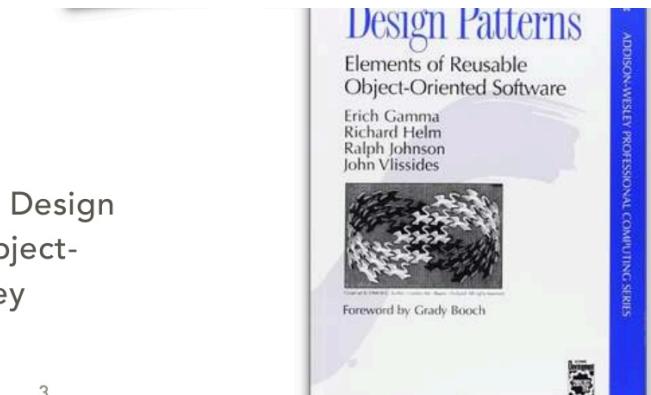


Libro: I. Sommerville. Ingegneria del Software. Pearson Addison-Wesley

Sito del prof: <https://www.dmi.unict.it/tramonta/>

- Libro di design pattern:
 - Fowler. UML Distilled. Pearson



3

- <https://www.dmi.unict.it/tramonta/se>
- <https://github.com/e-tramontana>

PROCESSI DI SVILUPPO

Descrizione della metodologia per sviluppare un software di grandi dimensioni (come, per es: *extreme programming*)

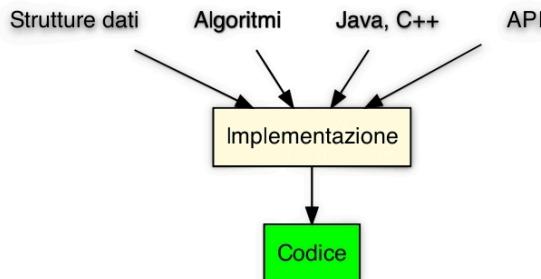
I processi definiscono varie fasi e il ritmo di lavoro è diverso a seconda del processo di sviluppo. Esse principalmente sono:

- **analisi**, specifiche dei requisiti: mira a comprendere cosa il software deve fare. Varia in base alle richieste del cliente;
- **progettazione**, consiste nel ricavare la struttura del software da realizzare, cioè l'insieme dei componenti che ha il software e le relazioni fra loro. Si individuano le singole classi (come componenti). Vene individuata anche una TRACCIA di quello che il software deve fare (non il codice di per sé) e una traccia dell'algoritmo da usare.
- **L'implementazione** produce codice funzionante per quella determinata progettazione documentata precedentemente.
- **Test**, o di convalida, si deve far eseguire il sistema realizzato con alcuni valori di esempio per vedere se produce quello che il programmatore si aspetta da tali input.
- **manutenzione**, modifica ulteriore del codice dopo che esso è stato consegnato al cliente. Si fanno delle modifiche dopo la consegna del software per continuare a soddisfare le esigenze del cliente che potrebbe volere altri requisiti successivamente.
 - ci possono essere modifiche successive in seguito a variazioni della legislazione in corso.

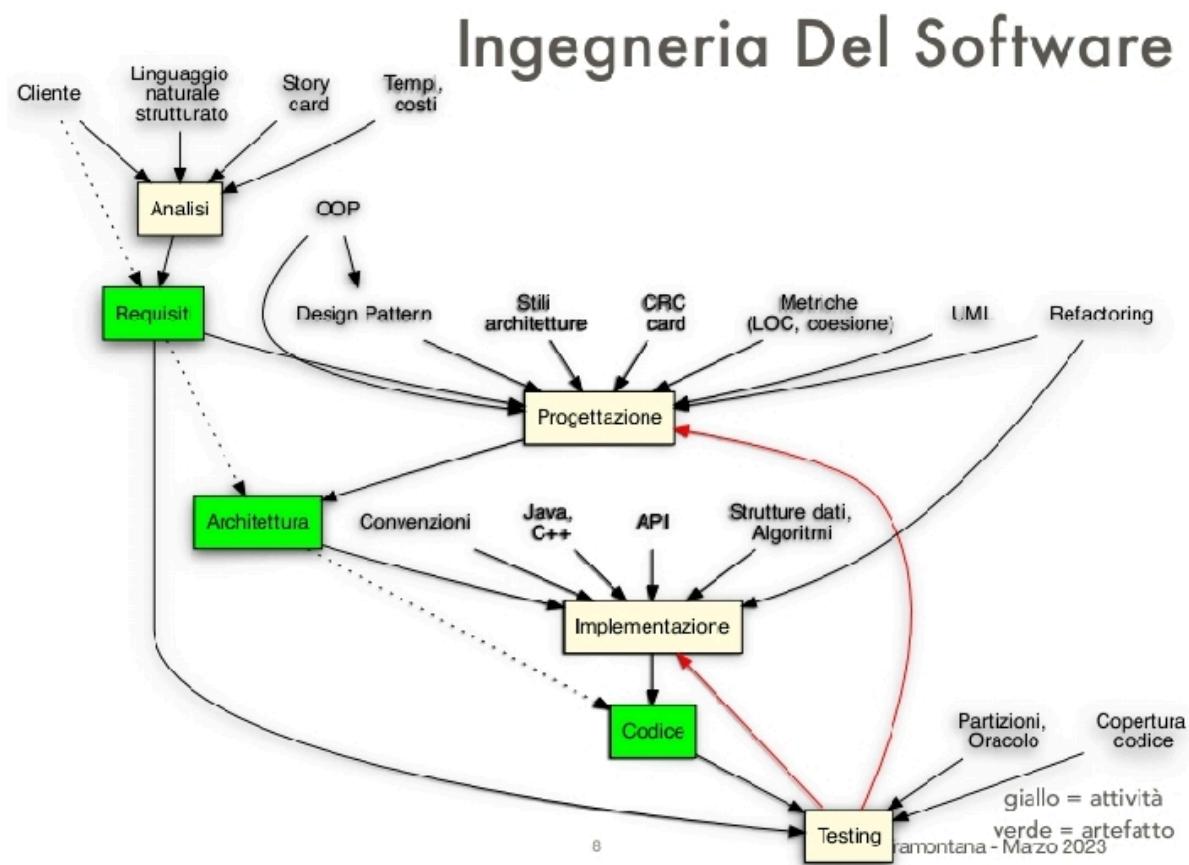
| *Ingegneria del software offre metodologie per sviluppare bene un software seguendo quelle fasi.*

Generalmente si pensa che per sviluppare un software si agisca in questo modo:

Sviluppo ...



In realtà lo sviluppo è così fatto:



Si deve poter **documentare** quello che si ascolta dal cliente, quindi servono le fasi di progettazione elencati prima.

Convenzioni: ci sono delle regole per la scrittura del codice. *Deve funzionare e deve essere scritto bene.*

Caratteristiche del software

- **Modificabilità**, si può modificare visto che non ha parti fisiche. I risultati di un oggetto fisico potrebbe creare danni irreversibili (modificare sportelli di un'automobile ecc..). Un software è prontamente modificabile in ogni momento. Si può aprire il codice sorgente o eseguibile e cambiare linee di

codice. I risultati della modifica potrebbero, comunque, creare dei problemi. Può risultare più utile senza errori o può continuare a funzionare con errori.

- *Solo il software non ha materia fisica. Un disco, un SSD serve SOLO per contenere il software e il suo codice. Quindi per questo ho bisogno di materia per memorizzare il software.*

La modificabilità permette di **intervenire** facilmente durante la fase di **manutenzione**.

Chi produce software fa del lavoro con un team e per poter fornire le nuove versioni al cliensi fa pagare, quindi è **un'opportunità di affari**.

Se il cliente si trova bene vuol dire che è una buona opportunità di guadagno perché il software verrà sempre usato.

- Un **software di successo** ha una **durata di vita** molto più **lunga** rispetto all'hardware. Si dovrà sempre adeguare ai nuovi hardware.

Qualità del software

Essa è valutata in base alla presenza o meno di alcune caratteristiche:

- **Correttezza**, se è corretto allora possiamo dire che è un sistema di qualità. Essa viene valutata in base alla aderenza o meno al suo obiettivo o scopo. Se il sistema software che viene eseguito con determinati input, da come risultato quello che ci si aspetta, allora esso è corretto. Altrimenti, se si ottengono risultati inaspettati, esso non è corretto. Quindi la correttezza si valuta **ESEGUENDO** il sistema software.
- *Come si fa a capire se la risposta attesa è corretta o meno?* Trovo la risposta nel documento prodotto durante le specifiche dei requisiti perché è lì che si trovano le informazioni riguardo la funzionalità del software. Se trovo un risultato (successivo a test) che non è presente nel documento dei requisiti allora vuol dire che è carente di informazioni e quindi è scritto male e in questo caso non si può dire se il software è di qualità o meno.

Questo discorso risponde alla domanda: "*Il sistema software soddisfa le specifiche che erano state raccolte?*"

Il sistema software corretto è quello che vuole il cliente? Sì a patto che il documento dei requisiti sia stato scritto bene. Altrimenti ci si ricavano informazioni non volute dal cliente.

- **Efficienza**: Se utilizza il minimo di risorse necessarie (troppa memoria, troppa RAM, troppo spazio su disco). essa si valuta guardando e valutando il codice
- **manutenibilità**, gli effetti di una piccola modifica sono piccoli e relative alla piccola cosa che si deve modificare. Se gli effetti di una piccola modifica si ripercuotono su tutto il sistema allora non è manutenibile. Si usano encapsulamento, information hiding e altre tecniche per ottenere questa caratteristica. *Si possono fare **piccole modifiche** al codice **senza sconvolgere tutto il sistema***.
- **dependability** (sicurezza e affidabilità), sicurezza -> **safety** e **security** e affidabilità -> **reliability**:
 - **security** = capacità di sistema software di preservare i dati che sono stati inseriti, cioè non mostrarli a chiunque quindi si intende la **protezione dei dati** visto che spesso ci sono dati sensibili.
 - **safety** = protezione del sistema hardware e software, durante l'esecuzione del software la vita delle persone e dell'hardware (surriscaldamento) sul quale gira vengono mantenuti. Si deve far sì che le operazioni devono essere eseguite senza mettere a rischio l'hardware. Se si danneggia allora può esserci rischio per le vite umane.

