metodi lunghi o dove il codice risulta poco comprensibile. Sarà fondamentale trovare un naming adeguato per i metodi, la lettura del codice deve essere il più di alto livello possibile per essere più vicino al linguaggio naturale. Se il frammento ha variabili locali al metodo sorgente allora queste saranno dei parametri in ingresso, altrimenti se sono usate solo nel frammento allora possiamo usare variabili temporanee.

   ```
   public void stampaDettagli(){
       this.stampaNome();
       System.out.println("Prezzo: 30 euro"); // Hard coded, possibili
   ripetizioni
   }

   // Diventa...
   public void stampaDettagli(){
       this.stampaNome();
       this.stampaPrezzo();        // Con interfaccia riutilizzabile
   }
   ```

2. **Sostituisci Temp con Query:** immaginiamo un metodo al cui interno esiste una variabile temporanea. Potrebbe accadere di voler riutilizzare la stessa espressione e quindi di creare un'altra variabile temporanea. Possiamo evitare questo problema creando un metodo query, così da poterlo riutilizzare quante volte vogliamo. Questo metodo di refactoring risulta particolarmente agevole laddove vi sia una sola assegnazione alla variabile.

   ```
   double prezzo = quantita * prezzo;

   // Diventa una chiamata alla funzione ...
   private double getPrezzo(){
       return quantita * prezzo
   }
   ```

3. **Dividi variabile Temp:** se una variabile temporanea nel corso della sua vita viene riassegnata è meglio usare due variabili temporanee distinte perchè rende più parlante il naming delle variabili stesse. Occupa più memoria, ma è trascurabile rispetto ai vantaggi che otteniamo: tutta la OOP si basa su questo assunto.

```
double temp = 2*(height+width); // temp è il perimetro del rettangolo
temp = height*width;    // Adesso temp è l'area del rettangolo

// Per avere nomi più appropriati ed avere un codice più leggibile diventa
...
final double perimeter = 2*(height+width);
final double area = height*width;
```

Diagrammi UML

Per produrre dei diagrammi UML di qualità dobbiamo innanzitutto soffermarci su come individuare gli attori che entrano in giro e i loro ruoli. Attrraverso un'analisi grammaticale individuiamo:

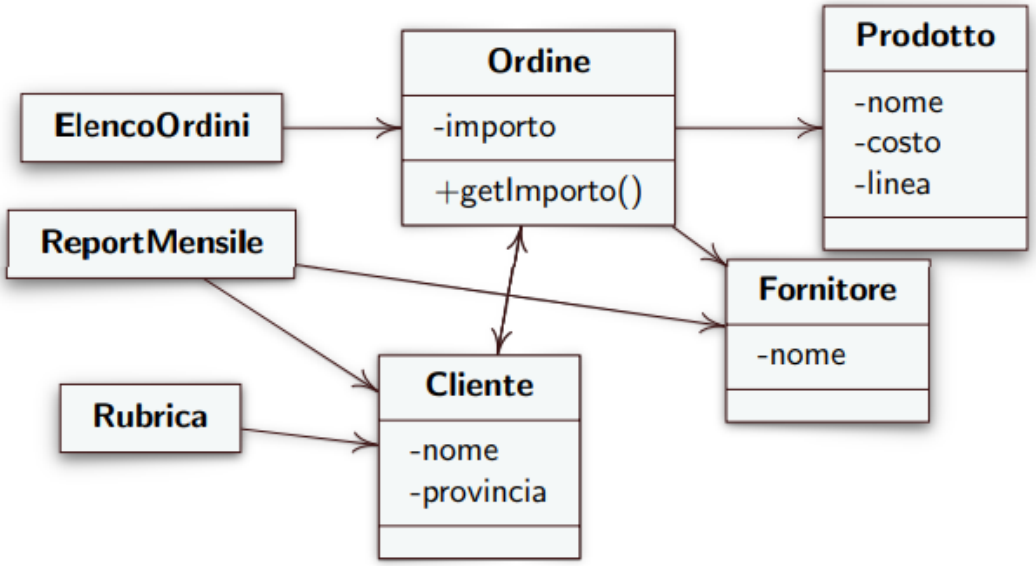
- **Sostantivi**: suggeriscono le classi e gli attributi
- **Verbi**: suggeriscono i metodi

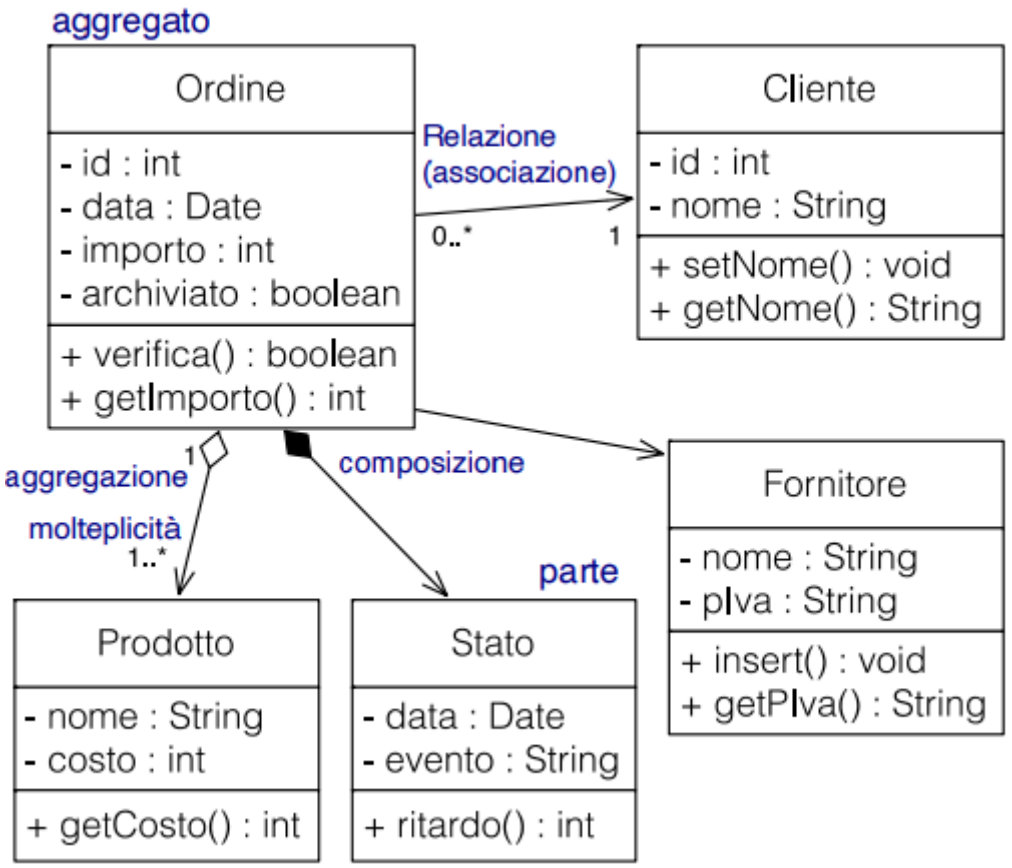
Per esempio, se abbiamo dei requisiti del tipo:

- ... dovrà essere possibile **cercare** un **cliente** e **mostrarne** i **dati anagrafici**
- ... la **scheda cliente mostra** i **dati** anagrafici ed **l'elenco** dei suoi **fornitori**
- ... su richiesta dell'utente si **calcola l'importo complessivo** degli **ordini** da lui **effettuati** in un intervallo di tempo
- ... ogni **ordine mostra nome fornitore, nome cliente, genere prodotti, importo complessivo**
- ... il **report mensile mostra** per ciascun **cliente provincia** di appartenenza e **totale** ordinato per ciascun **fornitore**

Individuiamo i seguenti ruoli:

- **Classi**: Cliente, Fornitore, Ordine, Prodotto, ReportMensile, SchedaCliente
- **Attributi**:
  - Cliente: dati anagrafici, nome, provincia
  - Prodotto: genere
  - Ordine: importo
  - Fornitore: nome
- **Metodi**:
  - Cercare un cliente
  - Mostrare dati anagrafici e fornitori per cliente
  - Calcolare totale ordini per un cliente
  - Selezionare ordini in un intervallo temporale
  - Calcolare totale ordini per un cliente per ciascun fornitore per mese





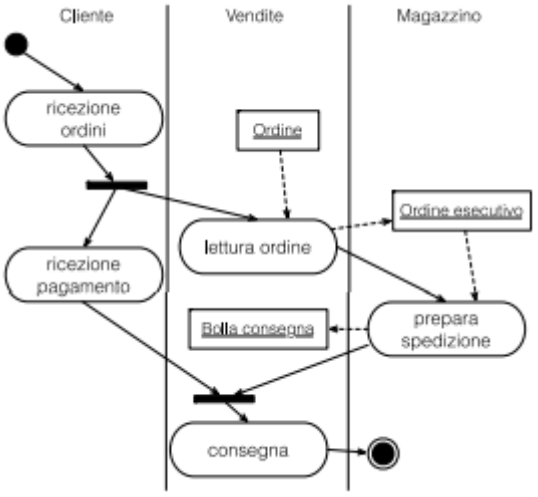
Il **diagramma di collaborazione** mostra le interazioni tra gli oggetti, che comunicano scambiandosi messaggi ovvero chiamate a funzioni. Il flusso viene indicato con frecce direzionate con etichetta:

1. Numero sequenziale di ordinamento
2. Nome metodo chiamato
3. Valori di ritorno

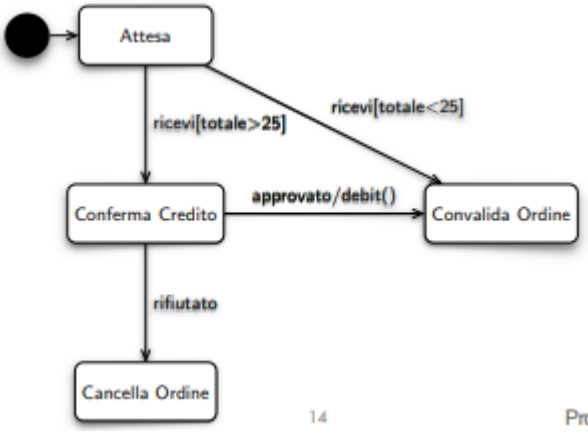
L'analisi grammaticale permette di individuare alcune classi ma non tutte. Il progettista potrebbe decidere di introdurne altre per migliorare l'architettura del software per esempio per sfruttare polimorfismo o design pattern.

I **diagrammi delle attività** sono il punto iniziale di una progettazione. Vengono usati per delineare le attività che il sistema software dovrà realizzare, non mostra quali oggetti svolgono le attività ma può essere una base per costruire il diagramma di collaborazione degli oggetti. Viene disegnato mediante rettangoli arrotondati, il cerchio pieno è lo stato iniziale. Qualora vi siano flussi paralleli magari decisi sulla base di condizioni si inserisce una barra trasfersale.

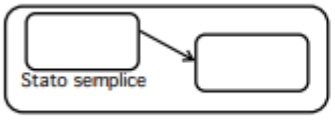
Eventuali corsie nei diagrammi sono partizioni per indicare il responsabile delle attività condotte nella specifica corsia. I rettangoli sono oggetti o dati, le frecce tratteggiate sono dati in input o output



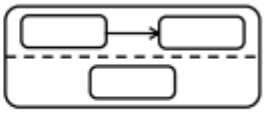
Tramite i diagrammi UML possiamo ovviamente descrivere anche gli stati degli oggetti, quindi il diagramma degli stati. Tale stato è caratterizzato da valori che causano la transizione verso quel particolare stato. Una transizione consiste nel passaggio da uno stato a un altro a seguito di un evento, necessita quindi di uno stato di arrivo a seguito di un'azione.



Uno stato può essere **composto**, ovvero che consiste in vari sottostati sequenziali o concorrenti. Se uno stato è sequenziale solo uno dei sequenziali può essere attivo in un certo momento. Stati esterni sono comunque condizione degli stati più interni (come if annidati, esempio alla lontana), quindi se è in uno stato interno siamo anche ovviamente nello stato esterno.

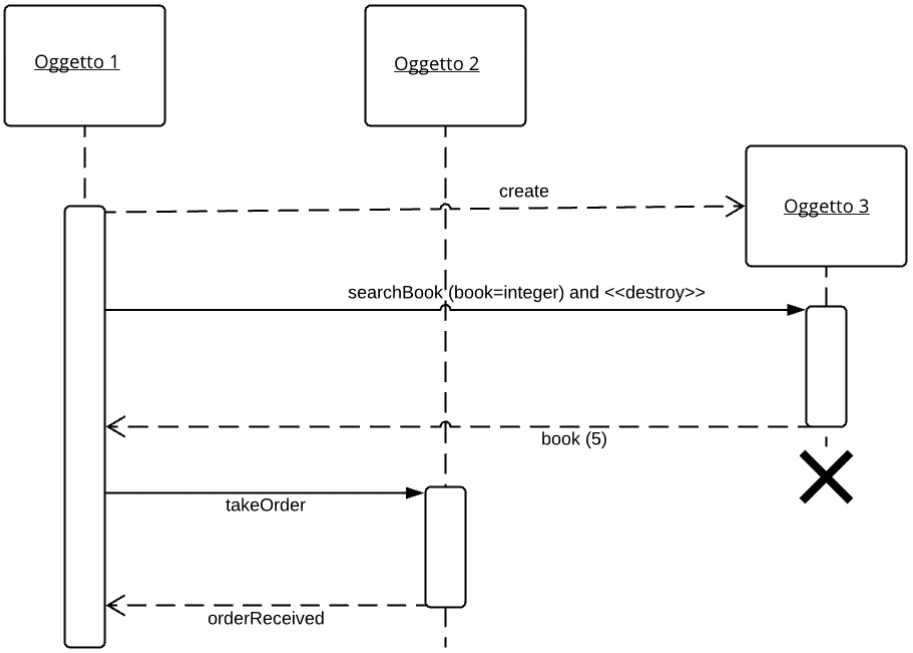


Stato composto sequenziale


Stato composto concorrente

Un **diagramma UML di sequenza** mostra le interazioni fra oggetti. L'asse temporale è inteso in verticale verso il basso, in orizzontale i nomi dei vari oggetti. In ciascuna colonna l'oggetto se esiste viene indicato con una linea tratteggiata, se è attivo con una barra di attivazione. La chiamata di metodo viene indicata da una freccia piena che va dalla barra di attivazione di un oggetto ad un

altro.


Oggetto 1
Oggetto 2
Oggetto 3
create
searchBook (book=integer) and <<destroy>>
book (5)
takeOrder
orderReceived

Design Pattern

Sono strutture software per un piccolo numero di classi che **descrivono soluzioni per problemi ricorrenti**, quindi una piccola parte di un sistema più grande. Specificano le classi ed oggetti coinvolti, comprese le loro interazioni. L'obiettivo è il riutilizzo di un insieme di classi notoriamente conosciute così da rendere anche più intuitivo a terzi il funzionamento del software che stiamo scrivendo e integrarle bene facendole comunicare correttamente.

Inoltre aiutano i principianti ad agire come esperti e supportano gli esperti nella progettazione di sistemi complessi, evitano di re-inventare concetti e soluzioni **riducendo i costi di sviluppo** e forniscono un **vocabolario comune** agli sviluppatori permettendo una migliore comprensione del codice in termini di funzionalità e caratteristiche non funzionali (affidabilità, modificabilià, testabilità).

Ogni design pattern specifica:

- **Nome**: permette di dare un'idea indicativa sullo scopo del pattern
- **Intento:** descrivere lo scopo del design pattern
- **Problema:** descrive il problema a cui è applicato il design pattern e le condizioni necessarie per applicarlo
- **Soluzione:** descrive le classi, le loro responsabilità e le loro relazioni
- **Conseguenze:** indicano i risultati, vantaggi e svantaggi nell'uso del design pattern

Nella descrizione del problema si ricorre all'uso del concetto di forze, ovvero inteso come obiettivi e vincoli (in genere questi due sono contrastanti).

Il libro Design Patterns di Gamma&Co divide i design patterns in categorie:

- **Creazionali:** creazione oggetti
- **Strutturali:** scelte struttura software
- **Comportamentali:** incapsulamento di algoritmi

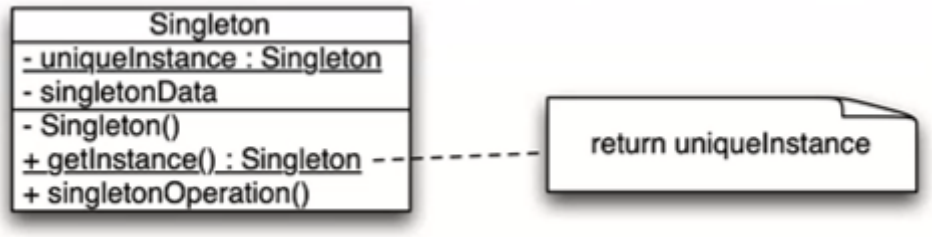
Pattern Creazionali

Singleton

Intento: assicurare che una classe abbia una sola istanza e fornire un punto d'accesso globale all'istanza

Motivazione: alcune classi devono avere una sola istanza in tutta l'applicazione, variabili globali sono visibili ovunque ma non limitano la creazione di ulteriori istanze e la classe deve occuparsi del punto d'accesso.

Soluzione: la classe Singleton ha un metodo statico `getInstance()`, che viene invocato sulla classe. Ha come valore di ritorno il riferimento a quell'istanza della classe. Ne segue che il costruttore è privato, per non permettere la creazione di istanze ad altre classi.



Esempio:

```
public class Fibonacci {
    private static Fibonacci obj = new Fibonacci();
    private int[] x = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
    private Fibonacci(){}

    public static Fibonacci getInstance(){
        if(obj == null) obj = new Fibonacci();
        return obj;
    }
}
```

Esiste una variante del Singleton detto **Multiton** che permette un numero finito di istanze. Potremmo adattarlo in questo modo:

```
public class Fibonacci {
    private static Fibonacci obj = new Fibonacci();
    private static instanceCount = 0;
    private int[] x = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
    private Fibonacci(){}

    public static Fibonacci getInstance(){
        if(instanceCount <= 5){
            obj = new Fibonacci();
            instanceCount++;
        }
        return obj;
    }
}
```

Nella scelta della progettazione facendo delle modifiche al Singleton per arrivare a più istanze abbiamo dovuto fare poche modifiche anzichè tutti i punti chiamanti del `getInstance()`, questo si chiama **principio delle conseguenze locali**, ovvero un cambiamento piccolo nel codice non deve causare altri problemi nel resto del sistema software.

Factory Method

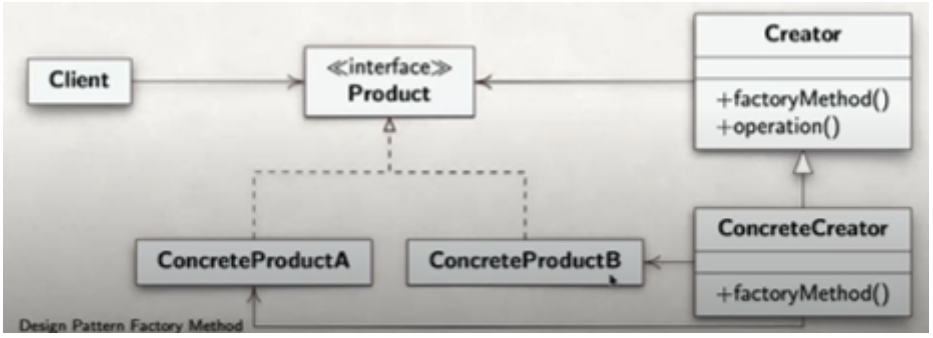
Intento: definire un'interfaccia per creare un oggetto ma lasciare che le sottoclassi decidano quale classe istanziare.

Problema: supponiamo di star costruendo un framework utile a costruire oggetti ma che ancora non sono conosciuti, e che dovranno essere completati da altri componenti. Il framework deve istanziare oggetti conoscendo solo le classi astratte che non può usare, e lo farà tramite il metodo `factoryMethod()` dato che incapsula la conoscenza su di quale istanza va creata.

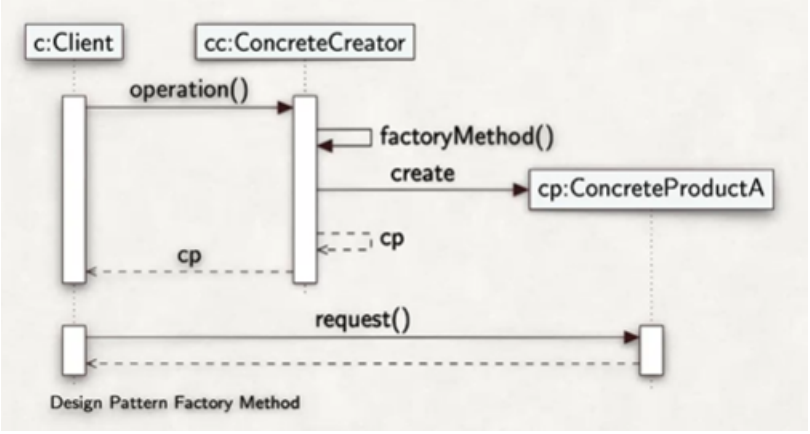
Soluzione

- **Product** è l'interfaccia comune degli oggetti creati da `factoryMethod()`, i client si interfacciano qui.
- **ConcreteProduct** è un'implementazione di Product. Non aggiungono metodi ereditati da Product altrimenti il design pattern non è utilizzabile. Se nel sistema esiste un solo ConcreteProduct il design pattern perde di utilità.
- **Creator** dichiara `factoryMethod()`, che ritorna oggetti di tipo Product e ha una sottoclasse ConcreteCreator. Creator conosce Product, non le concrete, al massimo può avere un tipo concreto di ritorno di default per `factoryMethod()`.
- **ConcreteCreator** implementa `factoryMethod()`, o ne fa l'override, sceglie quale ConcreteProduct istanziare e lo restituisce.

Implementiamolo solo se abbiamo più Product o se da qui a pochissimo avremo più di un ConcreteProduct, altrimenti stiamo facendo overengineering e va evitato.