- **affidabilità**, capacità di un software di funzionare nel tempo. Meno guasti (difetti di funzionamento, come per esempio l'interruzione dell'applicazione durante il salvataggio) ci sono nel tempo allora più è affidabile.
- **usabilità****, *l'applicazione che dialoga con l'utente ha preso le giuste precauzioni per mettere in condizioni operative corrette l'utente? Cioè: come l'utente sta mostrando l'applicazione (quindi tramite interfaccia)*. L'utente può capire in che condizioni operative si trova?*. L'utente si trova dentro o fuori all'aria aperto? Bisogna quindi valutare l'usabilità dell'applicazione e adattare le esigenze del cliente in base alle sue condizioni operative

JAVA

E' un linguaggio ad oggetti puro quindi tutto quello che scrivo sta in una classe.

```

import java.time.LocalDate; //direttiva, package time preso dalla libreria java
/**
 * Classe che stampa sullo schermo un messaggio e la data corrente
 */
//questo tipo di commento serve per fare capire a che serve questa determinata classe
//visto che ho scritto questo. Ogni volta che scrivo helloworld mi compare un tooltip che mi
dice cosa fa la classe che sto scrivendo.

public class HelloWorld { // definizione classe. public= classe accessibile da qualunque
altra classe

    // dichiarazione e assegnazione campi
    private static final String msg = "Lezione di Ingegneria del Software";
    private static final LocalDate d = LocalDate.now();

    //private static final: private = visibilità dell'attributo; static = attributo della classe
    //e non per il singolo oggetto. Fra le varie istanze tale attributo vive sulla classe e non
    //sulla singola istanza. E' utile per contare il numero di istanze di un oggetto o operazioni
    //simili. final = costante

    /**
     * Metodo da cui inizia l'esecuzione del programma
     *
     * @param args parametri passati al metodo all'avvio della classe
     */
    //args serve per formattare bene i commenti nel tooltip così che si capisca bene il
    //tooltip.

    public static void main(String[] args) {
        System.out.println("Hello World");
        System.out.println(msg);
        System.out.println(d);
    }
}

Output:
Hello World
Lezione di Ingegneria del Software
2020-03-03

```

public: tutti possono accedere.

private: solo alcuni possono accedere alla classe.

protected: solo alcuni possono accedere alla classe.

Convenzioni sul codice:

- il nome della classe inizia con la **lettera maiuscola**
- evito underscore e uso *CameCase*
- Tutti i **tipi non primitivi** di variabili iniziano con la maiuscola. Se non sono oggetti allora iniziano con la minuscola.
- Organizzare i singoli file in modo tale che contengano una singola classe. Una classe -> un file.
Nome classe deve coincidere con il nome del file

```

import java.time.LocalDate;
// import indica dove trovare la classe LocalDate

/**
 * Classe che stampa sullo schermo un messaggio e la data corrente
 */
public class HelloWorld { // definizione classe HelloWorld
    // dichiarazione e assegnazione campi
    private static final String MSG = "Lezione di Ingegneria del Software";
    private LocalDate d;

    /**
     * Metodo da cui inizia l'esecuzione del programma
     *
     * @param args parametri passati al metodo all'avvio della classe
     *
     * @version 0.1
     */
    public static void main(String[] args) {
        final String NAME = "Marco";
        System.out.print("Hello "); // scrive su schermo
        System.out.println(NAME);
        System.out.println(MSG);

        HelloWorld world = new HelloWorld(); // crea oggetto
        world.printDate(); // chiama metodo
    }

    /** questo metodo serve a stampare la data */
    private void printDate() { // metodo
        d = LocalDate.now(); // chiama metodo static
        System.out.println("data: "+d);
    }
}

```

```

import java.time.LocalDate; // indica dove trovare la classe LocalDate

public class HelloWorld { // dichiara classe HelloWorld
    private static final String msg = "Lezione di Ingegneria del Software";
    private LocalDate d; // dichiara campo d di tipo LocalDate

    public static void main(String[] args) {
        System.out.println("Hello World"); // scrive su schermo
        System.out.println(msg);
        final HelloWorld world = new HelloWorld(); // crea oggetto, new serve per
        istanziare un oggetto
        world.printDate(); // chiama metodo, non togliere "world." perchè il metodo
        printDate non è static e il main non lo vede, è un metodo appartenente all'istanza
    }

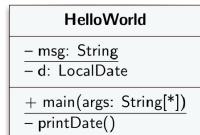
    private void printDate() { // metodo
        d = LocalDate.now(); // chiama metodo static now
        System.out.println(d);
    }
}

```

Note:

- d non è più un attributo ma fa parte dell'oggetto in sè e non della classe.
- In caso di static gli attributi vivono nella classe.
- Se un metodo è `static` allora posso chiamarlo dal main (presente nella stessa classe) oppure usando il nome della classe stesso.
 - I metodi che **NON** sono static non li posso chiamare sulla classe ma li devo invocare su un'istanza.
- *Per compilare si deve scaricare JDK.* `javac HelloWorld.java` e produce in output dei file con estensione `.class` e contiene il codice eseguibile del codice sorgente. Non è eseguibile dal SO perchè esso non si può lanciare sul SO.
- Per eseguire il codice si usa `java HelloWorld`.
- Quando si installa JAVA si installa anche JVM (Java Virtual Machine): quando si lancia `HelloWorld.class` esso si trova sopra la JVM (cioè che la JVM sa come interpretare ed eseguire il codice sorgente) e soltanto dopo la JVM comunica con il SO.

L'UML per questo codice è:



- Subito dopo (scendendo) `HelloWorld` posso mettere una **DOPPIA LINEA** e vuol dire che in questo diagramma non si illustra nessun attributo. Gli attributi possono anche non essere specificati nell'UML ma comunque possono esistere nel codice sorgente.
- il metodo `+main()` posso scriverlo anche in questo modo perchè sarebbe corretto ugualmente ma, come signature del main, si devono mettere obbligatoriamente nel codice.
- i metodi static vengono **SOTTOLINEATI**

Riepilogo veloce

- **class** permette di definire un tipo, e quindi le sue istanze
- **final** definisce un campo o una variabile che non può essere assegnata più di una volta (una costante). Una classe final non può essere ereditata, un metodo final non può essere ridefinito (override)
- **import** indica dove trovare la definizione di una classe che sarà usata nel seguito
- **new** permette di creare un'istanza di una classe
- **private** e **public** indicano l'accessibilità di classi, campi e metodi
- **static** è usata per dichiarare un campo o un metodo appartenente alla classe (e non all'istanza)
- **void** indica che il metodo non ritorna alcun valore
- Tipi usati: **String**, per rappresentare insiemi di caratteri; **LocalDate**, per accedere alla data attuale; **System** per scrivere sullo schermo
- In particolare **final** serve pure per **metodi** (non si possono ridefinire metodi, overload) e per le **classi** (non si può ulteriormente estendere tramite ereditarietà)

Tornando all'ingegneria del software...

L'obiettivo è riuscire a sviluppare un componente software che risulti:

- riusabile (software con vita lunga e usabili successivamente con nuovi sistemi. Si prendono codici già esistenti e si trasporta in sistemi diversi facendo le giuste correzioni)
- modificabile (modificare senza grande sforzo)
- corretto

Si suppone di avere i seguenti **requisiti**:

Dati vari file contenenti valori numerici, con un valore per ciascuna riga del file

1. **Leggere** da ciascun file la lista di valori
2. Tenere solo i valori **non duplicati**
3. Calcolare la **somma dei numeri letti** dal file (non duplicati)
4. Calcolare il **massimo** fra i numeri letti

Lettura da file

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;

public class CalcolaImporti { // classe Java vers 0.0.1
    private final List<String> importi = new ArrayList<>(); //creo istanza. List è
    un'INTERFACCIA cioè non ha metodi implementati, è una classe ASTRATTA
    // List e ArrayList sono tipi della libreria Java
    private float totale;

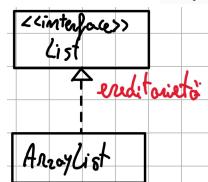
    public float calcola(String c, String n) throws IOException { // metodo c=cartella(path,
    percorso) n=nomeFile
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        //Number=legge anche un numero specifico di una linea
        // lettura file tramite le API Java: File, FileReader, LineNumberReader
        totale = 0;
```

```

        while (true) {
            final String riga = f.readLine(); // legge una riga dal file
            if (null == riga) break; // esce dal ciclo. null è scritto prima per evitare
di sbagliare e mettere solo un uguale, facendo così un'assegnazione
            importi.add(riga); // aggiunge in lista
            totale += Float.parseFloat(riga); // converte da String a float(non è un
cast) //Float è un tipo sul quale si possono essere invocati metodi.
        }
        f.close(); // chiude file
        return totale; // restituisce totale al chiamante
    }
}

```

La variabile `importi` fa uso di gerarchie di classi ed esse vengono rappresentate:



- La **linea tratteggiata** indica che una classe eredita da un'interfaccia
- La freccia, generalmente, viene chiamata **ASSOCIAZIONE**
- Gli attributi/metodi **static** vengono inizializzati appena si **avvia il programma**. Mentre, gli attributi **non static** vengono **inizializzati** quando viene **istanziata la classe**.
- `throws IOException` gestisce le **eccezioni**. Il chiamante del metodo deve gestire l'eccezione oppure deve rimandare a sua volta all'altro chiamante (può viaggiare a ritroso fra le chiamate e "in fondo" qualcuno deve saperla gestire)
 - In questo caso "mando" l'eccezione a `IOException`.
 - L'eccezione può essere gestita anche **internamente** al metodo con un'altra specifica sintassi
- Le variabili vengono inizializzate quanto più vicine al punto di utilizzo
- `ArrayList` è più veloce per la navigazione fra gli elementi perché sono contigui. In `List`, invece, si deve navigare sequenzialmente.

Il codice completo per l'esercizio è:

```

public class CalcolaImporti { // classe Java vers 0.1
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        // lettura file tramite le API Java: File, FileReader, LineNumberReader
        totale = massimo = 0;
        while (true) {
            final String riga = f.readLine(); // legge una riga dal file
            if (null == riga) break; // esce dal ciclo
            if (!importi.contains(riga)) { // se non presente, verifico se c'è
                qualcosa importi contiene la riga
                importi.add(riga); // aggiunge in lista
                float x = Float.parseFloat(riga); // converte da String a
float
                totale += x; // aggiorna totale
                if (massimo < x) massimo = x; // aggiorna massimo
            }
        }
    }
}

```

```

        }
    }
    f.close(); // chiude file
    return totale; // restituisce il totale al chiamante
}
}

```

Versione 2.0

```

public class CalcolaImporti { // classe Java vers 0.2
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();
            if (null == riga) break;
            if (!importi.contains(riga))
                importi.add(riga);
        }
        f.close();
        // calcola totale
        totale = 0;
        for (int i = 0; i < importi.size(); i++)
            totale += Float.parseFloat(importi.get(i));
        // calcola massimo
        massimo = Float.parseFloat(importi.get(0));
        for (int i = 1; i < importi.size(); i++)
            if (massimo < Float.parseFloat(importi.get(i)))
                massimo = Float.parseFloat(importi.get(i));
        return totale;
    }
}

```

*Quale codice è migliore fra le 2 versioni? Nessuna delle due. Perchè? Si sta facendo "**SPAGHETTI CODE**", cioè un codice scritto male*

Spaghetti Code

- Metodi lunghi, senza parametri, e che usano variabili globali
- Flusso di esecuzione determinato dall'implementazione interna all'oggetto, non dai chiamanti
- Interazioni minime fra oggetti
- Nomi classi e metodi indicano la programmazione procedurale
- Ereditarietà e polimorfismo non usati, riuso impossibile
- Gli oggetti non mantengono lo stato fra le invocazioni
- Cause: inesperienza con OOP, nessuna progettazione

- codice è monolitico: fa troppe cose in un unico flusso. Non è un codice Object-Oriented.
Conseguenze: non si può riusare, né verificarne la correttezza.
- Prima legge il file, poi vede se i valori sono duplicati, poi calcola il totale e poi ecc ecc...Fa 3 cose in contemporanea e quindi troppe operazioni

- E' il contrario della programmazione a oggetti perchè le soluzioni vanno trovate in modo suddiviso, quindi **creare metodi** specifici per una **singola operazione**.

```
Graficamente ho la seguente situazione:  
  
public class CalcolaImporti { // classe Java vers 0.2  
  
    private final List<String> importi = new ArrayList<String>();  
    private float totale, massimo;  
  
    public float calcola(String c, String n) throws IOException {  
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));  
        String riga;  
        while (true) {  
            riga = f.readLine();  
            if (null == riga) break;  
            if (!importi.contains(riga))  
                importi.add(riga);  
        }  
        f.close();  
        totale = 0;  
        for (int i = 0; i < importi.size(); i++) {  
            totale += Float.parseFloat(importi.get(i));  
        }  
        massimo = Float.parseFloat(importi.get(0));  
        for (int i = 1; i < importi.size(); i++)  
            if (massimo < Float.parseFloat(importi.get(i)))  
                massimo = Float.parseFloat(importi.get(i));  
        return totale;  
    }  
}
```

Qualità del codice

```
public class CalcolaImporti { // classe Java vers 0.2
    private final List<String> importi = new ArrayList<String>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)))
        String riga;
        while (true) {
            riga = f.readLine();
            if (null == riga) break;
            if (!importi.contains(riga))
                importi.add(riga);
        }
        f.close();
        // calcola totale
        totale = 0;
        for (int i = 0; i < importi.size(); i++)
            totale += Float.parseFloat(importi.get(i));
        // calcola massimo
        massimo = Float.parseFloat(importi.get(0));
        for (int i = 1; i < importi.size(); i++)
            if (massimo < Float.parseFloat(importi.get(i)))
                massimo = Float.parseFloat(importi.get(i));
        return totale;
    }
}
```

23

Prof. Tramontana - Marzo 2023

Problemi?

- Il metodo `calcola` di entrambe le versioni è **spaghetti code** (un **antipattern**)
- Il codice è monolitico: fa **troppe cose in un unico flusso**. Non è un codice Object-Oriented. Conseguenze: non si può riusare, né verificarne la correttezza
 - Come verificare che tutti i valori del file siano stati letti? Si dovrà modificare il metodo. Non è una soluzione, si dovrebbe **poter verificare il comportamento del metodo dall'esterno**
 - Analogamente per verificare il calcolo di somma e totale, in più punti si dovrebbero aggiungere alcuni controlli
 - Non si riesce a modificare facilmente o riusare il codice. Per es. se si volessero conservare tutti i valori letti, quali **ulteriori effetti** provoca la modifica?
- Quindi: difficoltà di comprensione e modifiche che coinvolgono varie operazioni

24

Prof. Tramontana - Marzo 2023

- Test del codice:** si fa eseguire il codice usando *appositi input* e si prende il valore risultante. Se è corretto (è quello che ci aspettiamo), allora il codice è corretto. Il test si fa senza modificare il codice, appunto. C'è un flusso che fa molte cose e quindi il codice scritto non va bene. I test servono per **verificare la qualità del software**.
- In questo unico flusso di codice non si può riusare il codice.
- Se il codice diventa più "importante" allora tali caratteristiche diventano fondamentali.

Spaghetti code

Può avere diverse **caratteristiche**:

- metodo molto lungo** o interi pezzi di codice molto lunghi
- metodo che non chiama altri metodi**, vuol dire che il metodo principale sta facendo troppe cose
- non si usano istanze di altri oggetti per mantenere i dati
- non si ha ben in mente la programmazione ad oggetti *OOP*, quindi **nessuna progettazione**
- non usa **ereditarietà e polimorfismo**
- ci sono molti **variabili locali**

Concetti base di progettazione (spiegato meglio più avanti)

1. **analisi dei requisiti** e su di essi bisogna fare un'**analisi grammaticale** (in modo da distinguere sostantivi, verbi e aggettivi). I sostantivi (quelli ripetuti più volte, i concetti attorno ai quali si aggirano i requisiti) saranno le classi oppure nomi di attributi.
2. Bisogna analizzare i sinonimi in modo da aggiustare i requisiti. I verbi indicano i metodi.
3. Se si usa bene la OOP, i nomi delle classi e degli attributi sono sostantivi. Un metodo, per esempio "calcola()" indica un verbo e non indica nulla di sostanziale. I nomi delle classi, dei metodi e degli attributi deve essere significativo in modo da avere la giusta astrazione.

Infatti, per quanto riguarda il codice di prima si può vedere in questo modo:

```
public class CalcolaImporti { // classe Java vers 0.2

    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
            riga = f.readLine();                                leggiFile()
            if (null == riga) break;
            if (!importi.contains(riga))                      inserisci()
                importi.add(riga);
        }
        f.close();                                         calcolaSomma()
        totale = 0;
        for (int i = 0; i < importi.size(); i++) {
            totale += Float.parseFloat(importi.get(i));
        }
        massimo = Float.parseFloat(importi.get(0));          calcolaMassimo()
        for (int i = 1; i < importi.size(); i++)
            if (massimo < Float.parseFloat(importi.get(i)))
                massimo = Float.parseFloat(importi.get(i));
        return totale;                                     getSomma()
    }
}
```

26

Prof. Tramontana - Marzo 2023

Se si suddivide in questo modo, il codice può diventare:

```
public class CalcolaImporti { // classe Java vers 0.1
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;
    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        totale = 0;
        massimo = 0;
        while (true) { //leggifile()
            String riga = f.readLine();
            if (null == riga) break;
            if (!importi.contains(riga)) { //inserisci()
                importi.add(riga);
                float x = Float.parseFloat(riga); //calcolaSomma()
                totale += x;
                if (massimo < x) massimo = x; //calcolaMassimo()
            }
        }
        f.close();
        return totale; //getSomma()
    }
}
```

E alla fine posso ottenere:

```
public class Pagamenti { // Pagamenti vers 1.1
    private List<String> importi = new ArrayList<>();
    private float totale, massimo;
    public void leggiFile(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while (true) {
```

```

        riga = f.readLine();
        if (null == riga) break;
            inserisci(riga);
    }
    f.close();
}

public void inserisci(String riga) {
    if (!importi.contains(riga))
        importi.add(riga);
}

public void calcolaSomma() {
    totale = 0;
    for (String v : importi) // enhanced for ---> for corto, si può usare solo
con le liste e non con gli array
        totale += Float.parseFloat(v);
}

public void calcolaMassimo() {
    massimo = 0;
    for (String v : importi)
        if (massimo < Float.parseFloat(v))
            massimo = Float.parseFloat(v);
}

public void svuota() {
    importi = new ArrayList<>();
    totale = massimo = 0;
}
public float getMassimo() {
    return massimo;
}
public float getSomma() {
    return totale;
}
}