Vediamo quindi il diagramma UML di sequenza per mostrare le interazioni tra i vari ruoli:



Vogliamo sempre legarci alle interfacce e non alle classi. Il client non deve conoscere i ConcreteProduct. Il client vuole farsi dare un'istanza chiedendola a ConcreteCreator, che gestirà `operation()` lanciando `factorymethod()`, che crea l'istanza di una ConcreteProduct che viene ritornata dal metodo factorymethod e infine ritornata da `operation()`.

```
public interface IVehicle {
    public void aggiungiPasseggero(String nome);
    public void accelera();
    public void stampaDettagli();
}
```

```
public class VehicleFactory {
    public static IVehicle factoryVehicle(String type){
        switch (type) {
            case "Auto": return new Auto();
            case "Moto": return new Moto();
            case "Autobus": return new Autobus();
        }
        return null;
    }
}
```

```
public class Client {
    public static void main(String[] args){
        IVehicle firstVehicle = VehicleFactory.factoryVehicle("Auto");
        firstVehicle.aggiungiPasseggero("Simone");
        firstVehicle.accelera();
        firstVehicle.stampaDettagli();
    }
}
```

```
public class Auto implements IVehicle{
    private LinkedList<String> listaPasseggeri = new LinkedList<>();
    int numeroPasseggeri = 5;

    @Override
    public void aggiungiPasseggero(String nome) { ... }

    @Override
    public void accelera() { ... }

    @Override
    public void stampaDettagli() { ... }
}
```

Esistono delle varianti del factory design pattern. Questi sono:

- **Creator e ConcreteCreator sono svolti dalla stessa classe** come l'esempio precedente
- **Il metodo factory potrebbe essere static**, è consentito. Se non è static prima dobbiamo procurarci un'istanza del ConcreteCreator
- **Il metodo factory potrebbe prendere in ingresso dei parametri**. Questo permette al client di suggerire la classe da usare per creare l'istanza
- Il metodo factory potrebbe utilizzare la **Riflessione Computazionale**, ovvero dei meccanismi che permettono di scrivere codice che ha al suo interno ragionamenti sul codice stesso. All'interno di questa soluzione creo una rappresentazione della classe, permettendo di parametrizzare il tipo.

Potrebbe capitare che quando istanzio una classe ConcreteProduct questa potrebbe voler lavorare con istanze di altre classi, quindi questa ha delle dipendenze esterne. Un buon modo per mettergliele a disposizione è passarle come parametri al costruttore del ConcreteProduct, questo metodo si chiama **Dependency Injection**. Tramite il pattern Dependency Injection **una classe**

riceve nel suo costruttore tramite parametri altri oggetti da cui dipende, che sono detti dipendenze. Questo permette di separare la costruzione delle dipendenze dal loro utilizzo effettivo, quindi non è il client a creare le istanze da cui dipende. Questo permette di **evitare complicazioni** e semplifica il codice (infatti altrimenti nel costruttore dovremmo validare e settare le dipendenze).

```
public class Studente implements IStudente {
    private PianoStudi piano;
    private Writer wrt;

    public Studente(PianoStudi s, Writer w){  // DEPENDENCY INJECTION
        piano = p;
        wrt = w;
    }
}
```

Object Pool

Questa è una tecnica di programmazione facilmente applicabile con il Factory Method. Torna utile quando vogliamo implementare una sorta di **deposito per delle istanze**, immaginiamo casi in cui in un sistema software creiamo degli oggetti che vogliamo usare più volte. Queste istanze possono essere conservate in un cosiddetto object pool, e su richiesta vengono fornite ai client, che quando finiranno di usarla restituiranno l'oggetto al repository.

Ciò torna utile perchè spesso abbiamo a che fare con **oggetti molto pesanti**, il che rende pesante la loro istanziazione. Implementare un pool così da renderle riutilizzabili risulta particolarmente vantaggioso computazionalmente parlando. In queste situazioni si implementa una sorta di **uso esclusivo** dell'oggetto.

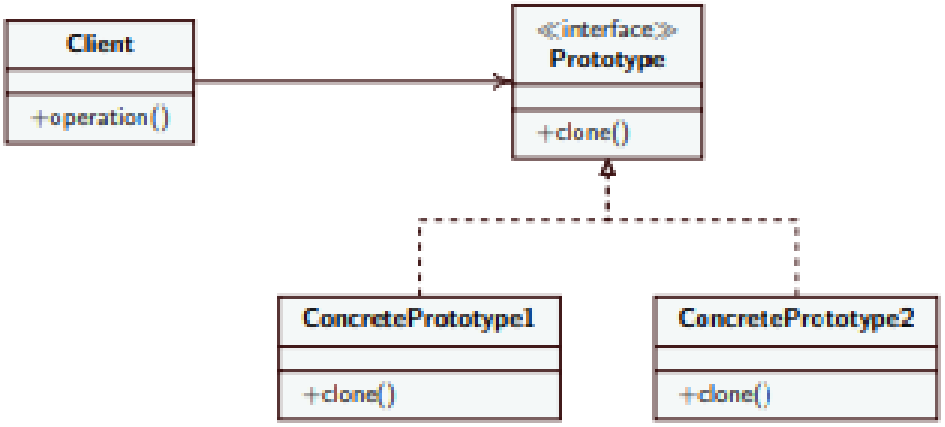
L'object pool potrebbe essere a dimensione fissa o variabile, ciò varia il comportamento nella situazione in cui finiscono le istanze nel repository. Basta implementare una lista parametrizzata, un buon posto per implementarlo è nel ConcreteCreator nel caso della Factory ed eventualmente inserire valori massimi come costanti ed attributi della classe che implementa la pool.

```
public class CreatorPool extends ShapeCreator{
    private LinkedList<Shape> pool = new LinkedList<>();

    public Shape getShape(){
        Shape s;
        if(pool.size()>0) s = pool.remove();
        else s = new Circle();
        return s;
    }

    public void releaseShape(Shape s){
        pool.add(s);
    }
}
```

Prototype

Intento: specificare i tipi degli oggetti da creare usando un oggetto prototipo.

Motivazione: creando degli oggetti potrebbe essere necessario avere uno stato iniziale simile a quello di un oggetto già istanziato. Solitamente quando si crea un oggetto se ne specifica la classe, quindi tale oggetto sarà strettamente accoppiato con la classe che sta istanziando. Ciò rende difficoltoso l'aggiunta di nuovi tipi.



- **Prototype** definisce l'interfaccia per le operazioni comuni ai vari oggetti e dichiara un metodo per clonare sè stessa.
- **ConcretePrototype** implementa l'operazione di clonazione
- **Client** è una classe che crea nuovi oggetti tramite la clonazione del prototipo senza indicare esplicitamente la concreta

Conseguenze: il client non conosce la classe usata poichè conosce solo l'interfaccia comune e questo favorisce il disaccoppiamento tra classi. Si possono registrare nuovi prototipi a runtime. Non servono gerarchie perchè ogni oggetto si ottiene mediante clonazione.

Implementazione

Possiamo usare un **manager di prototipi** come registro delle istanze per permettere ai client di recuperare i prototipi per poi clonarli. Distinguiamo due livelli di clonazioni:

- **Copia superficiale**: vengono tenuti i riferimenti degli attributi dell'oggetto da clonare, quindi clone e prototipo sarebbero strettamente legati. Spesso questo non basta.
- **Copia profonda**: clone e prototipo sono totalmente separati in termini di attributi, i cloni potrebbero essere istanziati dalla classe prototipo impostandone i dati senza farne mantenere un riferimento.

Esempio: in un'agenda possono essere inserite vari voci. Lo stato dell'agenda corrisponde all'insieme degli impegni. Una nuova istanza dell'agenda deve poter accedere agli impegni già inseriti.

```
public interface Prototype {
    Prototype clone();
    void stampa();
}
```

```
public class CompactModule implements Prototype{
    private final int id;
```

```
    private final int mode;
    private final String name;

    public CompactModule(int id, int mode, String name){
        this.id=id;
        this.mode=mode;
        this.name = name;
    }

    public Prototype clone() { return new CompactModule(id, mode, name);

    public void stampa(){ ... }
}
```

```
public class Client {
    public static void main(String[] args){
        Prototype firstProto = new CompactModule(8, 23, "CompactModule");
        firstProto.stampa();
        Prototype nuovaCopia = firstProto.clone();
        nuovaCopia.stampa();
    }
}
```

Pattern Strutturali

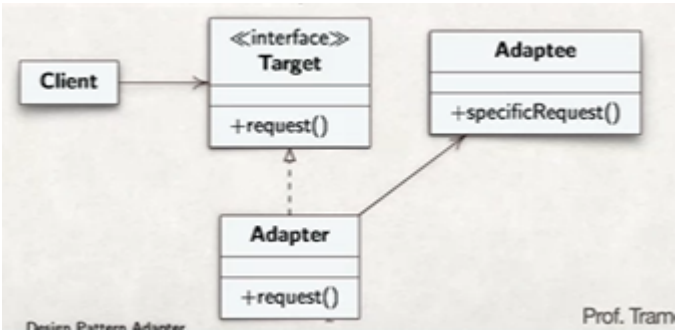
Adapter

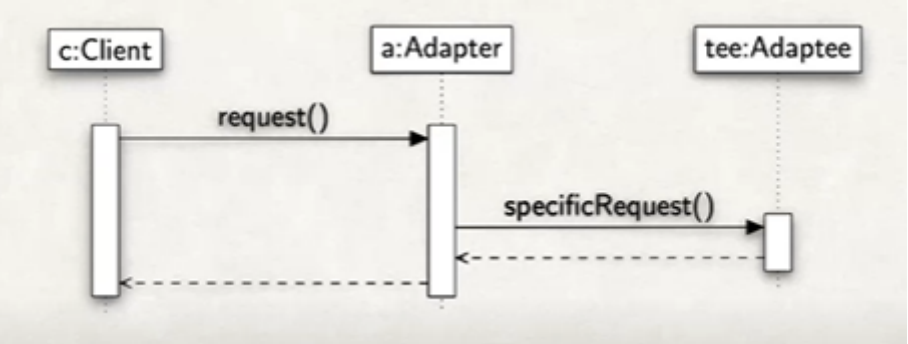
Intento: convertire l'interfaccia di una classe in una che i client si aspettano. Permette a delle classi di comunicare eliminando il problema delle interfacce incompatibili.

Problema: si vogliono usare componenti software incompatibili tra loro, e risulta sconveniente o non possibile modificare una o l'altra interfaccia.

Soluzione Object Adapter:

- **Target** è l'interfaccia che il client si aspetta
- **Client** usa oggetti conformi all'interfaccia Target
- **Adaptee** è l'oggetto che necessita l'adattamento
- **Adapter** converte la chiamata del Client all'interfaccia della classe Adaptee. Client utilizza Adapter come se fosse Adaptee in quanto ne tiene un riferimento.





Vediamo un esempio:

```
public interface ILabel {
    public String getNextLabel();
}
```

```
// ADAPTER
public class Label implements ILabel {
    private LabelServer ls;
    private String p;

    public Label(String prefix){
        p = prefix;
    }

    public String getNextLabel(){
        if(ls == null) ls = new LabelServer(p);
        return ls.nextLabel();
    }
}
```

```
// ADAPTEE
public class LabelServer {
    private int labelNum = 1;
    private String labelPrefix;

    public LabelServer(String prefix){
        labelPrefix = prefix;
    }

    public String nextLabel(){
        return labelPrefix+labelNum++;
    }
}
```

```
public class Client {
    public static void main(String args[]){
        ILabel s = new Label("Lab");
        String l = s.getNextLabel();    // l = Lab1
    }
}
```

Il metodo da adattare può variare nel nome, valori di ritorno, parametri d'ingresso

Soluzione Class Adapter: la classe Adapter è sottoclasse dell'Adaptee, evitando la fase dove dobbiamo creare dei riferimenti all'Adaptee. Qui si sfrutta l'ereditarietà di Java perchè estendiamo da una classe e implementiamo l'interfaccia usata dai client. La classe Adapter fornisce sia l'interfaccia di Target che quella di Adaptee, è quindi un **adapter a due vie**.

Conclusioni

- Client e Adaptee sono **indipendenti**, l'Adapter può anche modificare il comportamento di Adaptee.
- Possiamo inserire **precondizioni** (predicati che devono essere soddisfatti prima di inoltrare la chiamata a funzione) e **postcondizioni** (predicati che devono essere soddisfatti se tutto è andato a buon fine).
- Adapter risulta in un **layer di indirettezza**, e dato che ad ogni chiamata del client corrisponde una di Adapter il codice potrebbe risultare più difficile da comprendere.

Facade

Intento: fornire un'interfaccia unificata al posto di un insieme di interfacce per un sottosistema. Definire un'interfaccia di altissimo livello che renda il sottosistema più facile da usare. I client avranno meno metodi a disposizione.

Problema: la presenza di tante classi rende i sistemi molto complessi e può diventare difficile per i client capire quale interfaccia usare. Inoltre vogliamo ridurre al minimo il numero di possibili comunicazioni tra client e il sottosistema, quindi ridurre il numero di metodi.

Soluzione: fornisce un'interfaccia semplificata ai client per nascondere gli oggetti del sottosistema, ridurre la complessità dell'interfaccia, sarà il Facade a invocare i giusti metodi del sottosistema. I client interagiscono solo con l'oggetto Facade.

Conseguenze: l'implementazione del sottosistema è nascosta ai client e permette un accoppiamento debole tra sistema e client. Nel caso di sistemi grandi migliora i tempi di compilazione nel caso di modifiche alle classi perchè basta ricompilare fino al facade.

Implementazione: possiamo annidare le classi del sottosistema all'interno della classe Facade per forzare l'uso della classe solo su di esso. Altrimenti possiamo partizionare il progetto, ma il sottosistema potrebbe essere comunque visibile ai client. Nel caso in cui ci sia bisogno di permettere a client complessi di accedere comunque al sottosistema la soluzione del partizionamento del progetto è da preferire.

Esempio di codice:

```
public class Client {
    public static void main(String[] args) {
        Calcolatore calcolatore = new Calcolatore();
        calcolatore.aggiungiRAM();
        calcolatore.aggiungiSSD();

        calcolatore.stampa();
        calcolatore.getCosto();
    }
}
```

```java
public record Memoria(int dimensione, int costo, String tipo) { }
```

```java
public record CPU(int frequenza, int core, int costo) { }
```

```java
public record SchedaVideo(int velocita, int costo) { }
```

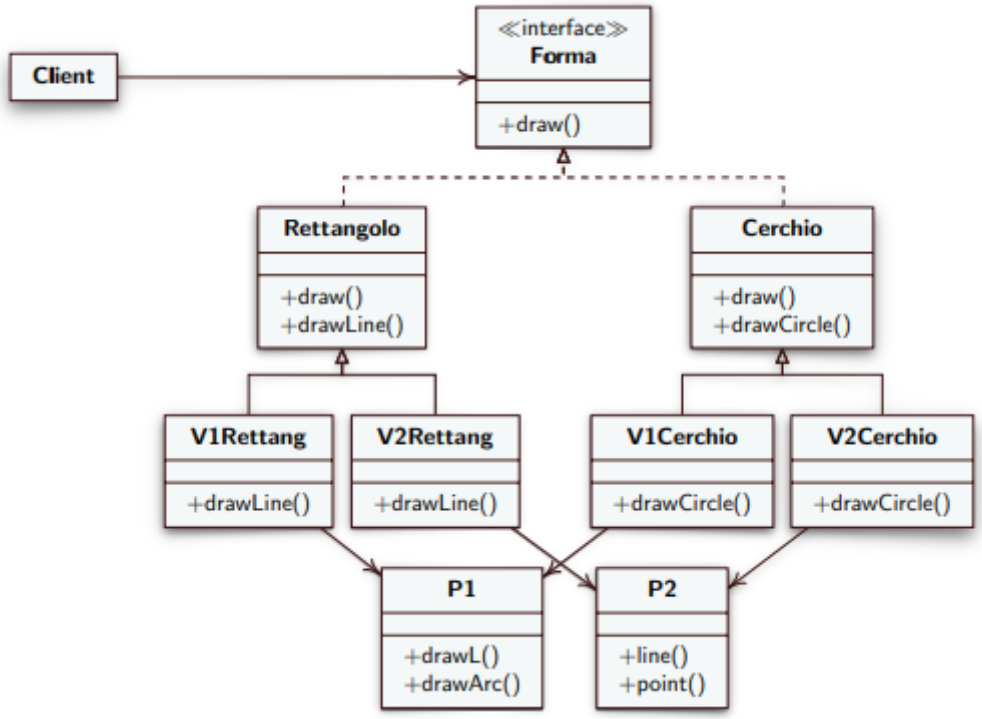
```java
// FACADE
public class Calcolatore {
    private LinkedList<Memoria> listaMemo = new LinkedList<>();
    private SchedaVideo schedaVideo = new SchedaVideo(100,50);
    private CPU cpu = new CPU(100,8,120);

    public void aggiungiRAM(){
        Memoria memo = new Memoria(16, 50, "RAM");
        listaMemo.add(memo);
    }

    public void aggiungiSSD(){
        Memoria memo = new Memoria(512, 50, "SSD");
        listaMemo.add(memo);
    }

    public int getCosto(){
        int costo = 0;
        for(Memoria memo : listaMemo){
            costo += memo.costo();
        }
        costo += schedaVideo.costo() + cpu.costo();
        return costo;
    }

    public void stampa(){
        System.out.println("Dettagli calcolatore:");
        System.out.println(this.cpu);
        System.out.println(this.schedaVideo);
        System.out.println("Memorie:");
        System.out.println(this.listaMemo);

    }
}
```

Bridge

Intento: vogliamo disaccoppiare un'astrazione dalla sua implementazione così da poterle alterare indipendentemente.

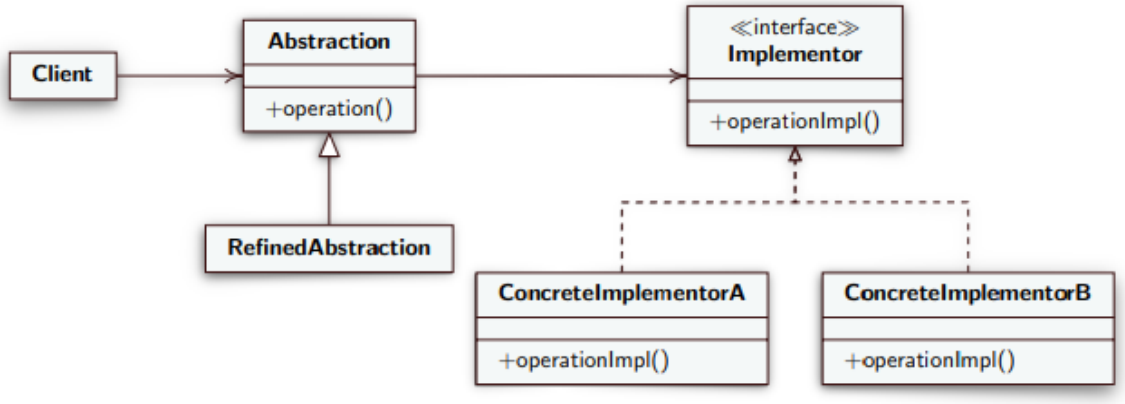
Motivazione: solitamente usiamo l'ereditarietà per dare ad un'astrazione più implementazioni. Ciò rende la classe astratta un'interfaccia e le sottoclassi un'implementazione dell'interfaccia. Ciò però rende difficile modificare, estendere e usare altre implementazioni indipendentemente.

Per esempio, immaginiamo di avere un sistema con astrazioni Rettangolo e Cerchio. Tali astrazioni devono essere implementate in vario modo, per esempio qui abbiamo V1 e V2, potrebbe rifletterne il colore per esempio.



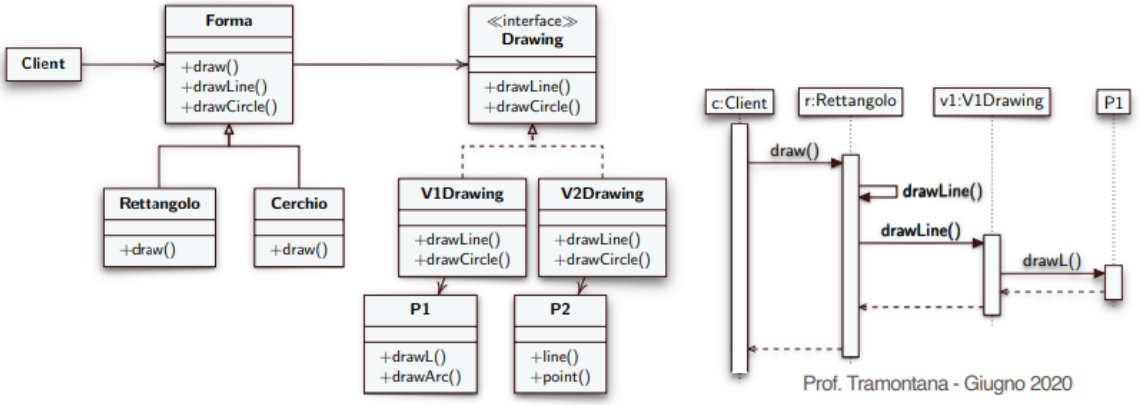
Tale soluzione non è ottimale poichè inserendo nuove astrazioni e implementazioni avremmo un aumento lineare delle classi, inoltre le piattaforme si legano in maniera indissolubile alle implementazioni. Per esempio aggiungendo solo una nuova astrazione, magari Triangolo, dovremo creare anche V1Triangolo, V2Triangolo e non va bene.