// chiamate da un'altra classe
public static void main(String[] args) {
    Pagamenti p = new Pagamenti();
    try {
        p.leggiFile("csv","importi");
    }
    catch (IOException e) {
        //Istruzioni in caso di errore
        //potrei mettere un messaggio di errore e chiudere il programma oppure
potrei rifare l'esecuzione con un nome file che scelgo io(se ne conosco uno di default)
    }

    p.calcolaSomma();
    p.calcolaMassimo();
}

```

In questo caso va bene e si ha che:

- i nomi dei metodi sono della zioni, quindi dei verbi: si chiarisce meglio l'azione da fare

Pagamenti
<ul style="list-style-type: none"> - importi: List<String> - totale: float - massimo: float <ul style="list-style-type: none"> + leggiFile(c: String, n: String) + inserisci(rica: String) + calcolaSomma() + calcolaMassimo() + svuota() + getMassimo() : float + getSomma() : float

-
- Il metodo main viene chiamato da un'altra classe.

try catch

L'errore IOException può essere lanciata dalla lettura del file o dalla readline(). Si gestisce con try {} catch (){}

Su [Github](#) c'è il codice completo. [Anche qui](#) c'è il codice (quello rivisto)

Considrezioni

I metodi che ritornano un valore sono metodi che **LEGGONO LO STATO**. Gli altri metodi (quelli *void*) **MODIFICANO LO STATO** e si tratta del paradigma **Command e Query**. Si distinguono:

- metodi di tipo command: si potrebbero avere calcoli piuttosto complessi e quindi sono più evidenti e quindi **pesano di più** sulle prestazioni. Si usano solo quando strettamente necessari.
- metodi di tipo query: usato per fare interrogazioni all'oggetto, ritorneranno qualcosa. Essi saranno veloci, leggono un valore e lo ritornano. Non ci si fanno problemi sullo spazio che occupa tale metodo e quindi sono **trascurabili**.

Ogni volta ci si chiede "*Mi interessava lo stato precedente di un oggetto se lancio un metodo di tipo Command?*"

- Il for semplificato si può chiamare su Liste oppure tipi di java, per esempio Iterable che è un'altra interfaccia.

Metriche

Al fine di analizzare il codice si usano delle metriche. La più usata è la **LOC** (si contano tutte le linee di codice) e si ha un ammontare di quanto lavoro è stato fatto

- Classe Calcolalimporti (vers. 0.1)
 - Metodi 1, LOC 26 (di cui 5 linee sono per i vari import)
- Classe Calcolalimporti (vers. 0.2)
 - Metodi 1, LOC 29
- Classe Pagamenti (vers 1.1)
 - Metodi 7, LOC 43 (media 6 LOC per metodo)

Quando un metodo è piccolo?

Si può confrontare il metodo implementato con altri metodi di sistemi che hanno un certo numero di classi:

- J Unit (JU), usato per scrivere codice di test. E' un sistema fatto da:
 - JU LOC 22K, Classi 231, Metodi 1200, Attributi 265, media di linee di codice per metodo = 18
 - 18 è una "soglia" calcolata, addirittura, per eccesso. Quindi se si fa meglio di così, allora va meglio.

Un altro esempio è:

- JHD LOC 28K, Classi 600, Metodi 4814, Attributi 1151, media 6

Come si contano le righe?

Bisogna seguire la convenzione stabilita per scrivere il codice. Bisogna seguire anche una convenzione sul conteggio. Se decido un metodo per un primo pezzo, uso lo stesso modo per il resto.

Un altro modo è contare i ; . Dipende tutto dal modo di procedere dell'azienda.

Conclusioni

- Devo sapere se il codice è corretto, quindi con i test. (metodi piccoli e ben suddivisi)
- Antipattern spaghetti code

Esempi domande esame

- Implementare un frammento di codice che usa l'enhanced for
- Dire come si può controllare se un codice è corretto
- Implementare un metodo query
- Dire qual è la differenza fra List ed ArrayList
- Dire a cosa serve il metodo contains di List

Miglioramenti sul codice

Le astrazioni sono dei blocchi (classi, metodi di più alto livello) di codice che permettono di programmare le successive cose. Avendo i metodi piccoli si possono creare i test.

Test

ESAME: Cos'è un test?. E' un codice che contiene un'invocazione del metodo che si vuole verificare e questo prende in ingresso dei parametri (o glieli passo al metodo oppure li uso per inizializzare tutta l'istanza di cui il metodo fa parte).

I parametri sono detti **DATI IN INPUT**.

Quindi i test hanno i seguenti **parametri**:

- Chiamata del metodo
- Scelta e passaggio dei parametri prima dell'esecuzione del metodo
- Si confronta il risultato con il valore atteso e si rendono **AUTOVALUTANTI** scegliendo proprio il valore atteso.
- Devono essere indipendenti. L'ordine di esecuzione dei test non deve modificarne l'esito

(**ESAME**) Da dove si prende il valore atteso?

- Si guarda il **documento dei requisiti** affinchè si capisca quali dati si possono accettare e quali no. Se non trovo tale valore grazie ai requisiti, allora il documento non è scritto bene.
- Si può trovare anche usando un oracolo ed è, per esempio, la calcolatrice che da il risultato. Quindi l'oracolo è un sistema esterno che, dati gli stessi input, mida un valore corretto. Prima di realizzare tutto il sistema, ovvero il software, si può fare un "modello del sistema" che avrà il compito di **ORACOLO**.

```
public class TestPagamenti { // per classe Pagamenti vers 0.9
    private Pagamenti pgm = new Pagamenti();

    private void initLista() {
        pgm.svuota();
        pgm.inserisci("321.01");
        pgm.inserisci("531.7");
        pgm.inserisci("1234.5");
    }

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

    public void testMaxValore() {
        initLista();
        pgm.calcolaMassimo();
        if (Math.abs(pgm.getMassimo() - 1234.5f) < 0.01)
            System.out.println("OK test massimo val");
        else System.out.println("FAILED test massimo val");
    }
    // continua ...
    2
    Prof. Tramontai

    // continua
    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();
        tl.testMaxValore();
    }
}
```

Output dell'esecuzione di TestPagamenti quando il file Importi.csv è presente nella cartella csv (dentro la cartella con gli eseguibili)
OK test leggi file
OK test somma val
OK test massimo val

Esecuzione quando il file non viene trovato
FAILED test leggi file
OK test somma val
OK test massimo val

Per ogni metodo si deve fare un test in una classe a parte.

`testMaxValore()` e `testSommaValori()` inizializzano la lista di per sé. Quindi ogni test deve essere **indipendente** dall'altro. L'ordine di esecuzione dei test non deve modificare il risultato

Se invece non uso il paradigma Command e Query, i metodi restituiscono qualcosa e potrei scrivere il test in questo modo:

```
public class TestPagamenti { // per classe Pagamenti vers 1.2
    private Pagamenti pgm = new Pagamenti();
    private void initLista() {
        pgm.svuota();
        pgm.inserisci(321.01f);
        pgm.inserisci(531.7f);
        pgm.inserisci(1234.5f);
    }

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

    public void testListaVuota() {
        pgm.svuota();
        if (pgm.calcolaSomma() == 0) System.out.println("OK test somma val empty");
        else System.out.println("FAILED test somma val empty");
        if (pgm.calcolaMassimo() == 0) System.out.println("OK test massimo val empty");
        else System.out.println("FAILED test massimo val empty");
    }
}
```

Esempio di test completo scritto bene

```
import java.io.IOException;

/**
 * La classe TestPagamenti è sottoclasse di MyTestSupport che implementa i
 * metodi assert (assertEquals e assertTrue).
 */
public class TestPagamenti extends MyTestSupport {
    private final Pagamenti pagam = new Pagamenti();
    private final String nomeFile = "Importi.csv";
    private final String percorso = "./";

    public static void main(String[] args) {
        TestPagamenti t = new TestPagamenti();
        t.testLeggiFile();
        t.testLeggiValoreDaFile();
        t.testSommaValori();
        t.testMaxValore();
    }

    private void initLista() {
        pagam.svuota();
        pagam.inserisci("321.01");
        pagam.inserisci("531.7");
        pagam.inserisci("1234.5");
        pagam.converti();
    }

    public void testSommaValori() {
        initLista();
        // chiama il metodo da testare
        pagam.calcolaSomma();
        // legge il risultato calcolato e lo confronta con il risultato atteso
        assertEquals(pagam.getSomma(), 2087.21f, "somma");
        // assertEquals() è definito in una classe esterna
    }

    public void testLeggiFile() {
        pagam.svuota();
        try {
            pagam.leggiFile(percorso, nomeFile);
            assertTrue(true, "apri e leggi file");
        } catch (IOException e) {
            assertTrue(false, "apri e leggi file");
        }
    }

    public void testLeggiValoreDaFile() {
        pagam.svuota();
        try {
            pagam.leggiFile(percorso, nomeFile);
        } catch (IOException e) {
            assertTrue(false, "leggi valore da file");
        }
        if (pagam.getDimens() > 1) {
            assertTrue(true, "valore letto da file");
            String elem = pagam.getElem(0);
        }
    }
}
```