Soluzione:



- **Abstraction** definisce l'interfaccia per i client mantenendo un riferimento ad un oggetto di tipo Implementor. Inoltra le richieste del client a questo oggetto Implementor (Es. Forma)
- **RedefinedAbstraction** estende l'interfaccia Abstraction (Es. Rettangolo/Cerchio)
- **Implementor** definisce l'interfaccia per le varie implementazioni. In genere Abstraction definisce operazioni di alto livello, Implementor quelle più di basso livello che rimanderanno alle implementazioni.
- **ConcreteImplementor** implementano l'interfaccia di Implementor fornendo operazioni specializzate

Nell'esempio mostrato avremo un diagramma di questo genere:



Nel codice avremo `Rettangolo.draw()` che chiama `drawLine()` di Forma, che a sua volta chiama `drawLine()` su un'istanza di Drawing. Inserire nuove classi, come Rombo, necessiterà solo di un'implementazione come sottoclasse di Forma

Conseguenze: Bridge permette alle implementazioni di essere configurate a runtime e senza dover ricompilare Abstraction e Client, che non devono conoscere Implementor. Bridge divide e organizza una singola classe che ha multiple varianti di alcune funzionalità in due gerarchie, l'astrazione e l'implementazione. Il client non dovrà conoscere necessariamente le varie implementazioni e viene rispettato il single responsibility principle.

```
public class Client {
    public static void main(String[] args) {
        Forma rettangoloBlu = new Rettangolo("Blu");
        rettangoloBlu.disegna();
        Forma cerchioRosso = new Cerchio("Rosso");
        cerchioRosso.disegna();
    }
}
```

```
public abstract class Forma {
    public abstract void disegna();
    protected Disegno disegno;
}
```

```
public class Rettangolo extends Forma {

    public Rettangolo(String colore){
        if(colore.equals("Rosso"))
            this.disegno = new DisegnoRosso();
        else if(colore.equals("Blu"))
            this.disegno = new DisegnoBlu();
    }

    public void disegna() {
        disegno.disegnaRettangolo();
    }
}
```

```
public class Cerchio extends Forma {

    public Cerchio(String colore){
        if(colore.equals("Rosso"))
            this.disegno = new DisegnoRosso();
        else if(colore.equals("Blu"))
            this.disegno = new DisegnoBlu();
    }

    public void disegna() {
        disegno.disegnaCerchio();
    }
}
```

```
public interface Disegno {
    void disegnaRettangolo();
    void disegnaCerchio();
}
```

```
public class DisegnoBlu implements Disegno {

    public void disegnaRettangolo() {
        System.out.println("Rettangolo blu.");
    }

    public void disegnaCerchio() {
        System.out.println("Cerchio blu.");
    }
}
```

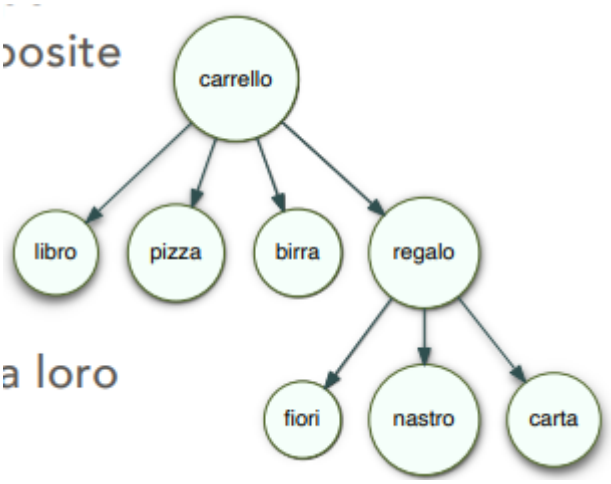
```
public class DisegnoRosso implements Disegno {

    public void disegnaRettangolo() {
        System.out.println("Rettangolo rosso.");
    }

    public void disegnaCerchio() {
        System.out.println("Cerchio rosso.");
    }
}
```

Composite

Intento: rappresentare in maniera conforme oggetti composti. Si sfrutta una struttura ad albero per rappresentare le varie parti. I client possono trattare i singoli oggetti o oggetti composti uniformemente.

Motivazione: utile quando vogliamo raggruppare elementi per formare entità più complesse. Distinguere totalmente elementi semplici ed elementi composti risulta essere sconveniente. Tramite Composite descriviamo il tutto mediante una struttura ricorsiva e i client non faranno distinzione tra i vari elementi, semplificando il codice.

Esempio carrello: un carrello è un elemento composto da elementi, come libri, pizze, birre, regali. I regali a sua volta sono elementi composti da carta, nastro, contenuto.

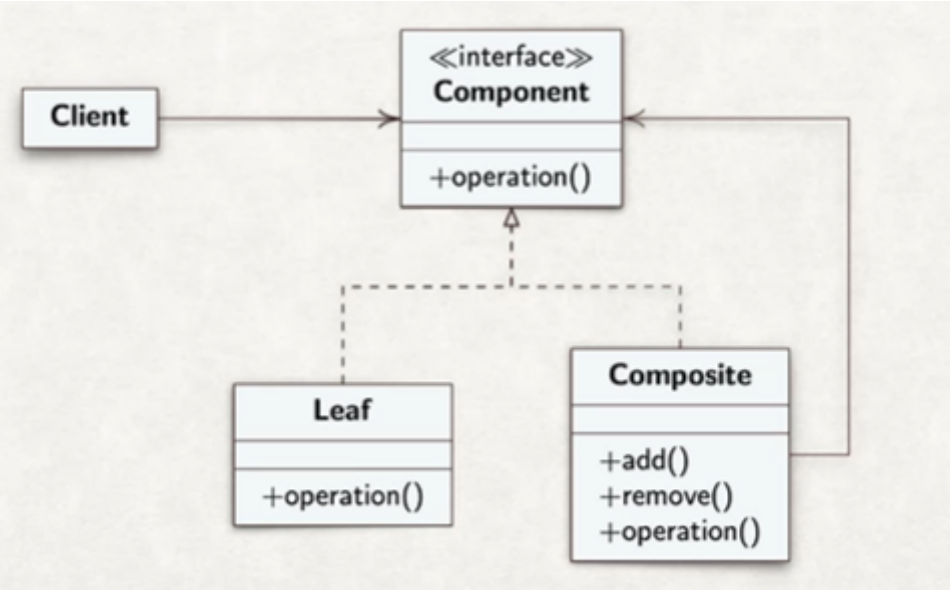


Soluzione

Component: interfaccia che rappresenta elementi semplici e composti. Dichiara le operazioni degli oggetti della composizione. Eventualmente può definire un'operazione per permettere ad un elemento di accedere all'oggetto padre.

Leaf: classe per gli oggetti semplici, sottoclasse di Component, implementa i metodi degli elementi semplici.

Composite: classe per gli elementi composti, anche lei sottoclasse di Component. Definisce il comportamento degli elementi aggregati, tiene un riferimento agli oggetti che contiene e implementa operazioni per gestirli.

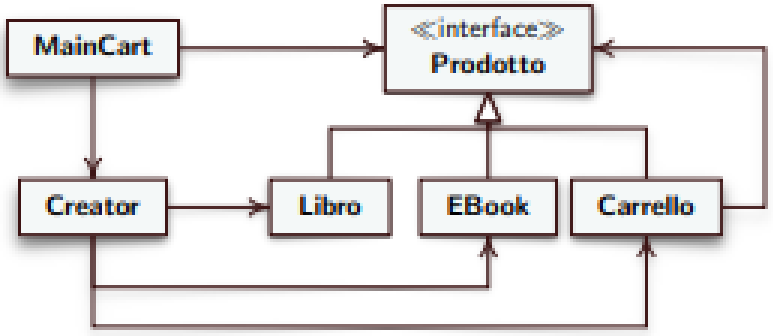


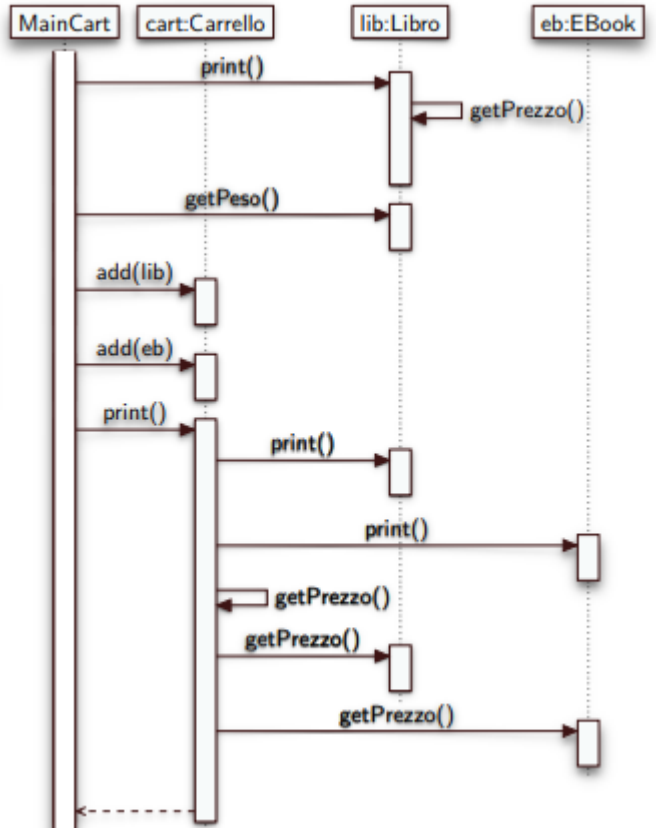
Sia Leaf che Composite implementano le operazioni di Component, poichè i Client andranno ad invocare quella indipendentemente. Composite ha un'associazione con Component dato che ha al suo interno riferimenti di tipo Component (gli elementi che contiene).

Mediante l'operazione `add()` possiamo aggiungere elementi di tipo Component al nostro elemento Composite, che ha una lista di elementi Component al suo interno. Analogamente `remove()` rimuove un elemento dalla lista.

In questo modo i metodi di aggiunta e rimozione non sono visibili ai Client, in questo caso abbiamo **sicurezza** perchè evitiamo a priori di eseguire queste operazioni su Leaf e questo non avrebbe senso, ma **perdendo trasparenza** perchè i client non hanno il potere di manipolare gli oggetti Composite. L'alternativa sarebbe dichiarare tali metodi in Component, in questo modo però perdiamo sicurezza.

Vediamo adesso un esempio.





Supponiamo di voler implementare Composite nel voler astrarre il concetto di Carrello, che è composto, mentre tutti gli elementi posizionabili nel carrello sono gli elementi semplici.

L'**aggregazione** è un tipo di associazione tra oggetti molto più forte, poichè i singoli componenti non hanno senso di esistere all'infuori dell'elemento aggregato. Mediante la **composizione** vogliamo mantenere la possibilità di permettere ai singoli elementi di esistere anche all'infuori dell'elemento Composite.

Vediamo un esempio software:

```
// COMPONENT - add() e remove() inclusi, quindi più trasparenza ma meno sicurezza
public interface Prodotto {
    public void print();
    public float getPrezzo();
    public int getPeso();

    public void add(Prodotto p);        // Leaf --> do nothing OR error
    public void remove(Prodotto p);     // Leaf --> do nothing OR error
}
```

```
// LEAF
public class Libro implements Prodotto {
    private String titolo;
    private float prezzo;
    private int peso;

    public Libro(String titol, float prez, int pes) {
        titolo = titol;
        prezzo = prez;
        peso = pes;
    }

    // Implementazione metodi ereditati e specifici di Libro

    @Override
    public void add(Prodotto p) { }

    @Override
    public void remove(Prodotto p) { }
}
```

```
// LEAF
public class EBook implements Prodotto {
    private String titolo;
    private float prezzo;

    public EBook(String titol, float prez) {
        titolo = titol;
        prezzo = prez;
    }

    // Implementazione metodi ereditati e specifici di EBook

    @Override
    public void add(Prodotto p) { }

    @Override
    public void remove(Prodotto p) { }
}
```

```java
// COMPOSITE
public class Carrello implements Prodotto {
    private List<Prodotto> nestedElem = new ArrayList<>();

    @Override
    public void print() {
        System.out.println("Carrello: ");
        for (Prodotto res : nestedElem)
            res.print();
        System.out.println("Prezzo totale: " + getPrezzo());
    }

    @Override
    public float getPrezzo() {
        return nestedElem.stream().map(e -> e.getPrezzo()).reduce(0f,
Float::sum);
    }

    @Override
    public int getPeso() {
        return nestedElem.stream().map(e -> e.getPeso()).reduce(0, Integer::sum);
    }

    @Override
    public void add(Prodotto p) {
        nestedElem.add(p);
    }

    @Override
    public void remove(Prodotto p) {
        nestedElem.remove(p);
    }

}
```

```java
public class MainCart {
    private static final Prodotto cart = Creator.getCarrello();
    private static final Prodotto lib = Creator.getLibro();
    private static final Prodotto eb = Creator.getEbook();

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

        cart.add(lib);
        cart.add(eb);
        cart.print();
        System.out.println("\nPeso del Carrello " + cart.getPeso());
    }
}
```

Decorator

Viene chiamato anche Wrapper.

Intento: modificare le responsabilità di un oggetto a runtime. Fornire un'alternativa flessibile alle sottoclassi per estendere funzionalità.

Motivazione: potremmo voler modificare le responsabilità dei singoli oggetti e non dell'intera classe. Le sottoclassi possono modificare o aggiungere comportamenti delle superclassi, ma non possono toglierli. Oltretutto non potremmo modificare il comportamento di una singola istanza.

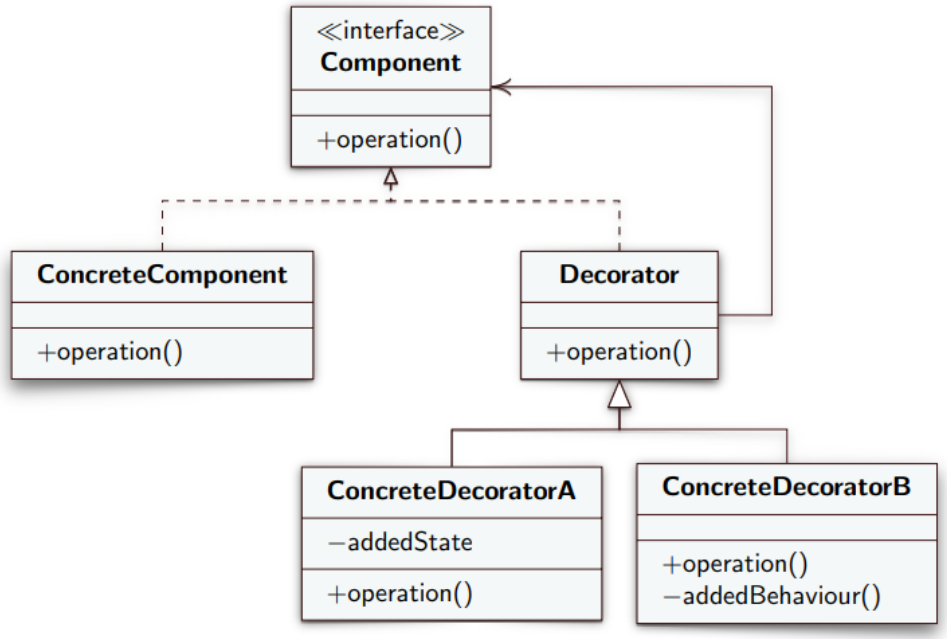
Soluzione

Component: definisce un'interfaccia per gli oggetti che possono avere aggiunte responsabilità dinamicamente.

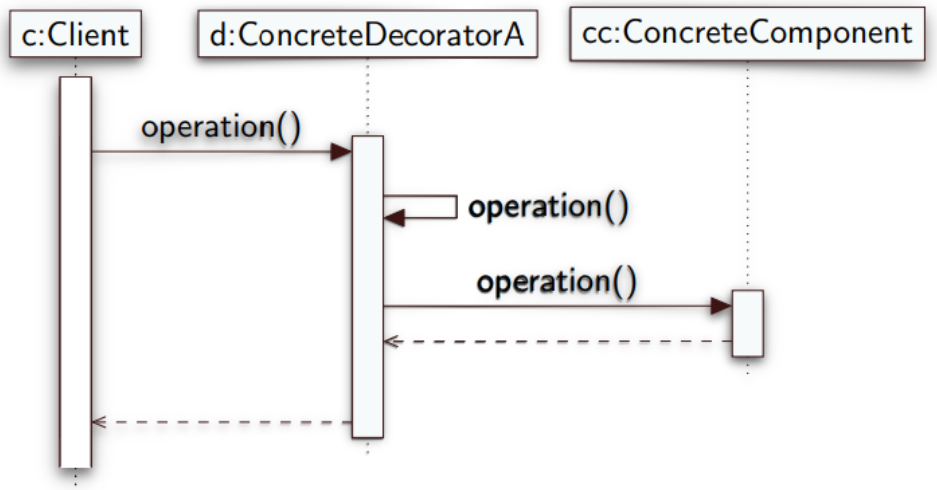
ConcreteComponent: definisce un oggetto componente da decorare

Decorator: mantiene un riferimento a un oggetto Component, definendo un'interfaccia conforme a quella di Component. Decorator inoltra le richieste che riceve dall'esterno a un Component.

ConcreteDecorator: implementa la responsabilità da aggiungere al component.



L'interfaccia Component definisce l'operazione generica che vogliamo fare sul componente di base. ConcreteComponent potrebbe essere per esempio Testo e `operation()` l'operazione di scrittura. Decorator è una classe che implementa in maniera differente `operation()`, tenendo al suo interno un riferimento di tipo Component. Su tale riferimento che tiene al suo interno chiama la stessa `operation()`. I vari ConcreteDecorator sono responsabilità aggiuntive che possiamo dare al Component, entrambi implementano a loro modo `operation()`, eventuali metodi nuovi dovranno essere privati perchè l'interfaccia è la stessa.



Client chiama il generico metodo `operation()` di ConcreteDecorator, che a sua volta chiama `operation()` sull'istanza ConcreteComponent che tiene al suo interno.

La classe Decorator tiene un riferimento a un tipo Component che deve essere inizializzato. Il Decorator ha un costruttore che prende in ingresso un tipo Component, potendo quindi ricevere sia ConcreteComponent che ConcreteDecorator.

Vediamo un esempio:

```
interface Component {
    public void operation();
}
```

```
class ConcreteComponent implements Component {
    @Override
    public void operation() {
        // IMPLEMENTAZIONE COMPONENT
    }
}
```

```
class Decorator implements Component {
    private final Component innerC; // RIFERIMENTO COMPONENT

    public Decorator(final Component c) {
        innerC = c;
    }

    @Override
    public void operation() {
        innerC.operation(); // INVIO CHIAMATA AL COMPOTENT
    }
}
```

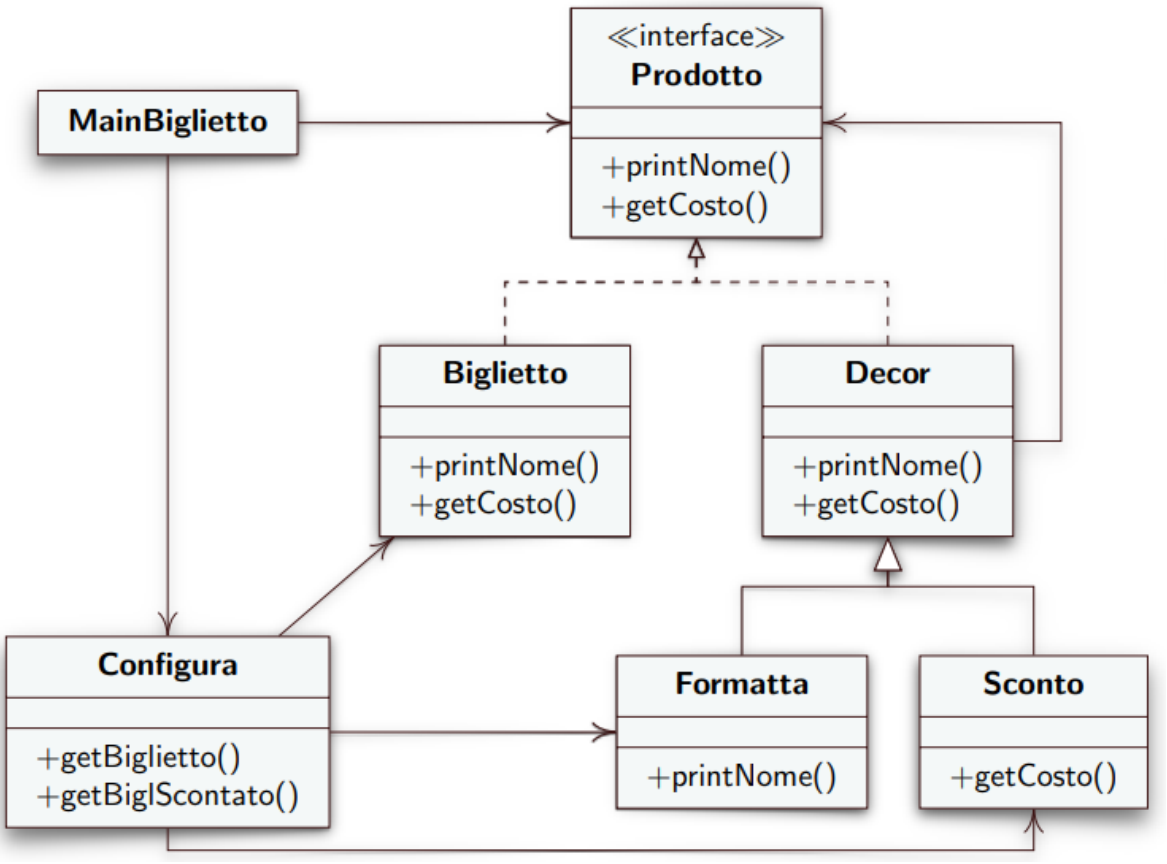
```
class ConcreteDecorator extends Decorator {
    public ConcreteDecorator(Component c) {
        super(c);
    }

    @Override
    public void operation() {
        super.operation();  // INOLTRO CHIAMATA AL COMPONENT
        // ALTRE OPERAZIONI CHE AGGIUNGONO RESPONSABILITA'
    }
}
```

```
Component c1 = new ConcreteDecoratorA(new ConcreteComponent());
```

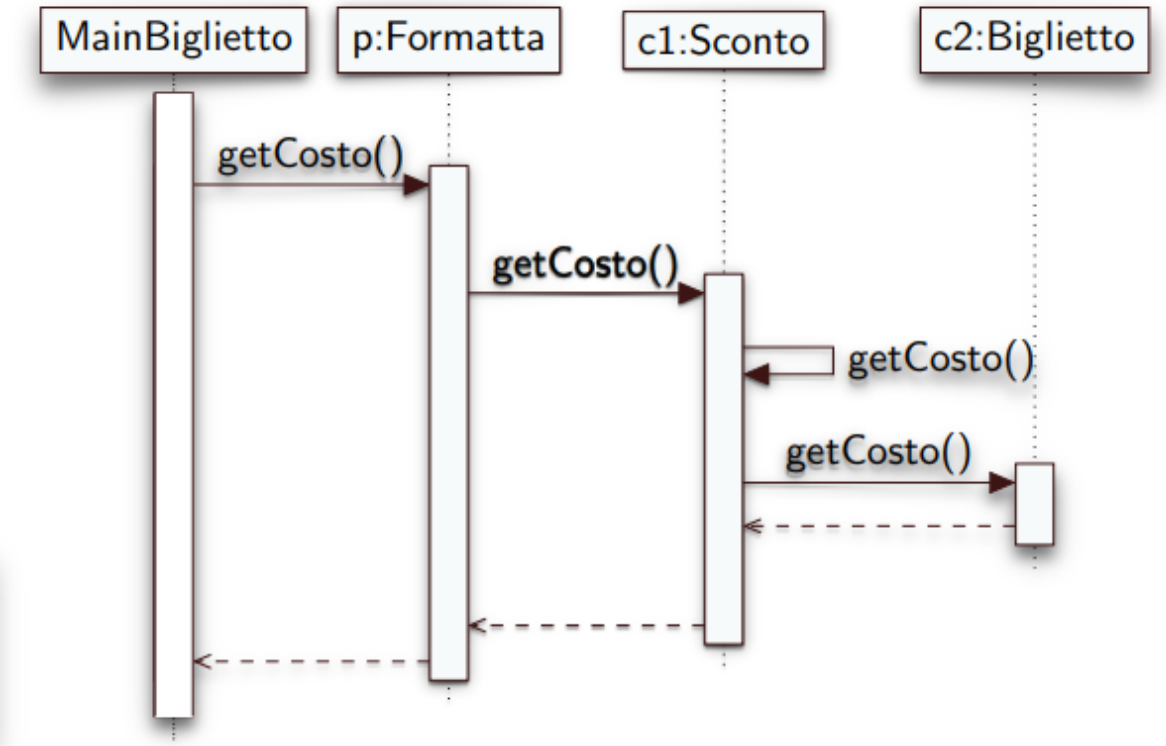
Con questo comportamento possiamo annidare tutte le volte che vogliamo dei Component.

Esempio pratico:

Abbiamo dei prodotti, per esempio Biglietto, ognuno con nome e costo, vi sono vari modi per calcolare il costo del prodotto in base agli sconti e modi di stampare i dettagli. Oltretutto vogliamo combinare a runtime i comportamenti di Formatta e Sconto sul componente Biglietto



La classe Biglietto porta dentro di sè le operazioni per stampa nome `printNome()` e `getCosto()`. Implementiamo due Decorator Formatta e Sconto che cambiano rispettivamente il comportamento di `printNome()` e `getCosto()`. Con questo design possiamo scontare il biglietto più volte in maniera molto più flessibile.



Vediamo l'implementazione software:

```
// COMPONENT
public interface Prodotto {
    public void printNome();
    public double getCosto();
    public Prodotto getWrapped();   // Per passare all'istanza più semplice
}
```

```
// CONCRETECOMPONENT
public class Biglietto implements Prodotto {

    @Override
    public void printNome(){
        System.out.println("[Biglietto base]: n.44322");
    }

    @Override
    public double getCosto(){
        return 100.0;
    }

    @Override
    public Prodotto getWrapped() {
        return this;
    }
}
```

```
// DECORATOR
public class Decorator implements Prodotto {

    private final Prodotto myComp;
```

```
    public Decorator(Prodotto myC){
        myComp = myC;
    }

    @Override
    public void printNome() {
        myComp.printNome();
    }

    @Override
    public void getCosto() {
        return myComp.getCosto();
    }

    @Override
    public void getWrapped() {
        return myComp;
    }
}
```

```
// CONCRETEDECORATOR A
public class Sconto extends Decorator {
    public Sconto(Prodotto myC) {
        super(myC);
    }

    @Override
    public double getCosto(){
        return super.getCosto()*0,95;
    }
}
```

```
// CONCRETEDECORATOR B
public class Formatta extends Decorator {
    public Formatta(Prodotto myC) {
        super(myC);
    }

    @Override
    public void printNome(){
        System.out.println("[Formatta]");
        return super.printNome();
    }
}
```

Ogni ConcreteDecorator deve alterare un solo comportamento, perchè ogni classe dovrebbe avere una singola responsabilità.

```
public class Configura {
    public static Prodotto getBiglietto(){
        return new Formatta(new Biglietto());
    }

    public static Prodotto getBigliettoScontato(){
        return new Formatta(new Sconto(new Biglietto()));
    }

    public static Prodotto getBigliettoSuperScontato(){
        return new Formatta(new Sconto(new Sconto(new Biglietto())));
    }
}
```

Nonostante a livello concettuale appaiono come lo stesso oggetto è bene ricordare che i riferimenti ai ConcreteComponent e ConcreteDecorator non sono gli stessi, seppur in qualche modo le decorazioni appaiano applicate ai componenti questi non sono la stessa cosa.

Pattern Comportamentali

State

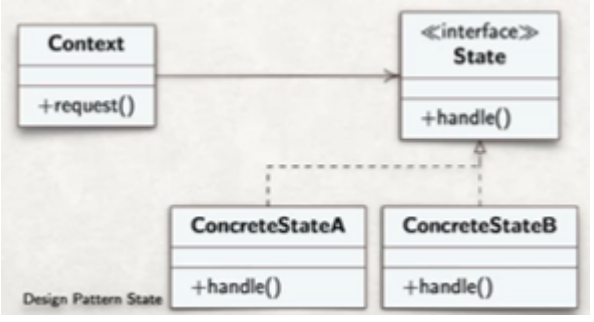
Intento: permettere ad un oggetto di alterare a runtime il suo comportamento quando cambia il suo stato interno, quasi come se l'oggetto cambiasse la sua classe.

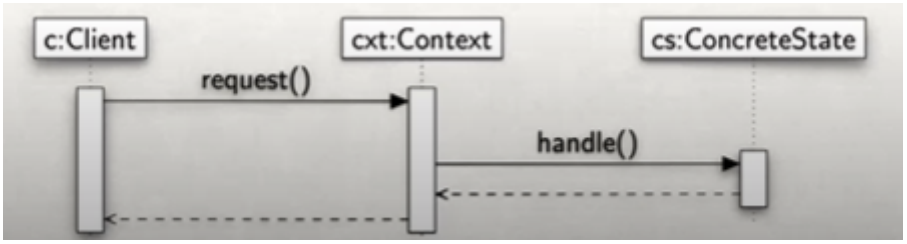
Problema: il comportamento dell'oggetto dipende dal suo stato e deve cambiare il comportamento a run-time. Le operazioni potrebbero avere grandi rami condizionali. Lo stato è spesso rappresentato da variabili enumerative.

Soluzione: consiste nel separare i comportamenti diversi in classi diverse. Ogni stato avrà una sua classe precisa:

- **Context** definisce l'istanza che interessa ai Client mantenendo un'istanza di una classe ConcreteState che definisce lo stato corrente
- **State** definisce un'interfaccia che incapsula il comportamento associato ad un ConcreteState
- **ConcreteState** sono le sottoclassi che implementano ognuno il comportamento di uno stato

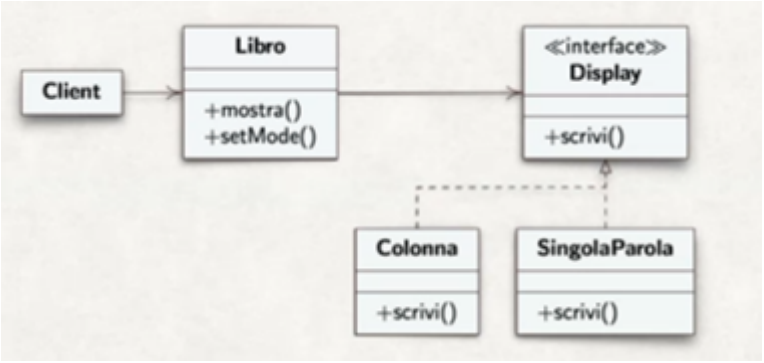
Come detto i client si riferiscono solo a Context, che al suo interno contiene un'istanza di tipo State implementata in uno dei ConcreteState.





Context può passare sè stesso come argomento all'oggetto ConcreteState, inoltre Context o ConcreteState decidono quale stato è il successivo in base alle circostanze.

Esempio: vogliamo avere vari modi per scrivere testo in un libro: modalità colonna e modalità singola parola per volta. Vogliamo usare codici molto diversi perchè vogliamo rendere la pagina il più leggibile possibile.




```
// CONTEXT
public class Libro{
    private String testo = "Questo è il testo del libro" + "con titolo Il Signore
Degli Anelli;
    private List<String> parole = Arrays.asList(testo.split("[\\s+]+")); //Questa
è una regex.

    private Display mode = new Colonna();

    public void mostra(){
        mode.scrivi(lista)
    }

    public void setMode(int x){
        switch(x){
            case 1: mode = new Colonna(); break;
            case 2: mode = new SingolaParola(); break;
        }
    }
}
```

```
// CONCRETESTATE 1
public class Colonna implements Display {
    private final int numCar = 38;
    private final int numRighe = 12;

    public void scrivi(List<String> testo){
        // Implementazione per scrivi colonna
    }
}
```

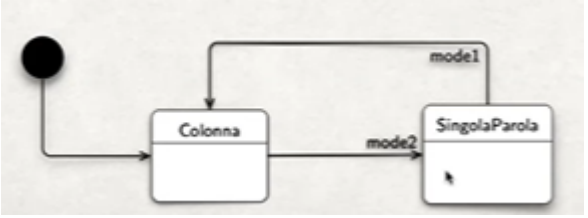
```java
// CONCRETESTATE 2
public class SingolaParola implements Display {
     public void scrivi(List<String> testo){
        // Implementazione per scrivi singola parola
     }
}
```

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

```java
//CLIENT
public class Client {
    public static void main(String[] args){
        Libro l = new Libro();
        l.mostra();
        l.setMode(2);
        l.mostra();
    }
}
```

Conseguenze: il comportamento associato a uno stato si trova nella singola classe ConcreteState, quindi è facile aggiungere nuove classi e quindi nuovi stati. La logica che gestisce il cambio di stato risiede in Context, e questa centralizzazione della decisione dello stato da assumere permette di evitare confusione e stati inconsistenti.

Di seguito vediamo un esempio di **diagramma UML degli stati**, che ha una sua grafica diversa dalle classi. Questo è un rettangolo con angoli arrotondati. Affiancato alla linea va indicato l'evento che causa il cambio di stato. La pallina nera permette di indicare lo stato iniziale. Nel rettangolo ovviamente va il nome dello stato, sotto potremmo indicare attività che subentrano in quel particolare stato.

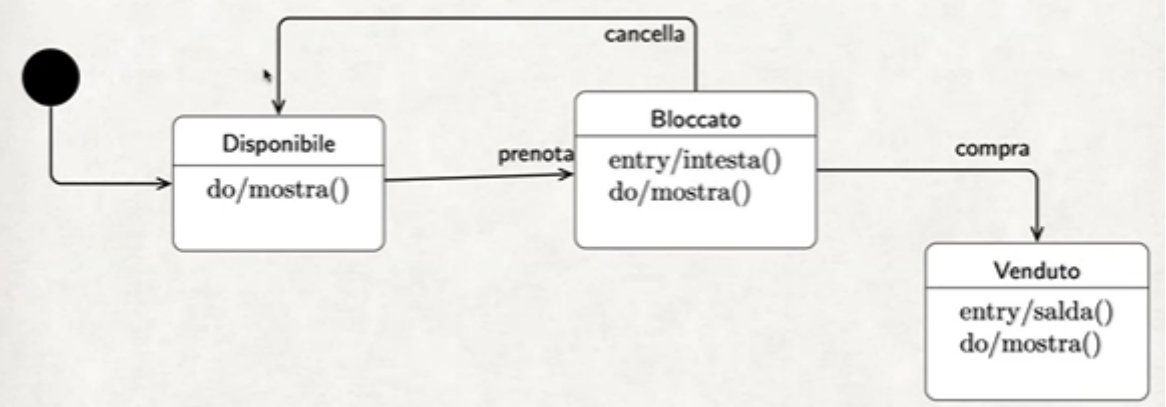


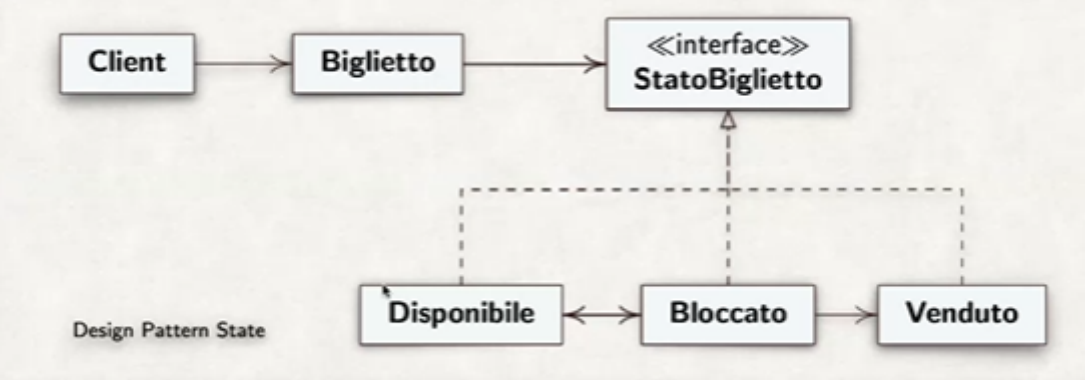
Immaginiamo di avere dei requisiti e di voler implementare il design pattern State:

Il sistema software dovrà fornire la possibilità di **prenotare** e **acquistare** un **biglietto**. Si potrà **annullare** la prenotazione, ma non l'acquisto. Ogni biglietto ha un **codice**, un **prezzo**, una **data** di acquisto, il **nome dell'intestatario**. Per la prenotazione si dovrà dare il nome dell'intestatario.

Procediamo con un'analisi semantica per costruire classi, attributi e metodi.

- **Classi**: Biglietto
- **Attributi**: codice, prezzo, data, nome
- **Operazioni**: prenota, acquista, annulla
- La classe Biglietto ha tre possibili stati: disponibile, bloccato, venduto





Vediamo il codice, prima come implementare questo comportamento in un'unica classe, quali sono le criticità, e successivamente come implementare il design pattern State.

```
public class Biglietto {
    private String codice = “XYZ”;
    private String nome;
    private int prezzo = 100;
    private enum StatoBiglietto { DISP, BLOC, VEND }
    private StatoBiglietto stato = StatoBiglietto.DISP;

    public void prenota(String s) {
        switch (stato) {
            case DISP:
                nome = s;
                System.out.println("Inserito nuovo intestatario. Biglietto
prenotato.");
                stato = StatoBiglietto.BLOC;
                break;
            case BLOC:
                nome = s;
                System.out.println("Inserito nuovo intestatario.");
                break;
            case VEND:
                System.out.println("Operazione non consentita.");
                break;
            }
    }

    public void cancella() {
        switch (stato) {
            case DISP:
                System.out.println("Lo stato era gia' Disponibile");
                break;
```

```
            case BLOC:
                System.out.println("Cambia stato da Bloccato a Disponibile");
                nome = "";
                stato = StatoBiglietto.DISP;
                break;
            case VEND:
                System.out.println("Operazione non consentita");
                break;
        }
    }

    public void mostra() {
        System.out.println("Prezzo: " + prezzo + " codice: " + codice);
        if (stato == StatoBiglietto.BLOC || stato == StatoBiglietto.VEND)
        System.out.println("Nome: " + nome);
    }

    public void compra() {
        switch (stato) {
        case DISP:
            System.out.println("Non si puo' pagare, bisogna prima intestarlo");
            break;
        case BLOC:
            System.out.println("Cambia stato da Bloccato a Venduto");
            stato = StatoBiglietto.VEND;
            System.out.println("Pagamento effettuato");
            break;
        case VEND:
            System.out.println("Il biglietto era gia' stato venduto");
            break;
        }
    }
}
```

Analisi del codice: effettivamente il codice si comporta correttamente dal punto di vista funzionale. Però ogni metodo ha vari rami condizionali, uno per ogni stato, e logica condizionale rende il codice difficile da comprendere e da modificare. Risulta più efficace separare i vari stati in più classi, le condizioni possono essere trasformate in messaggi per far variare lo stato dell'oggetto.

```
// State
public interface StatoBiglietto {
    public void mostra();
    public StatoBiglietto intesta(String s);
    public StatoBiglietto paga();
    public StatoBiglietto cancella();
}
```

```
// ConcreteState 1
public class Disponibile implements StatoBiglietto {
    @Override
    public void mostra(){ }

    @Override
```

```
    public StatoBiglietto intesta(String s) {
        System.out.println("DISP Cambia stato da Disponibile a Bloccato");
        return new Bloccato().intesta(s);
    }

    @Override
    public StatoBiglietto paga() {
        System.out.println("DISP Non si puo' pagare, bisogna prima intestarlo");
        return this;
    }

    @Override
    public StatoBiglietto cancella() {
        System.out.println("DISP Lo stato era gia' Disponibile");
        return this;
    }
}
```

```
// ConcreteState 2
public class Bloccato implements StatoBiglietto {
    private String nome;

    @Override
    public void mostra() {
        System.out.println("BLOC Nome: "+nome);
    }

    @Override
    public StatoBiglietto intesta(String s) {
        System.out.println("BLOC Inserito nuovo intestatario");
        nome = s;
        return this;
    }

    @Override
    public StatoBiglietto paga() {
        System.out.println("BLOC Cambia stato da Bloccato a Venduto");
        return new Venduto(nome).paga();
    }

    @Override
    public StatoBiglietto cancella() {
        System.out.println("BLOC Cambia stato da Bloccato a Disponibile");
        return new Disponibile();
    }
}
```

```
// ConcreteState 3
public class Venduto implements StatoBiglietto {
    private final String nome;
    private LocalDateTime dataPagam;

    public Venduto(String n) { nome = n; }
```

```
    @Override
    public void mostra() {
        System.out.println("VEND Nome: " + nome);
    }

    @Override
    public StatoBiglietto intesta(String s) {
        System.out.println("VEND Non puo' cambiare il nome nello stato Venduto");
        return this;
    }

    @Override
    public StatoBiglietto paga() {
        if (dataPagam == null) {
            dataPagam = LocalDateTime.now();
            System.out.println("VEND Pagamento effettuato");
        } else
            System.out.println("VEND Il biglietto era gia' stato pagato");
        return this;
    }

    @Override
    public StatoBiglietto cancella() {
        System.out.println("Operazione non consentita");
        return this;
    }
}
```

In questo modo i metodi non devono controllare in quale stato si trovano poichè la singola classe fa riferimento ad un singolo stato. L'intento è più chiaro, immediatamente visibile, l'interfaccia StatoBiglietto permette di ritornare il riferimento a un nuovo stato e per questo motivo viene detta fluent. Il fatto che nel precedente caso avevamo uno switch ci suggerisce immediatamente che l'OOP ci avrebbe aiutati. Abbiamo inoltre ridotto il numero di righe di codice per metodo, abbiamo complicato la struttura ma reso molto più evidente l'intento.

Observer

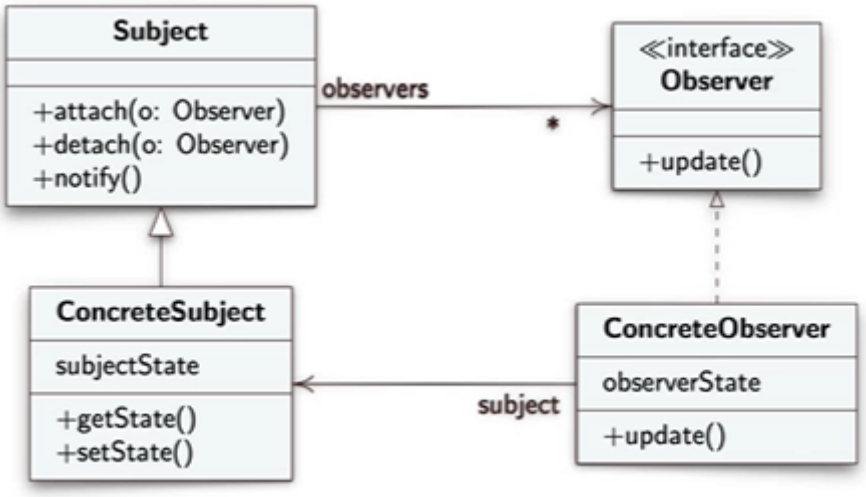
Detto anche Publish-Subscribe, Publish si riferisce a chi pubblica qualcosa e Subscribe si riferisce agli oggetti che si rapportano con esso.

Intento: definire una relazione uno a molti fra oggetti, così che quando un oggetto cambia stato tutti gli oggetti che dipendono da lui sono notificati e aggiornati automaticamente.

Motivazioni: utile in casi in cui un sistema deve mantenere la consistenza tra oggetti relazionati. Si potrebbe pensare che si viola il single responsability principle.

Applicabilità: casi in cui un cambiamento su un oggetto richiede il cambiamento di altri e non si conosce quanti e quali sarà necessario cambiare, o casi in cui un oggetto deve notificare agli altri oggetti cose senza però doversi preoccupare su chi sono tali oggetti.

Soluzione: gli oggetti chiave sono Subject e Observer. Un Subject può avere tanti Observer che dipendono da esso, e quando lo stato del Subject cambia gli Observer vengono notificati.



Subject mette a disposizione dei metodi:

- **attach()** permette agli observer di iscriversi, con parametro observer stesso. Tale oggetto iscritto viene aggiunto a una lista di tipo Observer (lo capiamo dall'asterisco vicino alla freccia)
- **detach()** permette agli observer di disiscriversi
- **notify()** scorre la sua lista di iscritti e chiama il metodo update di ogni Observer, li stiamo notificando chiamando il loro metodo di interfaccia update.