```

        assertEquals(Float.parseFloat(elem), 1946.28f, "valore da file");
    }

}

public void testMaxValore() {
    initLista();
    pagam.calcolaMassimo();
    assertEquals(pagam.getMassimo(), 1234.5f, "massimo valore");
}
}

```

Ricapitolando in generale

Correttezza: è stato possibile eseguire test che verificano il codice.

Soddisfare (e verificare) i requisiti permette di ottenere la qualità del sistema

- I test documentano le condizioni sotto le quali il codice funziona e protegge il codice da modifiche indesiderate. I test devono essere indipendenti fra loro ed auto valutarsi
- **Test Driven Development (TDD)**: scrivere prima il test e poi il codice che risolve un requisito. Lo sviluppatore sceglie nomi di classi e metodi (progettazione). Serve per organizzarsi il lavoro e non riguarda la scrittura del codice. E' considerata una buona pratica per lo sviluppo guidato dei test. **Si scrivono prima i test e poi il codice.**
 - Se il codice di test **non funziona/compila** allora posso scrivere il codice dell'applicazione.
 - Se **compila e non rileva errori**, allora si scrive altro codice di test e non si può scrivere codice per l'applicazione
 - Costringe di fare progettazione prima di scrivere il codice del software
 - Motto: **Red green refactor**: red (test non andato a buon fine, errore su console) -> green (test andati a buon fine) -> Refactor (il codice che ho scritto per il test lo scrivo meglio in modo da renderlo **MODULARE**
 -)
- **Responsabilità**: i compiti sono suddivisi su vari metodi (e quindi su più classi), questo permette di ottenere coesione del codice
 - Principio di Singola Responsabilità. La singola responsabilità è cruciale per la comprensione, il riuso, l'ereditarietà (OOP)
- **Astrazioni**: lo sviluppatore OOP costruisce astrazioni. Il nome delle astrazioni è estremamente importante: il nome che si dà a classi e metodi ne descrive l'obiettivo

`assert()` sono usati dalla libreria JUnit

Paradigmi

Paradigma **DRY** = codice scritto in questa versione di test è di questo tipo. DRY vuol dire Don't repeat yourself ovvero **non scrivere codice ripetuto**.

Paradigma **KISS** = Keep It Simple Stupid, oppure Keep It Stupid Simple ovvero indica un codice quanto più semplice possibile

Paradigma **YAGNI** = You aren't gonna need it, ovvero Non ti servirà in futuro. Se serve un test che mi giustifica la scrittura di un certo codice, allora lo scrivo. Altrimenti non lo scrivo.

Conclusioni

- Key points
 - Test driven development
 - Single Responsibility Principle
- Esempi di domande d'esame
 - Implementare un test per un metodo che prende in ingresso un intero
 - Dire come si compila una classe in Java
 - Implementare una classe Java che può essere data alla JVM per essere eseguita

Punti Importanti Da Ricordare

- Correttezza: è stato possibile eseguire test che verificano il codice. Soddisfare (e verificare) i requisiti permette di ottenere la qualità del sistema
- I test documentano le condizioni sotto le quali il codice funziona e protegge il codice da modifiche indesiderate. I test devono essere indipendenti fra loro ed auto valutarsi
- Test Driven Development (TDD): scrivere prima il test e poi il codice che risolve un requisito. Lo sviluppatore sceglie nomi di classi e metodi (progettazione)
- Responsabilità: i compiti sono suddivisi su vari metodi (e quindi su più classi), questo permette di ottenere coesione del codice
- Principio di Singola Responsabilità. La singola responsabilità è cruciale per la comprensione, il riuso, l'ereditarietà (OOP)
- Astrazioni: lo sviluppatore OOP costruisce astrazioni. Il nome delle astrazioni è estremamente importante: il nome che si da a classi e metodi ne descrive l'obiettivo

6

Prof. Tramontana - Marzo 2019

Con il **Test Driven Development (TDD)** serve ed è un buon metodo per scrivere codice in modo da ottenere correttezza e fare progettazione nell'implementazione. Questo "metodo" serve per lo **sviluppo** e non dà esiti sul test.

(**ESAME**) Assumendo questa tecnica si "finisce" di scrivere i test NON sono tutti quelli che servono per verificare la correttezza. Tutto quello che viene implementato ha una serie di incastri e input e i test non bastano perchè bisogna scrivere i test dopo che si conosce tutto il codice perchè i test si sono scritti "prima" di capire che il codice potesse funzionare in diversi stati.

Per quanto riguarda il **principio di singola responsabilità** bisogna individuare per ogni pezzo di codice la responsabilità del frammento stesso. Esso è il nome del metodo/classe deve **comunicare l'obiettivo** e senza usare nomi fuorvianti.

Di solito si ha un'unica responsabilità per ogni componente (metodo/classe/sottoinsieme di classi) che si sta implementando.

Più il componente è **COESO** (tutti i compiti della classe contribuiscono alla responsabilità del componente) più si può dire che la classe ha una singola responsabilità.

In caso i file si trovano in cartelle diverse si usa l'opzione `-cp` (class path) per fare in modo che, durante la compilazione, i file delle varie classi vengano trovati. (approfondiremo più avanti)

Conclusioni

- Key points
- Test driven development
- Single Responsibility Principle
- Esempi di domande d'esame
 - Implementare un test per un metodo che prende in ingresso un intero
 - Dire come si compila una classe in Java
 - Implementare una classe Java che può essere data alla JVM per essere eseguita

Classi e oggetti

A run-time vengono creati degli oggetti che corrispondono a quello che si è implementato. **Le relazioni fra le varie classi** sono definite nel codice (istanziare una classe A dentro una classe B e si crea una dipendenza). C'è una relazione anche **fra la classe e un'istanza di essa**.

Scegliere il nome corretto per la classe (in base all'obiettivo) permette di costruire un **VOCABOLARIO** per quell'applicazione che si sta sviluppando.

Una volta scritto il codice, quante istanze mi possono servire? Posso fare in modo di limitare le istanze?
La situazione è:

```
public class MainPagam { // versione 0.1
    public static void main(String[] args) {
        Pagamenti p = new Pagamenti(); // p è una istanza di Pagamenti
        try {
            p.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
            // p contiene tutti i valori letti dal file
        } catch (IOException e) {
        }
        System.out.println("totale: " + p.calcolaSomma());
        System.out.println("max: " + p.calcolaMassimo());
    }
}
```

Oppure potrei anche:

```
public class MainPagam { // versione 0.2
    public static void main(String[] args) {
        Pagamenti p = new Pagamenti(); // p e' l'unica istanza
        try {
            p.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
        } catch (IOException e) {
        }
        System.out.println("file 1 totale: " + p.calcolaSomma());
        System.out.println("file 1 max: " + p.calcolaMassimo());

        try {
            p.leggiFile("csvfiles", "PagMarzo.csv"); // lettura secondo file
            // p adesso contiene tutti i valori letti da entrambi i file
        } catch (IOException e) {
        }
        System.out.println("file 1 e 2 totale: " + p.calcolaSomma());
        System.out.println("file 1 e 2 max: " + p.calcolaMassimo());
    }
}
```

Si chiama nuovamente leggiFile() sulla stessa istanza.

La stessa istanza tiene la lista di valori da 2 file diversi. leggiFile() Non controlla se la lista di partenza è già riempita oppure no

Quindi il metodo leggiFile() della classe Pagamenti **NON** cancella i valori in lista, posso leggere più file e inserirli nella stessa lista con varie chiamate a leggiFile()

Potrei avere 2 istanze di Pagamenti:

```
public class MainPagam { // versione 0.3
    public static void main(String[] args) {
        Pagamenti p1 = new Pagamenti(); // prima istanza
        Pagamenti p2 = new Pagamenti(); // seconda istanza

        try {
            p1.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
            p2.leggiFile("csvfiles", "PagMarzo.csv"); // lettura secondo file
        } catch (IOException e) {
        }
        System.out.println("file 1 totale: " + p1.calcolaSomma());
        System.out.println("file 1 max: " + p1.calcolaMassimo());
        System.out.println("file 2 totale: " + p2.calcolaSomma());
        System.out.println("file 2 max: " + p2.calcolaMassimo());
    }
}
```

In questo caso non c'è un'unica istanza che contiene tutti i valori. In generale può servirmi la versione 0.2 o 0.3 e dipende dai requisiti. Si suppone che si vuole alla versione 0.2, cioè un'istanza deve contenere i dati di file diversi, quindi vietando che venga fatta nuova new di Pagamenti.

Supponiamo che l'applicazione non debba avere più di una istanza di Pagamenti, occorre il **Design Pattern Singleton** per vietare la creazione di più istanze.

Design Pattern

Un **Design Pattern** comprende **soluzioni** per problemi non banali con la propria descrizione che risultano essere frequenti. Un Design Pattern è suddiviso in **Intento**, **Descrizione** del problema, descrizione della **Soluzione**, **Frammento di codice** corrispondenti alla soluzione, **Conseguenze**.

Design Pattern Singleton

Si assicura che la classe abbia una sola istanza all'interno del software per fornire un accesso globale all'istanza.

Intento

In tutta l'applicazione deve essere possibile creare solo un'istanza. Il compilatore deve fornire un errore. Si vuole poter accedere all'istanza che si crea.

Motivazione (spiegazione del problema)

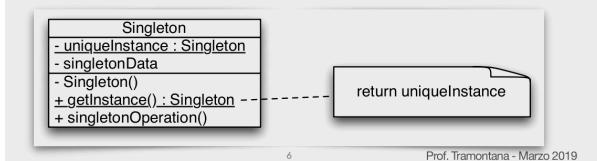
Si vogliono mettere i dati tutti insieme in modo da esserci unica entità. Classici esempi possono essere:

- in caso di una stampa, detto *spooler di stampa*
- conteggio del numero di finestre (*window manager*)
- carico da un database dati di clienti

Nel Sistema Operativo non ci possono essere più code di stampa ma una sola in modo da mantenere un ordine.

Soluzione

- La classe che deve essere un *Singleton* dovrà implementare un'operazione `getInstance()` sulla classe (ovvero, in Java è un metodo static) che ritorna l'unica istanza creata
- La classe *Singleton* è responsabile per la creazione dell'istanza
- Il costruttore della classe *Singleton* è privato, così da non permettere la creazione tramite `new` ad altre classi



- costruttore privato (protegge dalla creazione di altre istanze)
- metodo statico chiamato `getInstance` che restituisce l'unica istanza della classe.
- si tiene l'istanza in un attributo statico

Esempio 1:

```

// Classe Singleton che tiene una lista di
// interi
public class Fib {
    // l'unica istanza e' tenuta da obj
    private static Fib obj = new Fib();
    private int[] x={1, 2, 3, 5, 8, 13, 21,
    34, 55, 89, 144};
    private int i;
    private Fib() {
        i = 3;
    }
    public static Fib getInstance() {
        return obj; // restituisce l'istanza
    }
    public int getValue() {
        if (i<11) i++;
        return x[i-1];
    }
    public void revert() {
        i = 0;
    }
}

```

7 Prof. Tramontana - Marzo 2019

```

public class TestFib {
    public static void main(String[] args) {
        // richiede una istanza di Fib
        Fib f = Fib.getInstance();
        System.out.print("f "+f.getValue());
        System.out.println(" "+f.getValue());

        // richiede una nuova istanza
        Fib f2 = Fib.getInstance();
        System.out.print("f2 "+f2.getValue());
        System.out.println(" "+f2.getValue());

        // Si ha un errore a compile-time con:
        // Fib f3 = (Fib) f2.clone();
        // Fib f4 = new Fib();
    }
}

Output dell'esecuzione
f 5 8
f2 13 21

```

Output dell'esecuzione
f 5 8
f2 13 21

Superclasse Object

Il metodo `clone()` non è implementato in `Fib`, ma è a disposizione da Java per gli oggetti. Tutte le classi del software che si sta sviluppando sono sempre SOTTOCLASSI della classe `Object` implementata da Java. In questo modo ci sono dei metodi di `Object` che si possono richiamare. In automatico la classe `Object` è superclasse di tutte le classi che si vanno a creare.

Si potrebbe fare: `Object o1 = new Fib();` quindi c'è **compatibilità di tipi**.

Inoltre `List<Object> l;` possiamo mettere qualsiasi tipo di oggetto in questa lista.

Esempio 2:

Si vogliono registrare dei messaggi (detti messaggi di **log**) che ci dicono cosa sta succedendo durante l'esecuzione.

Esempio Classe Logs

```

public class Logs {           // Classe Singleton
    private static Logs obj;   // obj tiene l'istanza
    private List<String> l;    // tiene i dati da registrare

    private Logs() {           // il costruttore è privato
        empty();
    }
    public static Logs getInstance() { // restituisce l'unica istanza
        if (obj == null) obj = new Logs();
        return obj;
    }
    public void record(String s) { // accoda il dato
        l.add(s);
    }
    public String dumpLast() {    // restituisce l'ultimo dato
        return l.getLast();
    }
    public String dumpAll() {     // restituisce tutti i dati
        String acc = "";
        for (String s : l) // s tiene ciascun elemento in lista, ad ogni passata
            acc = acc.concat(s);
        return acc;
    }
    public void empty() {
        l = new ArrayList<String>();
    }
}

```

8 Prof. Tramontana - Marzo 2019

In questo caso l'inizializzazione si fa in `getInstance()`. Questa versione è "migliore" perché, se non si usa l'istanza, si ha **più spazio in memoria** e verrà inizializzata solo al richiamo di `getInstance()`. Ciò dipende dai requisiti e la versione con `Fib` può anche andare bene. In generale le versioni sono **ottime entrambe** perchè raggiungono l'obiettivo.

Potrebbe anche non servire quindi si ritarda il più possibile l'inizializzazione.

OT: In caso di esecuzione concorrente (in caso di più thread di esecuzione, più processi) questo codice crea problemi perchè potrebbe creare più di un'istanza ma ciò non è previsto dal Singleton. In questo caso era meglio la versione di Fib

Test Per Classe Logs

```

public class TestLogs {
    private Logs lg = Logs.getInstance();

    public void testSinglO {
        initLogs();
        Logs lg2 = Logs.getInstance();
        lg2.record("uno");
        lg2.record("due");
        if (lg.dumpLast().equals("due"))
            System.out.println("OK test logs singl");
        else
            System.out.println("FAILED test logs singl");
    }
    public void testLastO {
        initLogs();
        if (lg.dumpLast().equals("three "))
            System.out.println("OK test logs last");
        else
            System.out.println("FAILED test logs last");
    }
    public void testAllO {
        initLogs();
        if (lg.dumpAll().equals("one two three "))
            System.out.println("OK test logs all");
        else
            System.out.println("FAILED test logs all");
    }
}

private void initLogsO {
    lg.emptyO;
    lg.record("one ");
    lg.record("two ");
    lg.record("three ");
}

public static void main(String[] args) {
    TestLogs tl = new TestLogs();
    tl.testSinglO;
    tl.testAllO;
    tl.testLastO;
}

```

9

Output dell'esecuzione
OK test logs singl
OK test logs all
OK test logs last

Prof. Tramontana - Marzo 2019

Considerazioni design pattern singleton

La classe che si crea ha una sola istanza e nessun'altro può creare un'altra istanza della stessa classe.

- Se si usano solo metodi static in una classe non potrei ottenere lo stesso risultato di Singleton? Sì, si ottiene lo stesso effetto: non si vieta la crea la creazione di una nuova istanza ma comunque non è importante visto che gli attributi/metodi sono "*di classe*". Ma questa soluzione è **peggiore**.
- In caso si vogliano cambiare delle cose, quindi in caso di riuso, allora ci sono dei problemi. Si potrebbero cambiare i metodi con i quali la classe verrà usata.
- In caso di Singleton e di necessità di più istanze il punto da cambiare è solo `getInstance()`, e rendere **pubblico il costruttore** e scrivere semplicemente:

```

public Pagamenti(){} //si potrebbe anche eliminare

public static Pagamenti getInstance(){
    return new Pagamenti();
}```

## Multiton
Partendo da un singleton si può scrivere una variante **Multiton** dove si sa che il numero è **finito** di istanze è **limitato** mettendo un **contatore** e contare il numero di istanze create. Quindi rimane come una classe normale, non singleton appunto.

```

```

```java
private static int numeroIstanze = 0;
public Pagamenti(){} //si potrebbe anche eliminare

public static Pagamenti getInstance(){
 if(numeroIstanze < 10){
 numeroIstanze++;
 return new Pagament();
 }
 return null;
}

```

Principio delle **Conseguenze Locali**: un cambiamento in qualche punto del codice non dovrebbe causare problemi in altri punti.

Una variante Multiton di Pagamenti potrebbe associare una istanza a ciascuna cartella, la decisione sull'istanziazione dipende quindi dall'aver già creato (o meno) un'istanza per una data cartella.

Implementare come esercizio il Multiton di Pagamenti.

- In caso si adotti "tutto static" allora ogni modifica su un'istanza modifica anche tutte le istanze. E non è molto utile come metodo

In sintesi si può dire:

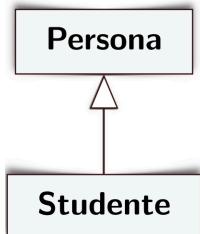
### Conseguenze Del Singleton

- La classe che è un *Singleton* ha pieno controllo di come e quando i client accedono al valore della sola istanza
- Evita che esistano variabili globali che tengono la sola istanza condivisa
- Permette di controllare il numero di istanze create in un programma, facilmente ed in un solo punto
- La soluzione è più flessibile rispetto a quella di usare static per tutte le operazioni e le variabili, poiché si può cambiare facilmente il numero di istanze consentite
- L'unico frammento di codice da modificare quando si vuol variare il numero di istanze create è quello della classe che è *Singleton*, mentre usando static si dovrebbero modificare tutte le invocazioni

# Riuso delle classi (Ereditarietà)

- Il codice viene riusato per **non fare copia-incolla** e per evitare di modificare per un nuovo scopo.
- Sulla sottoclasse si possono chiamare i metodi della superclasse. Serve per un riutilizzo semplice del codice.
- La sottoclasse serve per avere una **classe più specializzata** e più attributi/metodi.

Lo schema UML che rappresenta la **relazione** è:



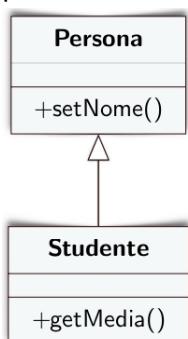
Di norma la **superclasse sta in alto** e la **sottoclasse sta basso** e non si fa **MAI** al contrario oppure in orizzontale (*quest'ultima casistica si fa in rari casi, ovvero quando non si ha altro spazio necessario*)

La sintassi Java è:

```
//Persona.java
public class Persona {...}
```

```
//Studente.java
public class Studente extends Persona {...}
```

- La sottoclasse **può usare tutti i metodi della superclasse**, aggiungerne altri oppure **ridefinire** i metodi della superclasse usando la stessa *signature* (*Override*) **MA** i metodi della superclasse **NON** possono essere *rimossi*.



In questo esempio la sottoclasse *Studente* ha i metodi `getMedia()` e `setNome()`.

## Esempio: (dipendenze/associazioni)

Si vuole rappresentare la seguente gerarchia seguendo lo schema UML:

```

public class Persona {
 private String nome, co;
 public void setNome(String n, String c) {
 nome = n; co = c;
 }
 public void printAll() {
 System.out.println(nome + " " + co);
 }
}

public class Studente extends Persona {
 private String matr;
 private List<Esame> esami = new ArrayList<>();
 public void setMatr(String m) { matr = m; }
 public void nuovoEsame(String m, int v) {
 Esame e = new Esame(m, v);
 esami.add(e);
 }
 public float getMedia() {
 if (esami.isEmpty()) return 0;
 float sum = 0;
 for (Esame e : esami) sum += e.getVoto();
 return sum / esami.size();
 }
 public void printAll() {
 super.printAll();
 System.out.println("matr: " + matr);
 for (Esame e : esami)
 System.out.println(e.getMatr() + ": " + e.getVoto());
 System.out.println("media: " + getMedia());
 }
}

```

```

//MainEsami
/**
 * Classe con metodo main che istanzia Studente e Persona e ne chiama i metodi
 */
public class MainEsami {

 public static void main(String[] args) {
 Studente s = new Studente("Alan", "Rossi");
 s.setMatr("M12345");
 s.nuovoEsame("Italiano", 8); // metodo della classe di s
 s.nuovoEsame("Fisica", 7);
 s.printAll(); // metodo della classe di s
 testUno(s);
 }

 /**
 * Mostra il controllo dei tipi quando si usa l'ereditarietà.
 */
 private static void testUno(Studente s) {
 System.out.println("Dopo l'assegnazione a p di tipo Persona");
 Persona p = s;
 s.nuovoEsame("Inglese", 9);
 // p e' dichiarato di tipo Persona
 // a runtime p punta all'istanza s
 p.printAll();
 // p.getMedia();
 }
}

```

- Il collegamento Studente ---> Esame ed MainEsami ---> Studente hanno una freccia diversa alla freccia usata per l'ereditarietà fra Studente -> Persona . Questa relazione è detta **ASSOCIAZIONE**. Questa indica una **dipendenza fra 2 classi** dove il punto di partenza della freccia indica che la classe ha una dipendenza verso la classe in direzione della freccia.
  - Vuol dire che se devo capire esattamente la classe Studente devo capire esattamente la classe Esame .
  - Se devo compilare Studente non potrò mai compilare in modo completo perchè **Studente usa la classe Esame (metodi e/o attributi) al suo interno**.
  - Per ovviare a questo problema devo compilare **prima** Esame e **poi** Studente
- Dentro MainEsami avrà una dipendenza da Studente cioè verranno usati metodi/attributi di Studente .

```
private List<Esame> esami = new ArrayList<>();
```

- Qui la classe Studente dipende da Esame
- In `ArrayList<>` non viene specificato il parametro perchè è specificato in `List<Esame>`

## Visibilità

- **private**: visibile solo all'interno della classe, si può accedere solo dall'interno della classe
- **public**: visibile ed invocabile da qualunque altro punto del codice anche fra diversi file o cartelle (in questo caso si deve specificare "dove trovare i *metodi/attributi public*" perchè Java fa uso dei **package**)
- **protected**: visibile solamente alle sottoclassi (oltre che alla classe stessa)

Di norma:

- I nomi delle classi che sono in alto alla gerarchia devono essere nomi **semplici e non composti**.

## Interfacce

Sono una definizione di tipo che indica il nome del tipo e possono indicare i nomi dei metodi pubblici di quel tipo **senza implementazione**.

Non ci sono

1. metodi privati
2. attributi non inizializzati
3. costruttori

La sintassi di Java:

```
public interface IAccount{
 public void setBalance();
}
```

Per esempio (con `List`):

```
List<Esame> esami = new ArrayList<>();
List<Esame> esami = new LinkedList<>();
```

- List è interfaccia e `ArrayList` e `LinkedList` sono sottoclassi

## Classi astratte

Sono delle classi dove c'è **almeno un metodo NON IMPLEMENTATO** fra tutti i metodi. Le classi astratte **non possono essere istanziate**.

La sintassi in Java:

```
public abstract class Libro {
 private String autore;
 public abstract void insert(); //metodo astratto non implementato
 public String getAutore() {
 return autore;
 }
}
```

- I metodi astratti possono essere usati all'interno della stessa classe perchè, non potendo istanziare una classe abstract, questo codice non sarà quello che **verrà eseguito dalla sottoclasse** (che avrà implementato il metodo **abstract**)

Per esempio:

```
public abstract class Libro {
 private String autore;
 public abstract void insert(); //metodo astratto non implementato
 public String getAutore() {
 insert(); //metodo astratto usato qui ma verrà eseguito nelle sottocalss
 return autore;
 }
}
```

```
public record Esame(String materia, int voto){}
```

è esattamente uguale alla classe Esame scritta in precedenza. Questo si fa perchè la classe Esame non fa molto e fa da **CONTENITORE**

- Quando si vuole inserire un valore si devono passare al costruttore `Esame e=new Esame(string,int);`
- Quando si vogliono leggere i valori sono come dei getter ma sono esattamente i nomi degli attributi.
  - `Esame.materia()` e `Esame.voto()`

Esempio completo:

```
//Studente.java new
import java.util.ArrayList;
import java.util.List;

/**
 * Classe Studente è sottoclasse di Persona.
 */
public class Studente extends Persona {
 private String matr;
 private List<Esame> esami = new ArrayList<>();

 public Studente(String nome, String cognome) {
 super(nome, cognome);
 }

 /**
 * Imposta la matricola.
 */
 public void setMatr(String m) {
 matr = m;
 }

 /**
 * Inserisce un nuovo esame.
 *
 * @param m nome materia
 * @param v voto ricevuto all'esame
 */
}
```

```

public void nuovoEsame(String m, int v) {
 Esame e = new Esame(m, v);
 esami.add(e);
}

/**
 * Calcola e restituisce la media
 *
 * @return float la media degli esami
 */
public float getMedia() {
 if (esami.isEmpty())
 return 0;
 float sum = 0;
 for (Esame e : esami)
 sum += e.voto();
 return sum / esami.size();
}

/** Scrive la lista degli esami sostenuti. */
@Override
public void printAll() {
 super.printAll();
 System.out.println("matr: " + matr);
 for (Esame e : esami)
 System.out.println(e.materia() + ":" + e.voto());
 System.out.println("media: " + getMedia());
}
}

```

## Classi ed interfacce

- Una classe può implementare un'interfaccia, ovvero, la classe fornisce un'implementazione dei metodi definiti dall'interfaccia
- Non è possibile istanziare interfacce
- Tramite l'interfaccia i client sanno cosa possono invocare. Per istanziazioni di oggetto e invocazioni di metodo, si può usare una qualsiasi delle implementazioni disponibili per l'interfaccia
- Un client che usa un'interfaccia rimane immutato quando l'implementazione dell'interfaccia cambia

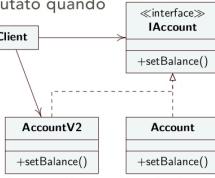
```

public class AccountV2 implements IAccount {
 public void setBalance() { }
}

public class M {
 public void main(String[] args) {
 IAccount a = new AccountV2();
 a.setBalance();
 }
}

```

8



Prof. Tramontana - Marzo 2020

- **implements** serve per implementare interfacce. Se a un certo punto della gerarchia ho un'interfaccia la posso usare in questo modo.

## Compatibilità di tipi

- L'ereditarietà permette di definire una **classificazione di tipi**. Una **sottoclasse è un sottotipo compatibile con la superclasse**, ovvero una sottoclasse è anche ciò che è la superclasse
- Esempio, si abbia Studente sottoclasse di Persona
  - Il tipo Studente è compatibile con il tipo Persona
  - La classe Studente fa tutto ciò che fa Persona, ed altre cose oltre quelle che fa Persona
- Una sottoclasse può prendere il posto della superclasse
- Esempio: si può usare un'istanza di Studente al posto di una di Persona. Dove compare p.setNome() con p di tipo Persona posso sostituire s.setNome() con s di tipo Studente
- Attenzione non vale il contrario, non si può usare la superclasse dove si usava la sottoclasse

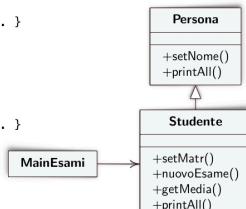
9 Prof. Tramontana - Marzo 2020

```
public class Persona {
 public void setNome(String n, String c) { ... }
 public void printAll() { ... }
}

public class Studente extends Persona {
 public void setMatr(String m) { ... }
 public void nuovoEsame(String m, int v) { ... }
 public float getMedia() { ... }
 public void printAll() { ... }
}

public class MainEsami {
 public static void main(String[] args) {
 Studente s = new Studente();
 s.setNome("Alan", "Rossi"); // metodo della superclasse di s
 s.setMatr("M12345");
 s.nuovoEsame("Italiano", 8); // metodo della classe di s
 s.printAll(); // metodo della classe di s

 s.nuovoEsame("Fisica", 7);
 Persona p = s; // p e' dichiarato di tipo Persona
 p.printAll(); // a runtime p punta all'istanza s
 }
}
```



10 Prof. Tramontana - Marzo 2020

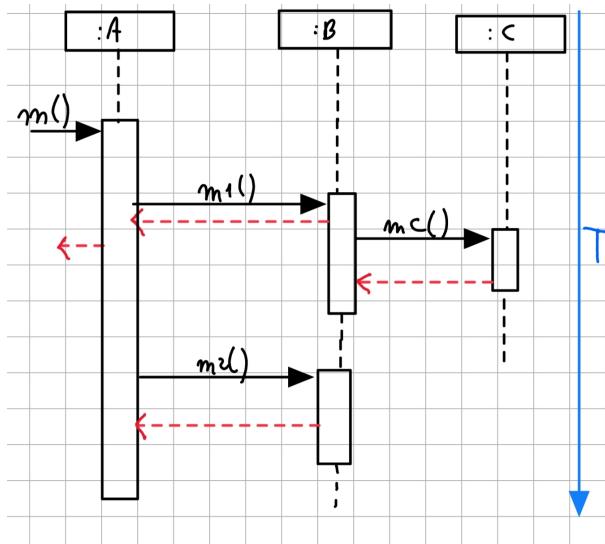
- In questo esempio, la riga p.printAll(); chiamerà il metodo di Studente perché l'istanza nel codice è unica ed è di Studente. Il compilatore non dà errori prima della compilazione perchè vede che in Persona c'è il metodo printAll() ma poi, effettivamente, a run-time chiama il metodo di Studente.