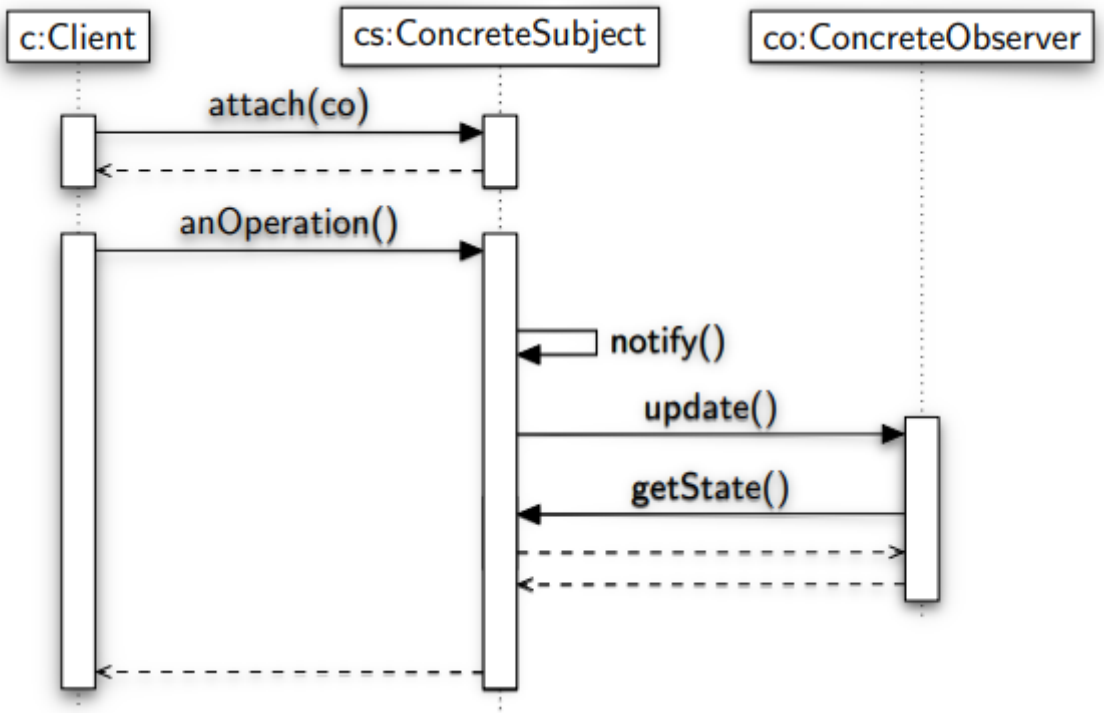
I **ConcreteObserver** devono **implementare il metodo update facendo operazioni che dipendono dal business**, per esempio potrebbe informarsi su qual è lo stato adesso e agire di conseguenza, eccetera.

ConcreteSubject è l'oggetto che contiene lo stato e che quando varia userà il metodo `notify()` ereditato da Subject per notificare i suoi iscritti Observer. Implementa anche setter e getter per il suo stato. Nel momento in cui varia lo stato chiama `notify()`.

Dentro il ConcreteObserver come detto vi è il metodo `update()`, e probabilmente avrà bisogno di conoscere lo stato in cui si trova ConcreteSubject. Esistono due varianti dell'Observer:

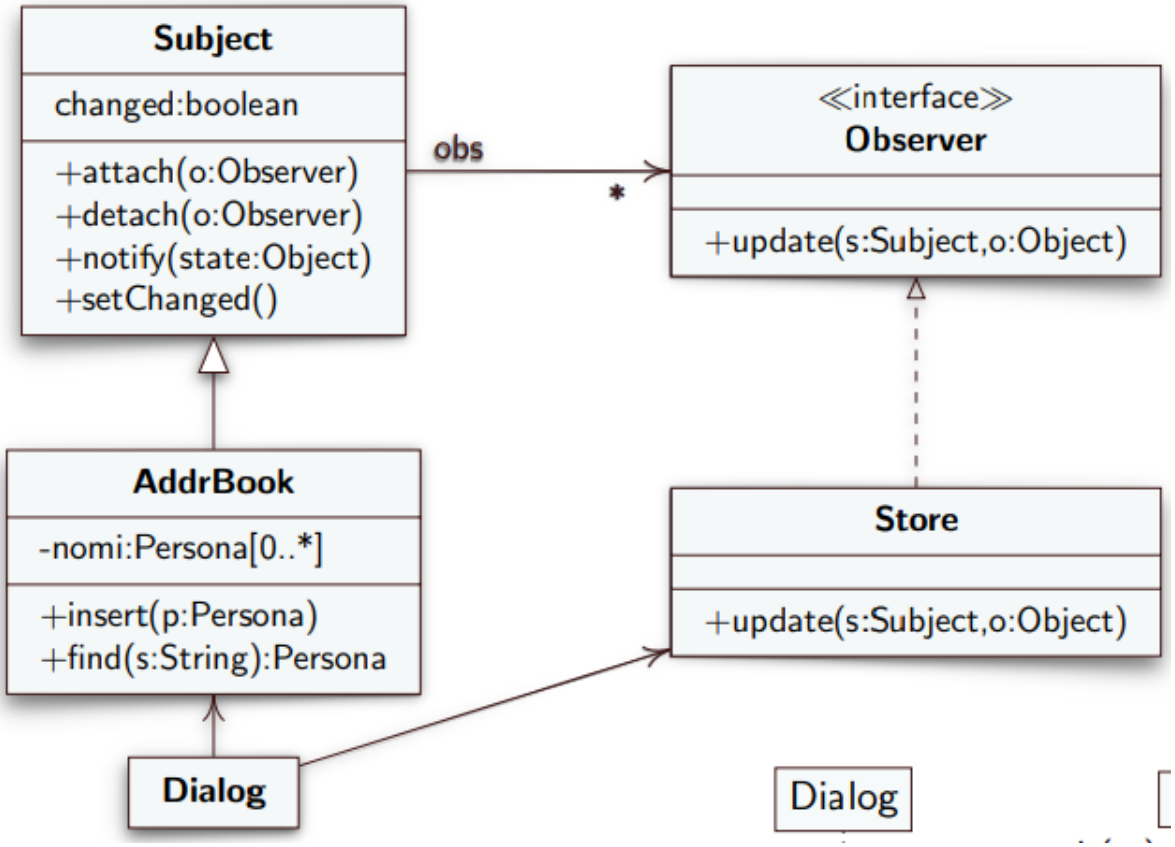
- **Pull Observer**: l'Observer chiama a sè lo stato di ConcreteSubject quindi in `update()` vi è una chiamata `getState()`.
- **Push Observer**: il ConcreteSubject manda come parametro il suo stato dentro `notify()`, così che ConcreteObserver possa prenderlo tramite `update()`.

Potrebbe essere utile passare l'identità del ConcreteSubject nel caso in cui ve ne siano di più, in questo modo gli Observer sanno già chi li sta notificando.



Con Observer abbiamo una ripartizione migliore del codice con minor dipendenze. Vi è minor accoppiamento tra la classe ConcreteSubject e tutti i suoi ConcreteObserver. Observer è un design pattern ampiamente utilizzato all'interno delle librerie Java.

Vediamo un esempio di codice:



```java
public class Subject {
     private List<Observer> obs = new ArrayList<>();
     private boolean changed = false;

     public void notify(Object state){ // Object è superclasse di tutte le classi
         if (!changed) return;
         for (Observer o : obs)
             o.update(this, state);
         changed = false;
     }

     public void setChanged() {
         changed = true;
     }

     public void attach(Observer o) {
        obs.add(o);
     }

     public void detach(Observer o) {
        obs.remove(o);
     }
}
```

```java
public class AddrBook extends Subject {
     private List<Persona> nomi = new ArrayList<>();

     public void insert(Persona p) {
         if (nomi.contains(p)) return;
         nomi.add(p);
         setChanged(); // la prossima notifica avverra'
         notify(nomi); // notifica i ConcreteObserver
     }

     public Persona find(String cognome) {
         for (Persona p : nomi)
            if (p.getCognome().equals(cognome)) return p;

         System.out.println("Cognome NOT FOUND");
         return null;
     }
}
```

```java
public interface Observer {
    public void update(Subject s, Object o);
}
```

```
public class Store implements Observer {

     @Override
     public void update(Subject s, Object o) {
         List<Persona> l = (List<Persona>) o;
         String nom;
         try (FileWriter f = new FileWriter("nomi.txt")) {
             for (Persona p : l) {
                 nom = p.getNome() + "\t" + p.getCognome() +
                 "\t" + p.getTelefono();
                 f.write(nom + "\n");
            }
         } catch (IOException e) { }
     }
}
```

```
public class Dialog {
     private static final AddrBook book = new AddrBook();
     private static final Store st = new Store();
     private static final Persona p1 = new Persona("Oliver", "Stone", "5", "NY");

    public static void main(String[] args) {
         book.attach(st);
         book.insert(p1);
     }
}
```

Conseguenze: Subject non ha bisogno di conoscere le ConcreteObserver, quindi vi è accoppiamento lasco. La notifica inviata da ConcreteSubject viene inviata a tutti. I ConcreteObserver possono essere rimossi a runtime senza problemi. Inoltre la notifica non dice cosa è cambiato ma indica il completamento di qualche operazione. Sarà compito dei ConcreteObserver gestire la situazione.

Avvertenze: se abbiamo più Subject che Observer anzichè tenere i riferimenti a Observer possiamo usare una **tabella comune** fra i Subject per mantenerne il riferimento. Non è obbligatorio che `notify()` venga chiamato subito, potremmo decidere di chiamarlo solo quando è opportuno farlo. Eventualmente gli Observer possono specificare gli eventi di interesse e l'update verrà inviato solo se tali eventi si verificano.

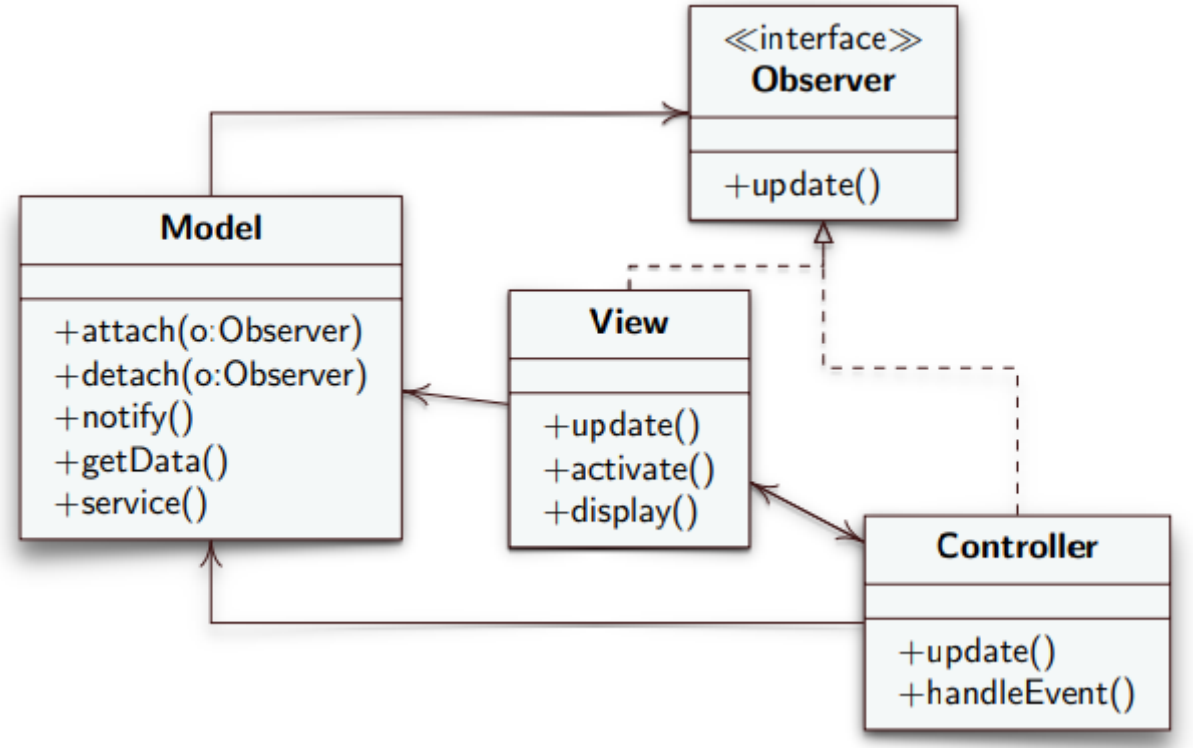
Come detto il problema affrontato da Observer si ritrova anche nella libreria java.util, dove troviamo Observable che svolge il ruolo di Subject, il quale notifica i ConcreteObserver mediante il metodo `notifyObservers()`. Da Java9 però Observer e Observable sono deprecate perchè il rallentamento delle notifiche potrebbe causare problemi. Si è cercato di risolvere mediante i **Reactive Streams** permettendo di non inondare i Subscriber. La classe Subscriber che svolge il compito di Observer e risulta implementato il **multithreading** per migliorare le prestazioni. Publisher invia item ai Subscriber in **maniera asincrona**, dato che i Subscriber tramite i metodi `cancel()` e `request()` hanno la possibilità di bloccare l'invio di messaggi e richiedere l'invio dei prossimi n messaggi.

Model View Controller (MVC)

In realtà è una variante del design pattern Observer, è un design pattern architetturale (per molti oggetti) per applicazioni interattive, si ritrova largamente in moltissimi ambiti. Individua tre componenti:

- **Model:** per funzionalità e dati. Non dipende dalla rappresentazione degli output e degli input. Registra View e Controller e li avvisa quando avvengono dei cambiamenti di stato significativi.
- **View:** per mostrare i dati. Generalmente ce ne sono molteplici e ognuna si associa ad un Controller. Sarà View a inizializzare il suo Controller e a mostrare i dati che legge da Model.
- **Controller:** per prendere gli input dell'utente e per interfacciarsi con Model. Traduce tali eventi in richieste per il Model o avvisi per la View.

Scrivere codice con questa struttura ci permette di cambiare facilmente le interfacce tra i vari componenti software. In questo pattern separiamo Model e View come facevamo con ConcreteSubject e ConcreteObserver, con l'aggiunta del Controller che si interfaccia con essi e prendere input. Laravel implementa MVC.

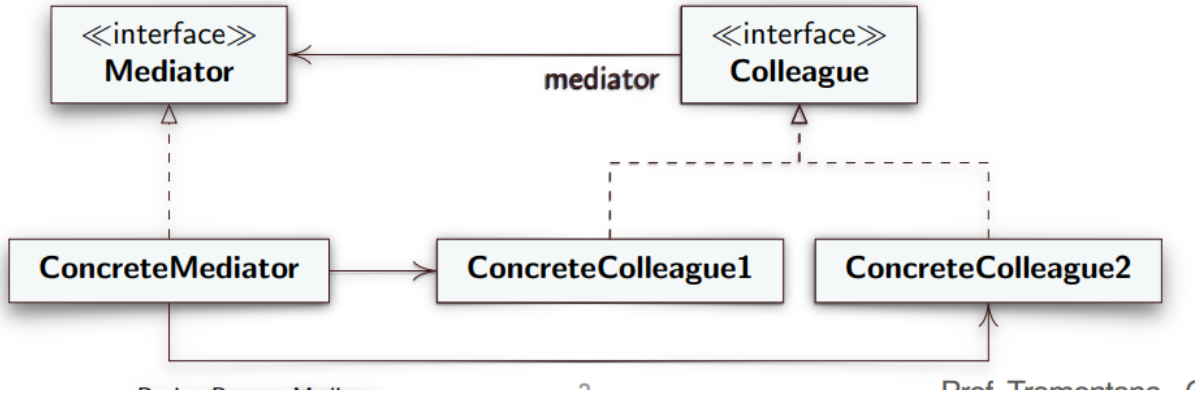


Mediator

Intento: definire un oggetto che centralizza le interazioni tra un gruppo di oggetti. Gli oggetti non devono tenere riferimenti verso tutti gli altri oggetti dato che causerebbe un accoppiamento stretto tra classi. Modificare una classe comporterebbe modificare tutte le altre. Più classi ci sono più ha importanza implementare il Mediator.

Motivazione: avere molte connessioni rende il sistema difficile da comprendere e da modificare, dato che il comportamento viene distribuito tra tutte le classi è difficile capirne l'intento. Avere una classe che gestisce le interazioni semplifica le interazioni stesse e il codice è più comprensibile. La classe Mediator conosce il modo con cui le classi vogliono comunicare tra loro.

Applicabilità: usare Mediator quando un insieme di oggetti comunicano in modo definito ma complesso, e quindi le dipendenze ne risultano difficili da comprendere. Molte dipendenze rendono difficile il riuso di un oggetto.



Soluzione:

- **Mediator:** definisce un'interfaccia per consentire agli oggetti Colleague di comunicare.
- **Colleague:** conoscono solo e unicamente il Mediator e lo sfrutta per inoltrare e ricevere messaggi.
- **ConcreteMediator:** implementa i metodi dichiarati in Mediator, definendo i comportamenti precisi per coordinare gli oggetti Colleague. Tiene un riferimento al suo interno per ogni ConcreteColleague.
- **ConcreteColleague:** comunicano unicamente con il Mediator per inviare e ricevere messaggi per e da altri ConcreteColleague.

Conseguenze: la complessità della comunicazione tra classi è tutta riposta sul Mediator, ciò rende gli oggetti Colleague più facili da modificare e riutilizzare. Eventuali errori non si ripercuotono su tutte le altre classi, ma al più solo sul Mediator. In genere a non essere riutilizzabile è proprio il Mediator.

```
// Mediator
public interface Mediator {
    void sendMessage(String destinatario, String messaggio);
    void addToRubrica(Peer peer);
}
```

```
// ConcreteMediator
public class Phone implements Mediator{
    List<Peer> rubrica = new LinkedList<>();

    public void sendMessage(String dest, String mes) {
        for (Peer nome : rubrica) {
            if(nome.getNome().equals(dest)){
                nome.save(mes);
                System.out.println("\nMessaggio: "+mes+"\nTo: "+dest);
                return;
            }
        }
        System.out.println("Impossibile inoltrare messaggio a "+dest);
    }

    public void addToRubrica(Peer peer) {
```

```
        rubrica.add(peer);
    }
}
```

```
// Colleague
public interface Peer{
    void save(String message);
    void send(String destinatario, String messaggio);
    void print();
    String getNome();
}
```

```
// ConcreteColleague 1
public class Tracker implements Peer{
    String nome;
    List<String> messaggi = new LinkedList<>();
    Mediator mediator;

    public Tracker(String nome, Mediator mediator){
        this.nome = nome;
        this.mediator=mediator;
    }

    public void save(String mes) {
        messaggi.add(mes);
    }

    public void print() {
        System.out.println("\nTracker messages:");
        for (String mes : messaggi) {
            System.out.println(mes);
        }
    }

    public void send(String dest, String mes) {
        mediator.sendMessage(dest, mes);
    }

    public String getNome(){
        return nome;
    }
}
```

```
// ConcreteColleague 2
public class Deliver implements Peer{
    ... ...
}
```

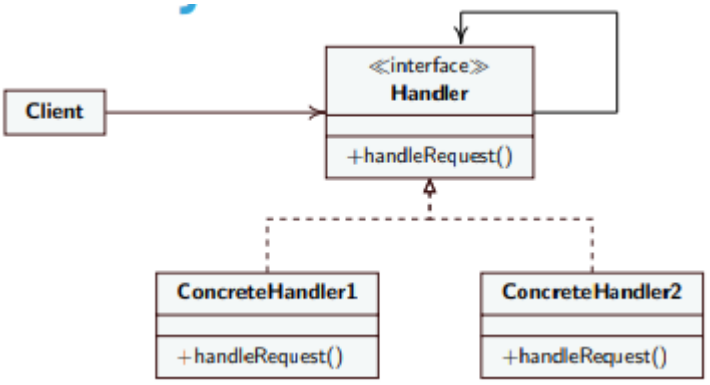
```
public class Client {
    public static void main(String[] args) {
        Mediator phone = new Phone();

        Peer antonio = new Tracker("Antonio", phone);
        Peer ciccio = new Deliver("Ciccio", phone);

        phone.addToRubrica(ciccio);
        phone.addToRubrica(antonio);

        ciccio.send("Antonio", "Sono il corriere. Registra ordine 2, grazie.");
    }
}
```

Chain of Responsibility

Intento: disaccoppare il mittente di una richiesta dal suo ricevente. Concatenare degli oggetti riceventi per passare la richiesta fino a un oggetto finale che eventulmente la gestirà.

Motivazione: immaginiamo un'interfaccia utente dove quest'ultimo può richiedere informazioni sul funzionamento di uno specifico componente. Qualora l'elemento cliccato non abbia una spiegazione associata il server restituirà informazioni di carattere più generico. L'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto che manda la richiesta. L'obiettivo del design pattern è dare l'opportunità a più oggetti di gestire la richiesta, i quali formeranno la catena di responsabilità.

Soluzione



- **Handler:** definisce l'interfaccia per inoltrare le richieste. Al suo interno ha riferimenti a oggetti di tipo Handler.
- **ConcreteHandler:** gestiscono le richieste di cui sono responsabili e ha un riferimento al suo successore a cui inoltrare le richieste che non può gestire.
- **Client:** manda richieste a un ConcreteHandler

Conseguenze: il client non deve conoscere chi risponderà alla sua richiesta, gli oggetti della catena devono solo tenere presente di quali richieste devono occuparsi e quali devono essere inoltrate al loro successivo. Tale struttura permette enorme flessibilità, potendo alterare facilmente la catena a runtime. La richiesta potrebbe non essere gestita se arriva alla fine della catena di responsabilità.

Implementazione:

```
public interface Chain {
    public void setNextChain(Chain nextChain);
    public void calculate(Numbers request);
}
```

```
public class Numbers {
    private int number1;
    private int number2;
    private String calculation;

    public Numbers(int n1, int n2, String calc){
        number1 = n1;
        number2 = n2;
        calculation = calc;
    }

    public int getN1(){ return number1; }
    public int getN2(){ return number2; }
    public String getCalc(){ return calculation; }
}
```

```
public class AddNumbers implements Chain{

    private Chain nextInChain;

    public void setNextChain(Chain nextChain) {
        nextInChain = nextChain;
    }

    public void calculate(Numbers request) {
        if(request.getCalc() == "add"){
            System.out.print(request.getN1()+ "+" +request.getN2()+ "="
+request.getN1()+request.getN2()));
        } else {
            nextInChain.calculate(request);
        }
    }
}
```

Similmente facciamo per Subtract ➡️ Multiply ➡️ Divide. Nel main avremo:

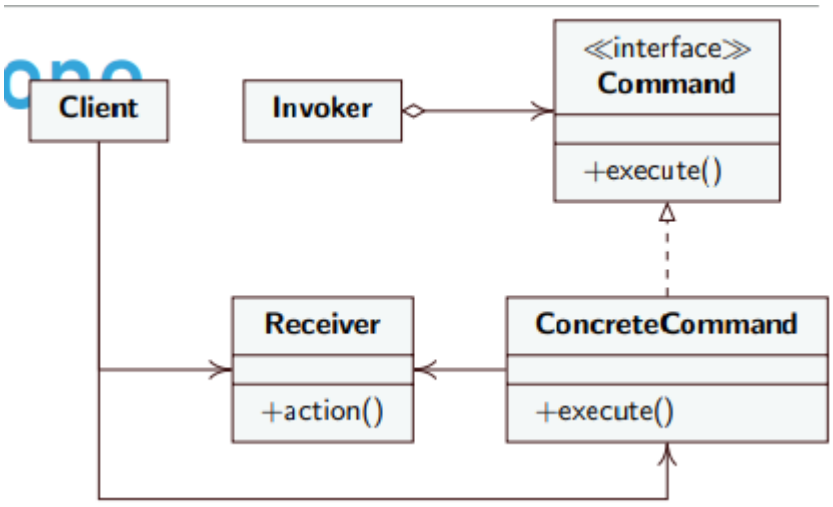
```
public class TestCalcChain {
    public static void main(String[] args){
        Chain chainCalc1 = new AddNumbers();
        Chain chainCalc2 = new SubtractNumbers();
        Chain chainCalc3 = new MultNumbers();
        Chain chainCalc4 = new DivideNumbers();

        chainCalc1.setNextChain(chainCalc2);
        chainCalc2.setNextChain(chainCalc3);
        chainCalc3.setNextChain(chainCalc4);
```

```
        Numbers requestAdd = new Numbers(4,2,"add");
        chainCalc1.calculate(requestAdd);               // Add: 4+2=6

        Numbers requestDiv = new Numbers(4,2,"div");
        chainCalc1.calculate(requestDiv);   // Add -> Sub -> Mul -> Div: 4/2=2

        Numbers randomRequest = new Numbers(4,2,"onf");
        chainCalc1.calculate(randomRequest); // Add -> Sub -> Mul -> Div -> NULL
    }
}
```

Command

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

Motivazione: potrebbe capitare di dover mandare una richiesta a oggetti senza conoscere nello specifico il ricevente o quale operazione soddisferà la richiesta. Quindi il tutto viene trasformato in un oggetto command che può essere archiviato o passato come parametro.



- **Command** definisce un'interfaccia per eseguire un'operazione.
- **ConcreteCommand** implementa la classe Command e chiama i metodi di Receiver.
- **Client** crea un ConcreteCommand impostandone il Receiver.
- **Invoker** chiede al comando di effettuare la richiesta.
- **Receiver** sa come effettuare le operazioni associate alla richiesta.

Client crea il ConcreteCommand impostandone il Receiver. L'oggetto Invoker tiene il ConcreteCommand ed effettua una richiesta tramite il comando `execute()` di Command. Il ConcreteCommand chiama le operazioni di Receiver per portare a termine la richiesta. Se non ci riesce e l'operazione è reversibile torna allo stato precedente.

Conseguenze: Command disaccoppia l'oggetto che invoca l'operazione da quello che sa come risolvere la richiesta. I comandi diventano oggetti e quindi possono essere manipolati a piacimento, possiamo quindi creare comandi composti da altri comandi implementando il design pattern Composite.

Implementazione: potrebbe semplicemente definire il legame tra ricevente e l'azione da eseguire oppure implementare tutto facendo a meno del ricevente. Se l'operazione è reversibile i ConcreteCommand dovranno conservare uno stato aggiuntivo oltre a quello attuale per permettere l'esistenza di operazioni di undo. Qualora si voglia rendere possibile l'esistenza di undo in sequenza Command conserverà una lista di comandi eseguiti.

```
// Receiver interface
public interface ElectronicDevice {
    public void on();
    public void off();
    public void volumeUp();
    public void volumenDown();
}
```

```
// Receiver
public class Television implements ElectronicDevice {
    private int volume = 0;

    public void on() {
        System.out.println("TV is on");
    }

    public void off() {
        System.out.println("TV is off");
    }

    public void volumeUp() {
        volume++;
        System.out.println("TV Volume is at: " + volume);
    }

    public void volumenDown() {
        volume--;
        System.out.println("TV Volume is at: " + volume);
    }
}
```

```
// Command
public interface Command {
    public void execute();
    public void undo();
}
```

```
// ConcreteCommand 1
public class TurnTVOn implements Command {
    ElectronicDevice theDevice;

    public TurnTVOn(ElectronicDevice newDevice){
        theDevice = newDevice;
    }

    public void execute() {
        theDevice.on();
```

```
    }

    public void undo() {
        theDevice.off();
    }
}
```

```
// ConcreteCommand 2
public class TurnTVOff implements Command{
    ElectronicDevice device;

    public TurnTVOff(ElectronicDevice d){
        device = d;
    }

    public void execute(){
        device.on();
    }

    public void undo(){
        device.off();
    }
}
```

```
// Invoker
public class Telecomando {
    Command on;
    Command off;
    Command lastCommand;

    public Telecomando(Command onC, Command offC){
        on = onC;
        off = offC;
    }

    public void turnOn(){
        on.execute();
        lastCommand = on;
    }

    public void turnOff(){
        off.execute();
        lastCommand = off;
    }

    public void undo(){
        lastCommand.undo();
    }

    public void redo(){
        lastCommand.execute();
    }
}
```

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

        ElectronicDevice tvSony = new Television("Sony");
        TurnTVOff commandOff = new TurnTVOff(tvSony);
        TurnTVOn commandOn = new TurnTVOn(tvSony);

        Telecomando telecomandoSony = new Telecomando(commandOn, commandOff);
        telecomandoSony.turnOn();
        telecomandoSony.turnOff();
    }
}
```

Iterator

Intento: fornire un modo per accedere agli elementi di un oggetto aggregato in maniera sequenziale senza esporre la sua implementazione. Utile quando si vuole fornire un'interfaccia comune per attraversare più collezioni.

Motivazioni: un oggetto aggregato, dovrebbe fornire un modo per accedere ai suoi elementi senza che ci sia bisogno di esporre la sua struttura interna. Dato che si potrebbe voler accedere la lista in vario modo, potremmo non voler appesantire l'interfaccia con le operazioni. L'idea chiave è quella di togliere la responsabilità dell'accesso dall'oggetto lista e metterla in un oggetto itertor, in grado di tenere traccia di quali oggetti sono stati visitati e dell'oggetto corrente.

Implementare i vari aggregati in vario modo, magari con strutture dati differenti ci renderebbe la vita difficile dato che per esempio un ArrayList dovrà scorrere diversamente rispetto una HashMap.

```
// A simple Notification class
class Notification {
    // To store a notification message
    private String message;

    public Notification(String message){
        this.message = message;
    }

    public String getMessage(){
        return message;
    }
}
```

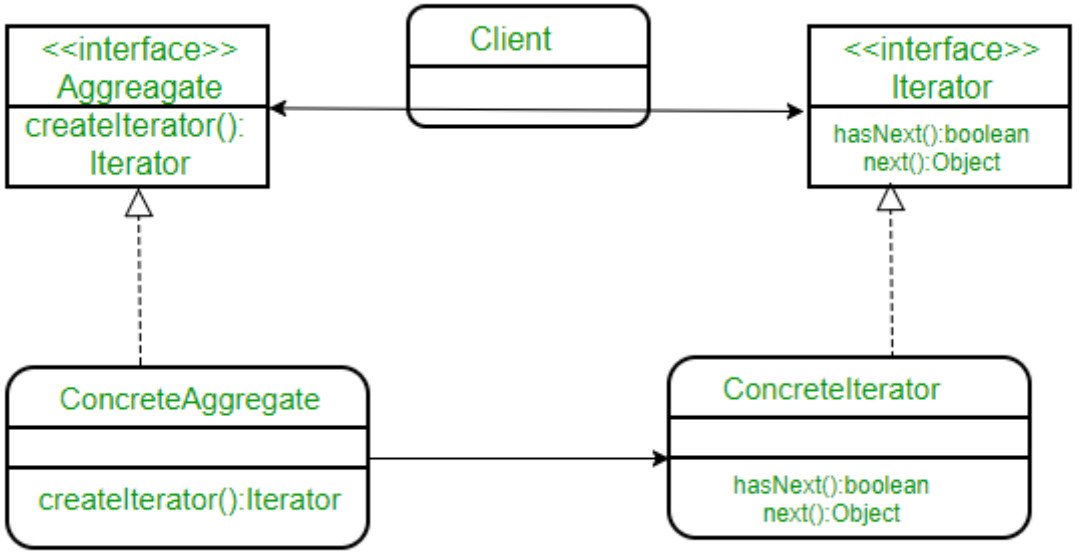
```
// Collection interface
interface CollectionIterator {
    public Iterator createIterator();
}
```

```
// Collection of notifications - AGGREGATE
class instagramNotifications implements CollectionIterator {
```

```
    ArrayList<Notification> notificationList;

    public NotificationCollection(){
        notificationList = new ArrayList<Notification>();

    }

    public void addItem(String str) {
        Notification notification = new Notification(str);
        notificationList.add(notification);
    }

    public Iterator createIterator() {
        return notificationList.iterator()
    }
}
```

Questa funzione `iterator()` ritorna un oggetto di tipo Iterator ed è possibile usarla su qualsiasi collection in java, quindi qualsiasi collezione di oggetti. In questo modo unifichiamo l'interfaccia di scorrimento delle collections.

```
public class PhoneAlerts{
    CollectionIterator instagramIter;
    CollectionIterator updateIter;

    public PhoneAlerts( CollectionIterator newNot, CollectionIterator newUpd){
        instagramIter = newNot;
        updateIter = newUpd;
    }

    public void showAll(){
        Iterator notifications = notificationIter.createIterator();
        Iterator updates = updateIter.createIterator();

        System.out.println("From Instagram: ");
        printIterator(notifications);

        System.out.println("Updates: ");
        printIterator(updates);
    }

    public void printIterator(Iterator iterator){
        while(iterator.hasNext()){
            Notification notification = (Notification) iterator.next();
            System.out.println(notification.getMessage());
        }
    }
}
```



Template Method

Intento: permettere la codifica di un algoritmo di una certa operazione delegando l'implementazione di alcuni suoi passi alle sottoclassi. Definire la struttura di un algoritmo in maniera parametrizzabile, attraverso la sottoclasse possiamo cambiare l'implementazione di alcune funzioni private usate nel template method senza alterarne la struttura principale.

Motivazione: immaginiamo di avere degli application framework. I framework non sono librerie in quanto su quest'ultime si costruisce altro codice, i framework spaziano su più livelli. Creano un contesto preconfezionato con degli spazi vuoti, appunto parametrizzati, così da adattarli a più situazioni.

Applicabilità: viene usato per implementare le parti invarianti di un algoritmo una sola volta e lasciare che le sottoclassi implementino le parti variabili. I comportamenti comuni sono fattorizzati in una sola classe. I template method definiscono quali operazioni sono hook, ovvero di aggancio con cui possono operare facendo override, e quali sono puramente astratte e che quindi necessitano di certo un override.

Conseguenze: il template method crea una classe con un metodo diviso in più parti lasciando alcune implementazioni vuote. In java faccio tutto ciò tramite classi astratte. Qui inverto il flusso di controllo: il codice implementato dalle sottoclassi viene eseguito a seguito di un'invocazione di un metodo definito nella classe padre. Un caso particolare è quello in cui implementiamo il template method usando metodi costanti ovvero che non modificano lo stato interno dell'oggetto, sono marcati con la keyword const o final.

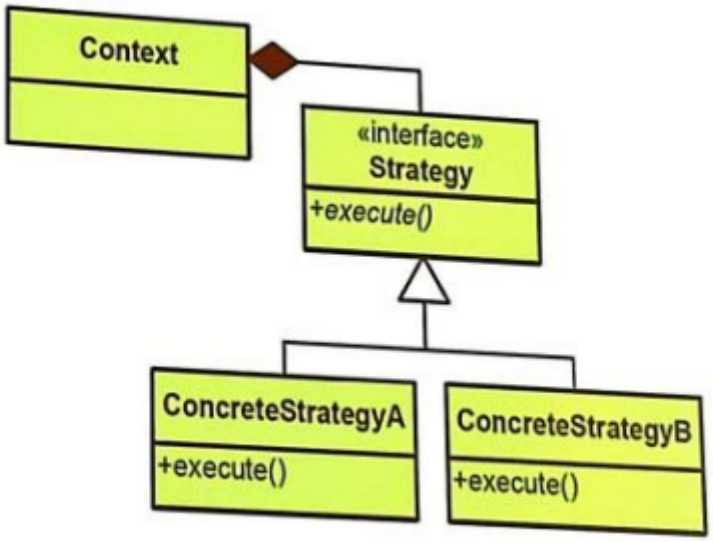
Strategy

Intento: definire una famiglia di algoritmi correlati, incapsularli singolarmente e renderli intercambiabili e consentire la modifica di un algoritmo a prescindere dal contesto dove viene usato.

Motivazione: immaginiamo di avere un testo che deve essere diviso in righe. Esistono vari algoritmi che possono farlo e non è desiderabile che le classi debbano adattarsi ad ognuno di essi. Dato che lo scopo è simile, è bene incapsularli per renderli intercambiabili. Integrare nel client la logica degli algoritmi renderebbe tutto più difficile. Un algoritmo che viene incapsulato viene chiamato Strategy, la strategia risulta essere cambiabile facilmente.

Esempio: immaginiamo di avere una classe Composition responsabile di mantenere e aggiornare le suddivisioni in righe. Le varie strategie non sono implementate in Composition ma lo sono in Compositor. Le sottoclassi Compositor implementano le varie strategie e Composition ne tiene un riferimento, ogni qual volta vorrà cambiare strategia inoltra questa responsabilità all'oggetto Compositor.

Applicabilità: quando molte classi correlate differiscono solo nel comportamento e quando queste forniscono un modo per configurare una classe con uno dei molti comportamenti simili. Oppure quando vi sono diverse varianti di un algoritmo, magari differiscono nei compromessi nell'utilizzo delle risorse.



I ConcreteStrategy implementano l'interfaccia Strategy, che è conosciuta dal Context. Il comando `execute()` implementa i diversi comportamenti delle concrete. Il client si rivolge al Context, che rimanda le richieste alle concrete.

Vantaggi: possiamo costruire famiglie di algoritmi, aggiungerne altre facilmente, ed è un'alternativa all'eredità, che vincolerebbe il comportamento delle sottoclassi alle superclassi, oltretutto è possibile farlo a runtime. Il comportamento desiderato può essere ottenuto evitando molti costrutti condizionali

Svantaggio: i client devono essere a conoscenza delle varie differenze tra strategie pertanto è bene implementare strategy solo quando la variazione nel comportamento è rilevante per i client. Vi è anche un eccesso di comunicazione tra Strategy e Context dato che Context va a chiamare la stessa funzione, e certi ConcreteStrategy più semplici potrebbero ignorare eventuali parametri in ingresso. Sicuramente è un piccolo spreco di risorse.

Template vs Strategy, State vs Strategy

Il pattern **Template** viene utilizzato per **fissare lo scheletro di un algoritmo** in cui i vari passi possono essere implementati in modi diversi, definendo così una struttura generale per l'algoritmo ma delegando l'implementazione dei passi specifici alle sottoclassi. Ciò consente di creare diverse varianti dell'algoritmo senza dover riscrivere l'intera struttura di base.

Il pattern **State**, simile al pattern Strategy, ha anch'esso uno scope di oggetto e **coinvolge la gestione dello stato di un oggetto**. Mentre nel pattern Strategy l'interfaccia comune definisce gli algoritmi e il contesto utilizza una strategia specifica (che deve essere nota agli utilizzatori), nel pattern State ci si concentra sulla gestione degli stati interni di un oggetto, che influenzano il suo comportamento.

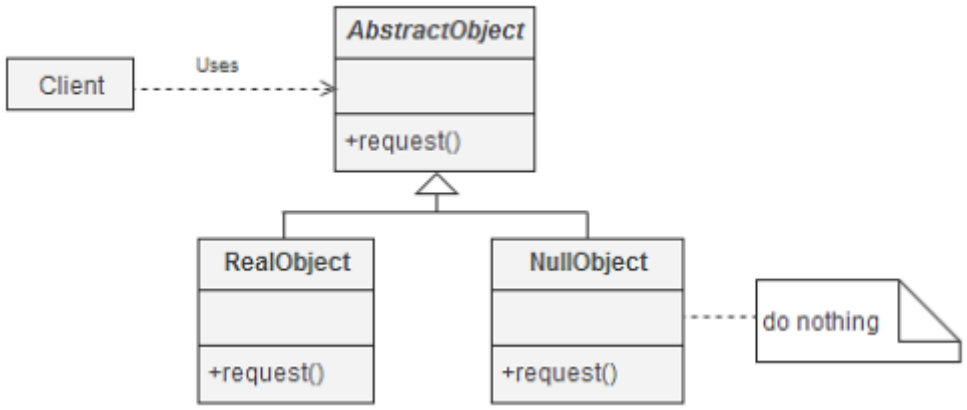
Nel pattern Strategy e nel pattern State i cambiamenti di algoritmo o di stato avvengono a livello di oggetto e possono avvenire frequentemente. Nel pattern Template, invece, i cambiamenti avvengono a livello di classe.

Null Object

Intento: prevede l'utilizzo di un oggetto speciale chiamato Null Object al posto di un riferimento nullo ad un oggetto. Viene utilizzato per fornire un'implementazione di default per un oggetto quando quello reale non è disponibile. L'obiettivo è evitare ai client la gestione esplicita degli oggetti nulli. Il Null Object implementa la stessa interfaccia dell'oggetto reale che rappresenta ma con logica vuota o predefinita. Pertanto viene considerata una tecnica di refactoring.

Motivazione: permette di evitare e gestire errori a riferimenti nulli e semplifica il codice evitando espressioni condizionali per controllare se l'oggetto è nullo. Inoltre evita che i client dipendano dagli oggetti concreti favorendo l'utilizzo delle interfacce, rendendo il codice più modulare.

Soluzione: basta creare una gerarchia di classe, una superclasse astratta conosciuta dal client e due sottoclassi, RealObject e NullObject. Entrambe hanno la stessa interfaccia ma NullObject ha come implementazione do nothing.



Conseguenze: i NullObject non sollevano eccezioni quando vengono chiamati, quindi potrebbero rendere difficile rilevare errori. Qualora un NullObject venga utilizzato in modo improprio potrebbe portare a comportamenti indesiderati. Pertanto, rende la pratica del debugging più difficoltosa poichè non vi sono segnali d'errore.

Applicabilità: quando vuoi che i client ignorino il fatto che l'oggetto possa essere effettivamente Null, e quando vogliamo mantenere il comportamento do nothing.

Processi di sviluppo software

Un processo software descrive i task necessari allo sviluppo di un prodotto software collegate tra loro, analizzando la loro priorità. Si procede con:

- **Analisi dei requisiti**
- **Progettazione**
- **Implementazione codice**
- **Convalida / Testing**
- **Manutenzione**

Durante **l'analisi dei requisiti** raccogliamo le specifiche necessarie, servizi richiesti e vincoli software. In genere i requisiti sono funzionalità molto piccole, e nel loro insieme formano una feature che servirà a soddisfare un bisogno dei clienti. Le aziende scelgono se accettare dei progetti o no sia in termini di fattibilità sia in termini di cosa il progetto porta al team in termini di esperienza.

Scendendo più nei dettagli, quando si fa quella che viene detta **l'ingegneria dei requisiti** si procede con:

- **Studio di fattibilità**
- **Analisi requisiti**
- **Specifica requisiti**
- **Convalida requisiti**

Distinguiamo due tipologie di requisiti:

- **Funzionali:** cosa il sistema deve fare.
- **Non-funzionali:** come il sistema lo fa in termini di affidabilità, efficienza, manutenibilità.

La **fase di progettazione** serve a stabilire i componenti e le loro relazioni che ci permetteranno di realizzare il software desiderato. Dividiamo la fase di progettazione in:

- **Suddivisione requisiti**
- **Progettazione architettura**
- **Specifica delle responsabilità dei sottosistemi**
- **Progettazione di interfacce, componenti, strutture dati, algoritmi**

Ognuna delle suddette attività produce un documento corrispondente per descrivere il modello. Il risultato della progettazione viene passato ai developer che passeranno alla **fase di implementazione**. Spesso la progettazione e l'implementazione vengono alternate continuamente.

La **fase di convalida** serve a mostrare che il software soddisfa le richieste del cliente. Quindi un modo per fare la fase di convalida è scrivere i test e confrontare gli output del sistema con quello che ci si aspetta oppure mediante revisione di persone esperte. Tale fase di convalida serve unicamente a validare il codice, non i programmatori che hanno scritto quel codice.

Possiamo distinguere varie categorie di test:

- **Unit Test**: mirano a testare i singoli componenti indipendentemente, come funzioni, oggetti o loro raggruppamenti

- **Test di sistema**: si testa l'intero sistema dando importanza alle feature principali
- **Alpha Test**: condotti dagli sviluppatori confrontandosi con dati reali del cliente, come per esempio dataset specifici, al fine di per verificare se il sistema soddisfa le richieste
- **Beta Test**: condotti dai clienti sul prodotto quasi completo

Da qui nasce la nomenclatura per:

- **Versione alpha:** con difetti e parti mancanti
- **Versione beta:** usata dai clienti, ha tutte le feature ma con difetti, eventualmente segnalati dagli utilizzatori
- **Versione gold:** completa e funzionante

Sviluppo a cascata

Esistono vari processi di sviluppo software, il primo è quello a **cascata** dove non si torna mai indietro. Ogni fase può durare molto, ma **non si possono fare correzioni** durante il processo di sviluppo. Quindi già da subito ci si focalizza sul prodotto completo, il vantaggio è che ci sarà **molta documentazione** ma vi sono **lunghi tempi di sviluppo**, **scarsa interazione con il cliente** se non solo all'inizio del progetto e vi è enorme difficoltà a introdurre cambiamenti suggeriti dal cliente. Questo tipo di sviluppo è **utile qualora le specifiche siano complete e stabili**. Il codice è tendenzialmente di alta qualità ed è utile a gestire sistemi complessi.

Sviluppi evolutivi e incrementali

Un'altra tipologia di sviluppo software è il sviluppo evolutivo, che ha due varianti:

- **Sviluppo per esplorazione:** gli sviluppatori lavorano con i clienti, che comprendono cosa va fatto man mano che il software viene scritto. Si parte con una serie di specifiche minimali, per poi andare ad incrementi. Il cliente ha la possibilità di dare subito dei feedback. Lo svantaggio è che non abbiamo una visione d'insieme del progetto, d'altra parte è l'unico modo di proseguire se non si conosce a priori cosa si vuole implementare precisamente.
- **Sviluppo Build and Fix:** simile allo sviluppo per esplorazione ma la documentazione è quasi inesistente dato che vi è una comprensione limitata del sistema. La fase di design è pressocchè inesistente e il codice prodotto è di bassa qualità. La fase di design è quasi inesistente. Succede quando il cliente non ha idea di cosa vuole che venga sviluppato. Dobbiamo quantomeno dare una linea di massima su quali sono le cose che il team di sviluppatori non può fare.

Possono nascere vari problemi, come l'allungamento dei tempi di sviluppo, nascita di sistemi difficilmente comprensibili e mancanza di visione d'insieme del progetto. Meglio applicarlo a sistemi di piccole dimensioni o singole parti di sistemi complessi.

Esistono altri tipi di processi di sviluppo software:

- **Processo di sviluppo incrementale:** simile allo sviluppo evolutivo, si implementano le funzionalità base con maggior priorità. Al codice sviluppato si aggiungono altri componenti e si itera in questo modo fino a completamento.

- **Processo di sviluppo Components Off The Shelf:** ovvero basato su componenti già esistenti. Lo sviluppo di nuovi componenti avrà come focus l'integrazione con i vecchi componenti. In genere viene impiegata in aziende di grandi dimensioni fondate molti anni fa (IBM, Microsoft...).
- **A spirale:** si focalizza su tanti prodotti parziali, graficamente rappresentata come una spirale. Ogni ciclo della spirale individua una fase del progetto, ogni ciclo si divide in 4 quadranti dove ognuna definisce un'attività da svolgere, queste sono:
    - **Identificazione obiettivi fase corrente**
    - **Valutazione rischi del progetto**
    - **Produzione e convalida**
    - **Revisione progetto, pianificazione prossima fase**

  Ogni ciclo potrebbe durare molti mesi.

Sviluppo Agile

Tra i principi fondanti dello **sviluppo Agile** vi è l'auto-organizzazione, collaborazione e comunicazione tra membri del team. Vi è un occhio di riguardo verso gli individui, meno documentazione, rivolgersi alle persone piuttosto. Si parla di collaborazione con il cliente, e non negoziazione. I cambiamenti sono sempre ben accetti se apportano migliorie al progetto.

L'**agilità** si rivela nel considerare positive le richieste di cambiamento anche in fasi avanzate, fornire release software frequenti. In presenza di persone esperte questo funziona perfettamente, quando vi sono persone alle prime armi funziona meno.

Extreme Programming (XP)

Consiste nello sviluppo e nella consegna di piccoli incrementi di funzionalità. Porta all'estremo le buone pratiche dello sviluppo software.

- Lo sviluppo degli incrementi ha durata di circa **2 settimane**
- I team di sviluppatori sono ristretti, **massimo 12 persone**
- **Costante miglioramento del codice esistente**
- Enfasi sulla **comunicazione** tra persone
- Poca documentazione, solo **Story Card** e **Class Responsibility Collabor**
- **Coinvolgimento di clienti e manager**
- **Prodotti testati sin dall'inizio**

XP si applica bene qualora vi siano progetti con requisiti non stabili dato che è estremamente adattivo. Vi sono sempre tempi di consegna stringenti.

Sono 12 le pratiche XP:

1. **Gioco di pianificazione:** mira a individuare gli obiettivi delle prossime due settimane. Ci si siede attorno un tavolo insieme al team, si ragiona su cosa va fatto, e consiste nella scrittura delle Story Card, piccoli documenti che raccolgono i requisiti mediante poche frasi. La dimensione ridotta obbliga i clienti alla sintesi, dato che sono scritte dal cliente. Si possono

associare priorità e stima dei tempi di sviluppo. Dobbiamo scegliere le Story Card da risolvere entro le prossime due settimane. Ogni Story Card viene divisa in task.

2. **Piccole release**: permette di avere feedback frequenti e un senso di realizzazione. Vedendo continue release cliente si fiderà di più. Possibilità di introdurre molti cambi.
3. **Metafore:** il cliente deve poter seguire le discussioni anche su temi più tecnici per poterlo mettere a suo agio e migliorare la comunicazione.
4. **Testing:** si testa tutto ciò che potrebbe andare male, si eseguono più volte al giorno non appena si rilascia nuovo codice. I test in realtà sono la specifica dei requisiti in formato eseguibile. Esistono due tipi di test:
    - Test funzionali: scritti dall'utente dal suo punto di vista. Consistono in una parte della specifica dei requisiti.
    - Unit test: scritti dagli sviluppatori, prima e dopo la codifica.
5. **Refactoring:** ogni qual volta si incontra del codice già esistente e migliorabile dobbiamo sempre cercare di migliorarlo. Meglio se si fa refactoring in coppia.
6. **Pair Programming**: si programma in coppia, uno dei due usa mouse e tastiera per pensare a scrivere codice (driver) mentre l'altro si pone domande sul codice che viene scritto, pensa ai test, pensa alla semplicità (navigator). Spesso ci si scambia in una singola sessione. Permette di spargere la conoscenza del sistema su più menti, evitando di avere situazioni in cui una singola persona conosce il funzionamento di un determinato componente.
7. **Cliente in sede:** necessario per avere un dialogo costante e per scrivere i test funzionali.
8. **Design semplice:** il giusto design si ha quando non ha parti duplicati, passa i test, poche classi e metodi (seguire i design pattern). Per documentare le classi si usano le CRC (Class Responsibility Collaboration) per avere una miglior visione d'insieme del progetto.
9. **Codice posseduto da tutto il team:** chiunque trovi un problema può risolverlo poichè ciascuno è responsabile del sistema.
10. **Integrazioni continua**: il codice testato viene integrato ogni poche ore, se e solo se tutti gli unit test vengono superati. Se un test fallisce chi ha prodotto l'errore deve risolverlo. Se non può farlo cancella il codice prodotto e ricomincia.
11. **Settimana da 40 ore:** se costantemente si supera questa soglia significa che qualcosa non va.
12. **Usare standard per i codici:** rendere il codice universalmente comprensibile è fondamentale per produrre codice di qualità. Singola responsabilità, codice semplice, stesse convenzioni per tutti.

SCRUM

Scrum è una struttura di base per lo sviluppo software per aiutare le persone a generare soluzioni per problemi complessi. L'ambiente si compone di:

- **Product Owner:** elenca il lavoro da svolgere mediante un **Product Backlog** (elenco dei lavori arretrati)
- **Scrum Team:** ovvero gli sviluppatori, che ha l'obiettivo di trasformare parte di lavoro in un incremento di risultati durante un'iterazione detta **Sprint**. Il team valuta, col supporto degli stakeholder (interessati al prodotto), risultati ed eventuali aggiustamenti per il successivo Sprint.

SCRUM applica un approccio iterativo/incrementali, e consiste in una serie di linee guida volutamente semplici (facili da seguire) e incomplete (ogni team lo adatta alle proprie esigenze). Si basa su due principi:

- **Empirismo:** le decisioni si basano sempre sulla conoscenza, che a sua volta si basa sull'esperienza.
- **Lean (Agile):** concentrarsi sulle cose essenziali per ridurre gli sprechi

Sono tre i pilastri SCRUM:

1. **Trasparenza:** il lavoro deve essere visibile sia a chi lo svolge sia agli stakeholders. Le decisioni importanti si basano su tre artefatti e questi sono il Product backlog, Sprint Backlog, Incremento. Avere trasparenza consente di avere una miglior ispezione del lavoro effettuato.
2. **Ispezione:** gli artefatti e il lavoro svolto vanno ispezionati frequentemente per scovare eventuali problematiche. Scrum fornisce la cadenza delle ispezioni mediante 5 eventi. Fare ispezioni permette di eseguire adattamenti.
3. **Adattamento:** se aspetti del processo di miglioramento diventano inaccettabili allora deve essere adattato. L'adattamento si attua subito per evitare ulteriori complicazioni.

Come abbiamo accennato gli eventi SCRUM sono cinque:

1. **Sprint:** hanno lunghezza fissa, variano da un paio di settimane a un mese.
2. **Sprint Planning:** per pianificare il lavoro da svolgere nello Sprint.
3. **Daily Scrum:** per ispezionare il progresso ed adattre lo Sprint Backlog (lavoro arretrato).
4. **Sprint Review:** per ispezionare i risultati dello sprint e determinare gli adattamenti.
5. **Sprint Retrospective:** per ispezionare l'ultimo Sprint riguardo gli individui e la Definition Of Done, ovvero una descrizione dello stato dell'incremento quando soddisfa le misure di qualità richieste.

Evoluzioni e metriche

La **manuntenzione** del codice è il processo di introduzione di modifiche dopo la sua consegna. In generale i costi di manutenzione sono molto alti, che possono arrivare fino all'80% dei costi totali.

Esistono cambiamenti di 4 tipi:

- **Correttivi per rimuovere errori**
- **Adattivi per adattare il sistema a un nuovo ambiente**
- **Perfettivi per migliorare o aggiungere funzionalità**
- **Preventivi per prevenire problemi futuri**

Leggi di Lehman

- **Cambiamento continuo:** i sistemi hanno bisogno di essere continuamente adattati altrimenti diventano meno soddisfacenti progressivamente
- **Aumento della complessità:** un sistema che evolve aumenta la sua complessità, a meno che il lavoro non sia volto a preservare o semplificare la struttura del codice

- **Auto-regolazione:** dimensioni, intervallo tra release e numero di errori di ciascuna release sono approssimativamente gli stessi
- **Stabilità organizzativa:** durante la vita di un sistema software il suo tasso di sviluppo è costante indipendentemente dalle risorse impiegate
- **Conservazione di familiarità:** mediamente l'incremento di crescita di un sistema software tende a rimanere costante o a diminuire
- **Continua crescita:** per mantenere la soddisfazione dell'utente il contenuto di un sistema deve essere continuamente incrementato
- **Diminuzione della qualità:** la qualità di un sistema diminuisce se non viene gestita a dovere

In generale sono leggi che valgono per sistemi grandi. Non è chiaro come funzionino per sistemi piccoli. Per controllare i costi di manutenzione il bisogna coinvolgere l'intero team di sviluppo, anche se tendenzialmente gli sviluppatori potrebbero non avere obblighi contrattuali per la manutenzione e quindi è naturale che la qualità di degradi nel tempo.

Il **refactoring** è il processo di cambiamento del software che non altera il suo comportamento ma ne migliora la struttura. L'attività di **reverse-engineering** consiste nell'analisi di un sistema per estrarre informazioni sul suo comportamento o struttura. L'attività di **re-engineering** consiste nell'alterare un sistema per ricostruirlo con una struttura differente.

Metriche

Servono a monitorare il prodotto mentre lo si costrituisce. La **complessità ciclomatica** indica il numero di percorsi indipendenti che possono essere intrapresi per valutare la complessità di un algoritmo. Il numero della complessità ciclomatica cc corrisponde di fatti al numero di test necessari per rendere sicuro il codice: per una sequenza basta un solo test, nel caso di condizioni se ne fanno due, nel caso di cicli $cc = archi - nodi + 2$.

La **dimensione** di calcola banalmente in linee di codice (LOC), oppure un NCNB per escludere appunto spazi e commenti, dato che di fatto verranno ignorati dal compilatore. In generale è consigliato avere una percentuale di commenti pari al 30% delle LOC così da rendere il codice più comprensibile.

Le **metriche di Chidamber&Kemerer (metriche CK)** sono metriche per sistemi ad oggetti per valutarne la complessità:

- **Weighted Methods per Class**: somma delle complessità dei metodi per una classe. Metodi di pari complessità hanno WMC pari al numero dei metodi della classe. Sono da preferire valori bassi perchè aumentano il lavoro per la manutenzione e diminuisce il riuso.
- **Depth of Inheritance Tree:** massimo numero di livelli dalla classe alla radice della gerarchia, con radice posta al livello zero. Più questo numero è alto e più è difficile da comprendere il comportamento delle classi, ma indica anche maggiore riuso.
- **Number of Children of a Class**: numero di sottoclassi per la classe. Maggiore è NOC maggiore è il riuso ma maggiore è anche l'influenza che ha quella classe sul resto del sistema.
- **Coupling Between Object Classes:** numero di classi con cui la classe interagisce. Più è alto e maggiore è la dipendenza della classe da altre classi, quindi minore possibilità di riuso e comprensione.
- **Response for a Class:** numero di metodi eseguiti al ricevimento di un messaggio. Questo è sinonimo di alta complessità quindi difficoltà di comprensione e testing.

- **Lack of Cohesion of Methods:** valori bassi indicano elevata possibilità di riuso, semplicità e single responsability principle.

Testing

Sappiamo che il software ha dei difetti tramite le specifiche, ovvero la lista delle cose che il software dovrebbe fare. Se il software ha comportamenti anomali allora significa che vi sono dei difetti e non vanno ignorati.

La fase di **Verifica&Validazione** assicura che il software soddisfi i bisogni degli utenti:

- **Verifica**: si riferisce a come il software è implementato. Solitamente sono white-box
- **Validazione**: si riferisce a porsi le domande sul prodotto, se è quello giusto per le reali richieste dell'utente. In genere questi sono black-box

I difetti che possono essere scovati sono di vario tipo:

- **Difetti di specifiche:** le specifiche sono ambigue o imprecise
- **Difetti di progettazione:** le componenti sono progettati in modo non corretto
- **Difetti di codice:** errori derivati dall'implementazione

Testare il software permette di rivelare facilmente la presenza di errori, ma non la loro assenza. Un test ha successo se scopre degli errori. Il debugging si riferisce alla pratica di localizzazione degli errori, si divide in ipotesi di dove sono localizzati e verifica di tale ipotesi. Si parte da dei dati in input per testare il sistema, potrebbero essere oltre che semplici dati anche file, eccesioni, stati del sistema. Da questi si passa a stimare gli output attesi basandoci sul Test Case. Un insieme di Test Case corrisponde a un Test Suite.

Un **test esaustivo** mostra se il programma è totalmente privo di difetti, ma non sono praticabili nella realtà. Le priorità dei test sono mostrare le capacità del software, testare le vecchie funzionalità con maggior priorità rispetto alle nuove e testare le situazioni tipiche è più importante rispetto a testare le situazioni limite.

Fare test è il processo che esercita un componente usando un set di test case per rivelare i difetti e valutarne la qualità. Ogni test deve contenere i risultati attesi per avere un metro di paragone, oltre che a contemplare data input corretti e non corretti. I test devono essere **ripetibili** e **riusabili**, questo si chiama **test di regressione**, ovvero una tipologia di test volta a verificare il funzionamento del software a seguito di nuove implementazioni o risoluzione di bug.

Un approccio dove i test vengono effettuati senza conoscere come è fatto il software è detto **test black-box**: i test case vengono progettati sulla base delle specifiche, è possibile predisporli nelle fasi iniziali dello sviluppo del software (**TDD**). Dall'insieme dei dati in input si individuano per ogni test il sottoinsieme di dati che caratterizzerà il test case. Per suddividere i vari test case possiamo farlo partizionando i dati in input in classi equivalenti, e per ogni classe scrivere un test così da permettere di coprire un grande dominio di test con pochi test. Ogni **partizione di equivalenza** consiste in data set che si prevede abbia comportamento uniforme, per ogni partizione si fa un test con dei valori che siano tipicamente agli estremi poichè i problemi capitano spesso agli estremi delle partizioni. Il **Fault Model** si verifica quando interi intervalli di valori delle partizioni di equivalenza non vengono elaborati correttamente.

Un altro approccio è quello **white-box** (glass-box o strutturale) che si focalizza sulla struttura del software da testare quindi necessita di avere il codice sorgente. Entrambi gli approcci hanno pro e contro.

I **test del percorso** servono per assicurare che i casi di test siano tali che **ogni percorso all'interno del programma sia eseguito** nei test almeno una volta. Definiti i nodi come le condizioni e gli archi come il flusso di controllo che può intraprendere il programma il numero di test da scrivere è pari alla complessità ciclomatica $archi - nodi + 2$, in questo modo abbiamo la certezza che tutti i percorsi sono eseguiti ma non è detto che lo siano tutte le combinazioni.

Fare **test sotto stress** consiste nell'**eseguire il sistema oltre il massimo carico previst**o, quindi testare i casi limite e vedere come si comporta il sistema, che dovrebbe fallire ma non in maniera catastrofica. Si indaga quindi sul livello dei disservizi del sistema. Questi test sono particolarmente importanti nel caso dei sistemi distrubuiti poichè la portata del sistema dipende anche dalla rete internet.

La **code coverage** è una misura in percentuale della parte del codice sorgente eseguita quando viene eseguito un test suite. In generale avere una code coverage molto alta o pari al 100% non significa non avere difetti. In generale è bene che parti critiche del sistema abbiano coperture maggiori di quelle meno critiche.

Un **test combinatorio** è una tecnica che mira a esplorare tutte le possibili combinazioni dei parametri in input in modo efficiente. Per farlo bisogna identificare e definire bene le classi di equivalenza. In molti casi però **è impossibile** poichè si richiederebbero eccessive risorse (problema NP-Completo). L'approccio migliore è quello **strategico**, quindi applicare un insieme di test case per ricoprire ampie combinazioni rappresentative.

Il **bug trend** misura della frequenza con cui vengono individuati i bug.

Laboratorio di Java

Java è un linguaggio imperativo, ma da Java 8 include caratteristiche della **programmazione funzionale**, che è più concisa, espressiva e facile da parallelizzare rispetto alla programmazione ad oggetti. Abbiamo l'interfaccia List<> che è più generico, permettendoci di istanziare vari tipi di Liste come ArrayList e LinkedList. Una volta definito il tipo della lista, come per esempio String, possiamo evitare di riscriverlo nuovamente poichè l'inferenza viene lasciata al compilatore (Java 7).

```
List<String> names = new ArrayList<>();
List<String> names = new LinkedList<>()
```

Espressioni lambda

Un'**espressione lambda** è una ***funzione anonima*** che prende in ingresso parametri indicati a sinistra della freccia e un corpo a destra della freccia e valori di ritorno. I tipi non devono essere specificati, viene fatto dal compilatore tramite inferenza.

Nei linguaggi funzionali un'espressione lambda è una funzione pura cioè il cui risultato dipende solo dagli input, mentre in Java (che è imperativo ma con caratteristiche della programmazione funzionale) le espressioni lambda possono avere uno stato interno e quindi non dipendere solo dall'input.

```
s -> System.out.println("Ciao "+s);
(x,y) -> x+y;   //Questo valore viene tornato
() -> System.out.println("Buongiornissimo");
(x,y) -> { System.out.println("x: "+x); return x+y; } // Devo specificare return
```

Una classe anonima è una classe senza un nome assegnato, quindi definita e instanziata mediante una versione estesa dell'operatore new. Possiamo usarle per implementare le interfacce. Per esempio:

```
public interface Hello {
    public void greetings(String s);
}
```

```
public class Sera {
    private Hello myh;

    public Sera(Hello h) {
        myh = h;
    }

    public void saluti() {
        myh.greetings("buonasera");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Sera sr = new Sera(new Hello() {
            public void greetings(String s) {
                System.out.println("Ciao, "+s);
            }
        });
    }
}
```

La classe anonima implementa l'interfaccia Hello direttamente nel punto in cui si istanzia. Ma grazie alle espressioni lambda possiamo accorciare moltissimo quel codice, è possibile farlo quando l'interfaccia ha un singolo metodo.

```
Sera sr = new Sera(s2 -> System.out.println("Ciao, " + s2));
```

Quell'espressione lambda sarà l'implementazione dell'unico metodo dichiarato in Hello. Il compilatore capirà per inferenza che s2 è di tipo String poichè così è il parametro in ingresso di Hello.

Data una lista del tipo:

```
private List<String> listaNomi = Arrays.asList("Nobita", "Nobi", "Shizuka");
```

Possiamo cercare specifici valori in due modi, imperativo e dichiarativo.

```java
public void trovaImperativo() {
    boolean trovato = false;

    for (String nome : listaNomi)
        if (nome.equals("Nobi")) {
        trovato = true;
        break;
    }

    if (trovato) System.out.println("Nobi trovato");
    else System.out.println("Nobi non trovato");
}
```

```java
public void trovaDichiarativo() {
    if (listaNomi.contains("Nobi"))
        System.out.println("Nobi trovato");
    else
        System.out.println("Nobi non trovato");
}
```

L'implementazione imperativa rende il codice molto più lungo. Questo potrebbe rendere più difficoltoso comprendere cosa il codice sta facendo a seguito di una rapida lettura. L'uso dello stile dichiarativo rende il codice più autoesplicativo, nonostante comunque dei cicli all'interno della funzione `contains()` vengano svolti.

La programmazione funzionale è dichiarativa. In Java 8 possiamo passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi. Ricordiamo che la differenza tra metodi e funzioni è che i primi appartengono ad una classe, mentre le funzioni no. Metodi o funzioni che ricevono, creano o ritornano funzioni sono dette funzioni di ordine superiore.

Nella programmazione funzionale, in aggiunta allo stile dichiarativo, si aggiunge il concetto di **funzioni di ordine più alto**, ovvero funzioni che hanno come parametri delle funzioni. In Java 8 possiamo passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi. Chiaramente un metodo è una funzione associata a una classe mentre una funzione in sè non vive all'interno di una classe. Metodi o funzioni che ricevono, creano o ritornano funzioni sono di ordine superiore.

Collection

Collection è un'interfaccia che definisce i metodi astratti come `add()`, `remove()`, `size()`, `contains()`, `containsAll()` e molti altri.

	get	add	contains	remove
ArrayList	costante	costante	O(n)	O(n)
LinkedList	O(n)	costante	O(n)	costante
TreeSet		O(lg n)	O(lg n)	O(lg n)

ArrayList è un array espandibile. Gli elementi sono contigui e l'accesso al generico elemento avviene in tempo costante. LinkedList è una lista e ogni elemento ha i riferimenti al successivo e al precedente. Vector è simile ad ArrayList ma è sincronizzato nel senso che nel momento in cui ci sono più thread che provano ad accedere agli elementi esso consente l'accesso uno alla volta,

questa cosa ha un suo costo computazionale quindi se non serve usiamo ArrayList. Abbiamo anche TreeSet.

La complessità dipende dal tipo di struttura dati utilizzata. Per esempio ArrayList essendo un array espandibile ha tempi d'accesso al singolo elemento costanti. Qualora non vi sia spazio nel blocco di memoria scelto cresce del 50%. LinkedList è una lista e ogni elemento ha un riferimento al successivo e al precedente, invece Vector è simile ad ArrayList ma è synchronized (supporta l'accesso agli elementi da parte di più thread) ma è legacy.

Stream

L'interfaccia Collection dichiara un metodo `stream()` che restituisce uno Stream<T> ovvero una sequenza di elementi di tipo T. Questi possono essere manipolati grazie a metodi di supporto. Le operazioni che restituiscono altri Stream sono intermedie, altrimenti sono terminali.

Le operazioni **stateless** non hanno uno stato interno, dipendono solo dall'input. Le operazioni **stateful** per dare un output devono conoscere anche gli altri elementi dello stream e non solo l'input attuale. Un esempio sono i metodi `reduce()` e `max()`.

- `of()` metodo statico di List, ritorna una **lista non modificabile** contenente gli elementi passati

  ```
  LinkedList<String> lista = List.of("Nobita", "Doraemon", "Suneo");
  ```

- `filter(Predicate<T> p)` il predicato è una condizione booleana sugli elementi dello Stream. La funzione filter ritorna uno Stream contenente gli elementi che soddisfano il predicato.
  Predicate è un'interfaccia funzionale che definisce il metodo `test(Object o)`, prende in ingresso un Object e restituisce un boolean.

  ```
  Predicate<Integer> positive = x -> x>0;
  ```

  Le interfacce funzionali si indicano con `@FunctionalInterface`

- `count()` conta gli elementi di uno stream, è terminale.

  ```
  List<int> numeri = List.of(20,10,30,50,40);
  long contaMaggioriTrenta = numeri.stream()
                                      .filter(n -> n>30)
                                      .count();
  ```

- `count()` conta gli elementi di uno stream, è terminale.
- `of()` metodo statico di Stream, ritorna uno Stream con gli elementi passati.

  ```
  Stream<String> nomi = Stream.of("Nobita", "Doraemon", "Suneo");
  ```

- `reduce()` è un metodo di Stream, prende in ingresso un valore dello stesso tipo dello stream (valore iniziale) e un'espresione lambda con ingresso due valori e ritorno un valore. Reduce si usa quando si vuole passare da un insieme di valori a un singolo valore.

```
lista.stream().reduce(0, (accum, v) -> accum + v);
lista.stream().reduce(0, Integer::sum) // Stessa cosa
```

- `map()` è un metodo di Stream che prende in ingresso una funzione mapper e restituisce uno stream con i risultati dell'esecuzione della funzione mapper su ciascun elemento dello stream iniziale e ogni risultato viene inserito in un nuovo stream.

```
List<Integer> lista = List.of(3,7,10);
lista.stream().map(x -> x*2)        // nuovo stream contiene 6,14,20
```

- `collect()` è un metodo di Stream che prende in ingresso un Collector, che implementano metodi per trasformare uno stream in raggruppamenti. Nonostante ciò, da Java 16 abbiamo il metodo Stream.toList() come alternativa più semplice.

```
List<int> numeriList = numeri.stream()
                                    .collect(Collectors.toList());
// OPPURE
List<int> numeriList = numeri.stream()
                                    .toList();
```

- `Comparator.comparing()` prende in ingresso una funzione per estrarre una chiave per poterla confrontare col resto dello stream. Utile nella funzione max, sorted.
- `max()` restituisce un Optional contenente un elemento massimo basandosi su un criterio passato in input chiamato Comparator. Valuterà gli elementi dello stream.

```
Optional<Persona> maxNumber = numeri.stream()
                                  .max(Comparator.comparing(Persona::getEta));
```

- `sorted()` operazione intermedia stateful che restituisce uno stream con elementi ordinati in base al Comparator passato
- `distinct()` operazione intermedia stateful che restituisce uno Stream con elementi distinti
- `forEach()` operazione terminale che esegue un'azione su ciascun elemento dello Stream

```
chiavi.stream()
        .map(Chiave::getValore)
        .distinct()
        .sorted(Comparator.comparing(Chiavi::getValore))
        .toList;
```

Possiamo generare Stream tramite:

- `iterate()` restituisce uno stream infinito a seguito dell'esecuzione di una funzione seme. Usare limit per troncare i risultati. La funzione viene applicata ad ogni valore prodotto per ottenere il successivo.

```
Stream.iterate(2, n -> n*2)
        .limit(4)
        .forEach(System.out::println)
```

```
OUTPUT:
    2   4   8   16
```

- `generate()` restituisce uno stream infinito di valori tramite una funzione Supplier. Non viene applicata ad ogni valore prodotto.

```
Stream.generate(()->Math.round(Math.random()*10))
        .limit(5)
        .forEach(System.out::println);
```

```
OUTPUT:
    2   5   1   9   5
```

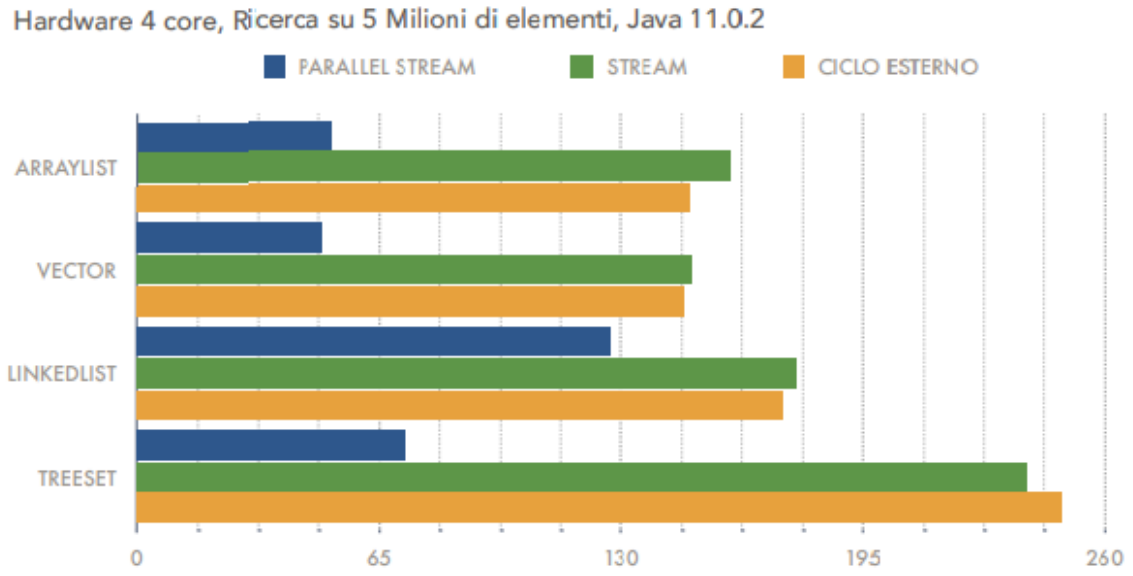
Se volessimo debuggare quindi vedere cosa fa ogni operazione intermedia non possiamo scorrere la lista con `forEach()` perchè questo metodo non restituisce altri stream. Quindi è bene usare il metodo `peek()`.

Programmazione parallela

Dato che i processori hanno vari core sarebbe bene utilizzarli a dovere e va fatto con la programmazione parallela, che è più complessa della sequenziale. Possiamo farlo in Java mediante la classe Thread per lanciarne di nuovi ma bisogna gestire la concorrenza. Dato che `map()` e `filter()` sono stateless e generano un stream distinti è molto più semplice la parallelizzazione dei threads. *Nota bene che se l'applicazione nel frattempo aggiorna uno stato globale l'operazione diventa stateless*.

Collection implementa anche il metodo `parallelStream()`, che **potrebbe** dare degli stream in parallelo senza avere bisogno dei thread. Le prestazioni dipendono dal numero di elementi nello stream, le operazioni da svolgere e su quale struttura dati lavoriamo (ArrayList e LinkedList per esempio potrebbero avere complessità differenti rispetto alle medesime operazioni), oltre che l'hardware utilizzato. In generale l'uso di stream paralleli è vantaggioso con ArrayList e TreeSet, mentre con LinkedList a causa dell'accesso sequenziale non si hanno vantaggi enormi.

```
long c = nomi.parallelStream()
             .map(String::toUpperCase)
             .filter(s -> s.equals("NOBI"))
             .count();
```



Data una lista di oggetti Persona trovare il nome di una che ha il ruolo Programmer

```
Optional<Persona> r = team.stream()
                          .filter(p -> p.isRuolo("Programmer"))
                          .findAny();

if (r.isPresent()) System.out.println(r.get().getNome());
```

- `findAny()` è terminale e restituisce un Optional, per valutarla non è necessario analizzare tutto lo stream. Operazioni del genere, ovvero che rendono possibile la non esecuzione di parte di stream si dicono short-circuiting

Le operazioni `map()` e `filter()` sono stateless, ovvero non hanno uno stato interno, prendono un elemento dello stream e danno zero o un risultato. Invece `reduce()` e `max()` per esempio accumulano un risultato quindi non sono stateless. Operazioni come `sorted()` e `distinct()` devono conoscere gli altri elementi dello stream e sono dette stateful.

IntStream

Consiste in uno stream di valori int. Ha dei metodi diversi.

- `rangeClosed()` restituisce una sequenza di int nell'intervallo specificato con estremi inclusi e incremento pari a 1
- `sum()` somma i valori presenti nello stream, restituisce un int

```
int somma = IntStream.rangeClosed(1,3).sum();    // somma = 1+2+3 = 6
```

- `mapToInt()` applicata a uno stream esegue la funzione passata e restituisce un IntStream

```
int result = Stream.of("ciao", "sono", "simone")
                   .mapToInt(x->x.length())
                   .sum();
```

- `boxed()` restituisce uno stream di Integer partendo da un IntStream

```
Stream<Integer> s = IntStream.rangeClosed(1,10)
                                .boxed();
```

- `mapToObj()` restituisce uno stream di oggetti a partire da IntStream

```
IntStream.rangeClosed(0,3)
        .mapToObj(i -> lista.get(i));
```

Interfaccia Function

L'interfaccia funzionale `Function<T,R>` definisce un metodo chiamato `apply()` che prende in ingresso un oggetto di tipo generico T e ritorna un tipo generico R. Per esempio:

```
Function<String, Integer> stringLength = x -> x.length();
```

il metodo `apply()` sta prendendo in ingresso un parametro di tipo String, che è x, e il corpo è `x.length()`. Restituisce il valore dato da lenght, quindi un integer.

```
int result = Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
                    .map(stringLength)
                    .reduce(0, Integer::sum);

System.out.println(result);
```

```
OUTPUT:
    31
```

In questo esempio map prende in input la funzione stringLength definita sopra per calcolare la lunghezza di ogni parola e quindi `reduce()` per sommarle partendo da 0. Sottolineiamo il fatto che non dobbiamo vedere stringLength come se fosse un valore ma una vera e propria funzione, che quindi definisce un comportamento che può essere passato a `map()`.

Interfaccia Supplier

Un Supplier<T> è un'interfaccia funzionale che definisce un metodo `get()`, e rappresenta una funzione che prende in ingresso nessun parametro e restituisce un valore di tipo T.

```
Supplier<String> sup = () -> "Ciao ciao";
String s = sup.get();
```

```
OUTPUT:
    Ciao ciao
```

In questo caso quindi definisce una funzione che restituisce qualcosa quando su di essa viene chiamato il metodo get.

Vediamo come potremmo utilizzarlo col Factory Method. Ipotizziamo di implementarlo banalmente in questo modo:

```
public class Creator {
    public static Prodotto getProdotto(String name) {
        switch (name) {
            case "primo": return new ProdottoA();
            case "secondo": return new ProdottoB();
            case "terzo": return new ProdottoC();
            case "quarto": return new ProdottoD();
            default: return new ProdottoA();
        }
    }
}
```

```
Prodotto p1 = Creator.getProdotto("primo");
```

Se usassimo un Supplier definito come

```
Supplier<Prodotto> prodSupplier = ProdottoA::new;
```

La linea di codice per creare p1 sarebbe tradotta in questo modo

```
Prodotto p1 = prodSupplier.get();
```

Quindi possiamo creare una mappa per far corrispondere al nome di un prodotto la sua creazione.

```
Map<String, Supplier<Prodotto>> map = Map.of("primo", ProdottoA::new, "secondo",
ProdottoB::new, "terzo", ProdottoC::new);
```

In questo modo abbiamo tradotto parte di switch. Adesso usiamo questa mappa per istanziare i prodotti:

```
public static Prodotto getProdotto(String name) {
     Supplier<Prodotto> s = map.get(name);
     if (s != null) return s.get();
     return new ProdottoA(); // Default
}
```

Qualora il costruttore avesse più parametri in ingresso non possiamo usare Supplier e dobbiamo usare un'altra interfaccia funzionale.

Esercizi svolti sugli stream

Data una lista di Persona stampare e contare i nomi dei programmatori.

```
public void conta(String ruolo) {
     System.out.print("Hanno ruolo " + ruolo + ": ");

     long c = team.stream()
                     .filter(p -> p.getRuolo().equals(ruolo))
                     .peek(p -> System.out.print(p.getNome() + ", "))
                     .count();

     System.out.println("Ci sono " + c + " " + ruolo);
 }
```

```
conta("Programmer");
```

> Hanno ruolo Programmer: Simone, Giulia, Andrea
> Ci sono 3 Programmer

Data una lista di Persona stampare i ruoli presenti e per ciascun ruolo la lista delle persone aventi quel ruolo.

```
public void scriviRuoli() {
     team.stream()
         .map(p -> p.getRuolo())
         .distinct()
         .peek(r -> System.out.print("\nRuolo " + r + ": "))
         .forEach(r -> team.stream()
                           .filter(p -> p.getRuolo().equals(r))
                           .forEach(p -> System.out.print(p.getNome() + " ")));
}
```

`map` prende il ruolo, `distinct` li prende distinti, stampiamolo con `peek`. Dopo di che usiamo il `forEach` e ricreiamo un nuovo stream dove filtriamo per ruolo e stampiamo il nome con un nuovo `forEach`.

Data una lista di Persona creare una lista di Pagamento con costo calcolato in base a ciascuna persona e stampare i pagamenti

```
public void pagamenti() {
     pagati = team.stream()
                     .filter(p -> daPagare.contains(p.getNome()))
                     .map(p -> new Pagamento(p, p.getCosto() * 30))
                     .peek(v -> System.out.print(v.getPers().getNome() + " " +
v.getImporto() + " "))
                     .collect(Collectors.toList());
 }
```

Data una lista di stringhe produrre una lista che contiene solo le stringhe che cominciano con un certo prefisso.

Suggerimento: usare il metodo `substring(int beginIndex, int endIndex)` della classe String.

```
List<String> lista = new ArrayList<>();
lista.add("Simone");
lista.add("Ernesto");
lista.add("Sincero");

List<String> listaSi = lista.stream()
                            .filter(p -> p.substring(0,2).equals("Si"))
                            .collect(Collectors.toList());
System.out.println(listaSi);
```

["Simone", "Sincero"]

Data una lista di stringhe produrre una stringa contenente le iniziali di ciascuna stringa della lista

```
List<String> lista = new ArrayList<>();
lista.add("Ciao");
lista.add("Sono");
lista.add("Simone");

String stringaInit = lista.stream()
                          .map(x->x.substring(0,1))
                          .collect(Collectors.joining())
                          .toString();
System.out.println(stringaInit);
```

CSS

Data una lista di terne di numeri interi per ciascuna terna verificare se essa costituisce un triangolo. Restituire la lista dei perimetri per le terne che rappresentano triangoli.

Suggerimento: in un triangolo ciascun lato è minore della somma degli altri due.

```
List<int[]> terne = new ArrayList<>();
terne.add(new int[] {1,2,3});
terne.add(new int[] {3,2,3});
terne.add(new int[] {3,10,3});
terne.add(new int[] {3,2,4});
terne.add(new int[] {3,2,12});

terne.stream()
      .filter(terna -> terna[0] < (terna[1]+terna[2]))
      .filter(terna -> terna[1] < (terna[0]+terna[2]))
      .filter(terna -> terna[2] < (terna[1]+terna[0])
      .forEach(x -> System.out.print(x[0]+x[1]+x[2]+" "));
```

```
8 9
```

Data una lista di numeri interi verificare se ciascuna terna formata prendendo dalla lista tre numeri contigui costituisce un triangolo

Esempio: lista {2, 3, 5, 7, 8}, terne {2, 3, 5}, {3, 5, 7}, {5, 7, 8}

```
List<Integer> lista = new LinkedList<>();
lista.add(2);
lista.add(3);
lista.add(5);
lista.add(7);
lista.add(8);

IntStream.rangeClosed(0, lista.size()-3)
          .mapToObj(i -> new int[]{lista.get(i), lista.get(i+1), lista.get(i+2)})
          .filter(t -> t[0] < t[1]+t[2])
          .filter(t -> t[1] < t[0]+t[2])
          .filter(t -> t[2] < t[1]+t[0])
          .forEach(t -> System.out.println(t[0]+" "+t[1]+" "+t[2]));
```

```
3 5 7
5 7 8
```

Data una lista di numeri interi positivi verificare se la lista è ordinata.

```
List<Integer> lista = List.of(2,3,5,6,8);
boolean isSorted = !IntStream.rangeClosed(0,lista.size()-2)
                             .filter(i -> lista.get(i)>lista.get(i+1))
                             .findAny()
                             .isPresent();
System.out.println(isSorted);
```

true

Data una lista di istanze di Persona, stampare e contare i nomi dei programmatori

```
List<Persona> listaPersone = new LinkedList<>();
listaPersone.add(new Persona("Simone", "Programmatore"));
listaPersone.add(new Persona("Fabio", "Agricoltore"));
listaPersone.add(new Persona("Lorenzo", "Programmatore"));
listaPersone.add(new Persona("Giulia", "Ingegnere"));

long conta = listaPersone.stream()
                         .filter(x->x.getMansione().equals("Programmatore"))
                         .peek(Persona::printNome)
                         .count();

System.out.println("Numero programmatori: " + conta);
```

Simone
Lorenzo
Numero programmatori: 2

Data una lista di istanze di Persona, stampare i ruoli presenti e per ciascun ruolo la lista delle persone aventi quel ruolo

```
List<Persona> listaPersone = new LinkedList<>();
listaPersone.add(new Persona("Simone", "Programmatore"));
listaPersone.add(new Persona("Fabio", "Impiegato"));
listaPersone.add(new Persona("Lorenzo", "Programmatore"));
listaPersone.add(new Persona("Giulia", "Ingegnere"));


listaPersone.stream()
            .collect(Collectors.groupingBy(Persona::getMansione))
            .forEach((ruolo, personeRuolo) -> {
                      System.out.println("Ruolo: "+ruolo);
                      personeRuolo.forEach(persona ->System.out.println("
"+persona.getNome()));
            });
```

Ruolo: Ingegnere
Giulia
Ruolo: Programmatore
Simone
Lorenzo
Ruolo: Impiegato
Fabio

Data una lista di nomi di persona, creare la lista di istanze di Pagamento con il costo calcolato in base a ciascuna persona, e stampare i pagamenti

```
List<Pagamento> pagamenti = new LinkedList<>();
pagamenti.add(new Pagamento("Simone", 30));
pagamenti.add(new Pagamento("Simone", 50));
pagamenti.add(new Pagamento("Giulia", 30));
pagamenti.add(new Pagamento("Lorenzo", 70));

pagamenti.stream()
        .collect(Collectors.groupingBy(Pagamento::getIntestatario))
        .forEach((in, imp)->{
                    System.out.println("Intestatario: "+in);
                    System.out.println("Pagamenti: "+ imp.stream().map(x-
>x.getImporto()).collect(Collectors.toList()));
                    System.out.println("Totale: "+
IntStream.rangeClosed(0,imp.size()-1).mapToDouble(i-
>imp.get(i).getImporto()).sum());
 });
```

Intestatario: Simone
Pagamenti: [30, 50]
Totale: 80.0

Intestatario: Lorenzo
Pagamenti: [70]
Totale: 70.0

Intestatario: Giulia
Pagamenti: [30]
Totale: 30.0

Data una lista di istanze di Persona, creare una lista con istanze di BustaPaga con l'importo calcolato in base al costo di ciascuna persona, e ordinare la lista per nome persona

```
List<Persona> persone = new LinkedList<>();
persone.add(new Persona("Simone", 1400));
persone.add(new Persona("Giulia", 1900));
persone.add(new Persona("Lorenzo", 1600));
persone.add(new Persona("Chicca", 1700));


List<BustaPaga> bustePagaOrdinatePerNome = persone.stream()
    .map(persona -> new BustaPaga(persona, persona.getStipendio()))
    .sorted(Comparator.comparing(BustaPaga::getNomePersona))
    .collect(Collectors.toList());

// Stampa le buste paga ordinate per nome persona
bustePagaOrdinatePerNome.forEach(bustaPaga -> System.out.println("Nome:
"+bustaPaga.getNomePersona()+", Importo: " + bustaPaga.getImporto()));
```

Nome: Chicca, Importo: 1700
Nome: Giulia, Importo: 1900
Nome: Lorenzo, Importo: 1600
Nome: Simone, Importo: 1400

Data la lista di istanze di BustaPaga, stampare il nome di ciascuna persona e l'importo e calcolare la somma degli importi

```
List<BustaPaga> bustePaga = new LinkedList<>();
bustePaga.add(new BustaPaga("Simone", 1600));
bustePaga.add(new BustaPaga("Giulia", 1800));
bustePaga.add(new BustaPaga("Lorenzo", 2000));
bustePaga.add(new BustaPaga("Chicca", 1700));

int somma = bustePaga.stream()
    .peek(x-> System.out.println("Persona: "+x.getNome()+", Importo:
"+x.getImporto()))
    .mapToInt(x->x.getImporto())
    .sum();
System.out.println("Totale: "+somma);
```

Persona: Simone, Importo: 1600
Persona: Giulia, Importo: 1800
Persona: Lorenzo, Importo: 2000
Persona: Chicca, Importo: 1700
Totale: 7100