## Diagramma UML di sequenza

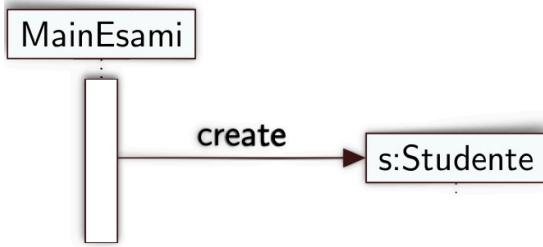
Il diagramma delle classi UML è **STATICO**. Quello che succede a **run-time** viene rappresentato dal **diagramma di sequenza**.

- In questo tipo di diagramma **il tempo scorre verso il basso** (*blu nel disegno sotto*). Quindi gli avvenimenti più recenti sono più in alto.
- In alto vengono messi dei rettangoli che rappresentano le classi sulle quali si vuole rappresentare la situazione a runtime
- Istanza di una classe A** è scritta in un rettangolo in alto --> :A, e vuol dire che c'è un'istanza di tipo A
- Linea della vita è la linea tratteggiata che scende dall'istanza.
- Se si vuole rappresentare un'**interazione** si rappresenta con una linea che arriva in un rettangolo che si sovrappone alla linea tratteggiata e rappresenta che si sta invocando un metodo sull'istanza.
- Il rettangolo viene detto **BARRA DI ATTIVAZIONE** e non indica la durata proporzionata al tempo ma solo la durata rispetto ad altre chiamate di metodo su altre istanza o sulla stessa istanza.
  - Si è attivata l'esecuzione su un'istanza.
- Quando il flusso di esecuzione **"ritorna"** al chiamante si mette una **linea tratteggiata rossa** (*rossa nel disegno*) al contrario che indica il completamento dell'esecuzione. Questa linea si può anche non mettere e il significato è esattamente uguale
- Seguendo questa logica allora due barre di attivazione non saranno mai allo stesso livello.

In questo esempio viene invocato il metodo `m()` sull'istanza di tipo A che, a sua volta, invoca il metodo `m1()` sull'istanza di tipo B.



- Le istanze, seguendo il disegno, di A B e C e voglio mostrare il momento di creazione, posso usare la seguente notazione con `create`:



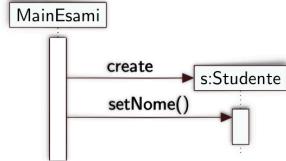
- Dalla classe MainEsami viene istanziata la classe Studente con nome `s` e mostra che in un momento successivo all'esecuzione di MainEsami viene creata l'istanza `s`.

- Esempio di chiamata di metodo su un'istanza di Studente

```

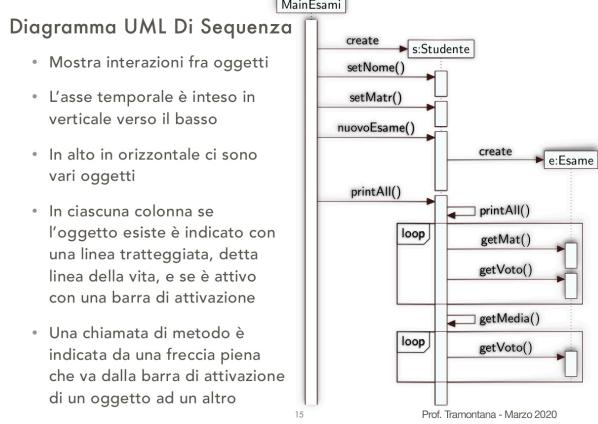
public class MainEsami {
 public static void main(String[] args) {
 Studente s = new Studente();
 s.setNome("Alan", "Rossi"); // chiama metodo di s
 }
}

```



- In questo caso viene chiamato il metodo `setNome()` sull'istanza `s` dalla classe MainEsami

## Un diagramma UML di sequenza completo (con loop)



- Un metodo chiama un altro metodo sulla stessa istanza e ciò viene rappresentato con una freccia tipo "**cappio**". Per esempio il metodo `printAll()` oppure `getMedia()`
- I rettangoli chiamati **loop** serve per dire che ci sono **chiamate allo stesso metodo in un ciclo** per evitare di mettere molte volte la stessa barra orizzontale etichettando il rettangolo con "loop" senza dire il numero di cicli.

Riprendendendo i diagrammi UML di sequenza:

- Le linee **TOCCANO** letteralmente i punti di partenza e di arrivo

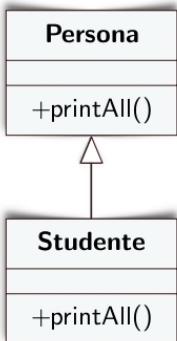
## Late binding e polimorfismo

**Late binding** = collegamento fatto in ritardo (all'ultimo momento)

*Il polimorfismo non può esistere senza late binding.*

Codice di esempio con il relativo UML:

```
public void m() {
 Persona p = new Persona();
 Studente s = new Studente();
 Persona px;
 if (i > 10)
 px = p;
 else
 px = s;
 px.printAll(); // durante la compilazione si verifica che tutte le istruzioni sono
 corrette. Quindi vede inizialmente che px è Persona quindi per lui va bene se invoca
 printAll()
}
```



- `px.printAll()` varia in base all'assegnazione di `px`. Di conseguenza **la stessa linea di codice** può fare la **chiamata alla classe Persona oppure da Studente**. Può variare ad ogni iterazione (per esempio).
- Una riga di codice è **nello stesso punto del codice** e, in base a delle condizioni verificate oppure no, può **invocare metodi di classi diverse**. Questo meccanismo è detto **LATE BINDING** (collegamento fra **chiamante** e **chiamato** viene fatto ogni volta che si presenta la **necessità di saltare a parti di codice diverse**) e per questo si dice che "viene fatto all'ultimo momento". Questo meccanismo permette il **POLIMORFISMO**.
- Quindi un metodo viene invocata **in base all'istanza** che si ha a run-time

*La chiamata del metodo è valutata a runtime.*

Al contrario, **EARLY BINDING** vuol dire che, quando il compilatore incontra `px.printAll()`, mentre si scrive il codice, sa che `px` è di tipo `Persona` a prescindere. A run-time, ovviamente, varia e quindi si passa al late binding. In Early binding non c'è polimorfismo, chiaramente.

## Altro esempio

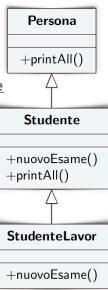
- Nei sistemi ad oggetti possono esistere metodi con lo stesso nome e la stessa signature (in classi diverse)
- Quando si usa l'ereditarietà e sono stati definiti metodi con lo stesso nome, la chiamata ad un metodo può avere effetti diversi, ovvero il comportamento è polimorfo

```
Studente s;
// ...
s.nuovoEsame("Maths", 8);
s.printAll();
```

- La variabile s potrebbe tenere il riferimento ad un'istanza di Studente o di StudenteLavor, quale metodo nuovoEsame() sarà chiamato è deciso a runtime, in base all'istanza. Lo stesso vale quando si ha una variabile di tipo Persona e per il metodo printAll()

17

Prof. Tramontana - Marzo 2020



*Se la OOP non avesse la caratteristica di polimorfismo, per ottenere lo stesso effetto, si dovrebbero fare dei controlli per capire il tipo di istanza e, in base a questo, fare la chiamata corretta di quella istanza*

## Sottoclassi e Dispatch

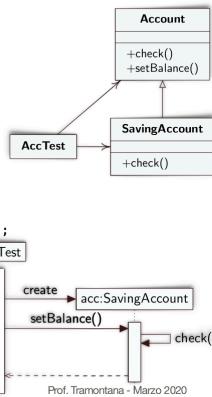
Dato il seguente programma:

```
public class Account {
 protected float balance;
 public void setBalance(float amount) {
 System.out.println("in account set-balance");
 if (check(amount))
 balance = amount;
 }
 public boolean check(float amount) {
 System.out.println("in account check");
 return (balance + amount) >= 0;
 }
}

public class SavingAccount extends Account {
 public boolean check(float amount) {
 System.out.println("in saving-account check");
 return (balance + amount) >= 1000;
 }
}

public class AccTest {
 public static void main(String[] args) {
 Account acc = new SavingAccount();
 acc.setBalance(1234);
 }
}
```

19



Prof. Tramontana - Marzo 2020

- La classe SavingAccount fa **OVERRIDE** del metodo check()
- AccTest interagisce con Account e SavingAccount
- Il metodo check() chiamato è quello di SavingAccount perché c'è *late binding*
- Viene fatta la *chiamata alla sottoclasse* e ciò viene scoperto dal compilatore solamente a runtime e fa chiamate classi che non conosce prima del runtime

Questo meccanismo viene detto **DISPATCH**. Sta a significare che:

- Viene invocato setBalance() su acc.
- A *runtime* acc è di tipo SavingAccount
- A *compiletime* acc è di tipo Account
- Sull'istanza di SavingAccount c'è il metodo setBalance()? No. Allora risalgo la gerarchia finché non si trova tale metodo.

**\*DISPATCH:** cerco il metodo sull'istanza che si ha a runtime nella stessa classe. Se tale metodo non viene trovato allora si risale la gerarchia finché non si trova tale metodo invocato.\*

Riassumendo:

- Quale versione di check() è chiamata da setBalance()?
  - Quando un metodo (setBalance()) è chiamato su un oggetto, viene controllato il suo tipo a runtime (SavingAccount)
  - Si cerca il metodo setBalance() sul tipo a runtime: non trovato
  - Se non trovato, si cerca la superclasse: Account
  - Se trovato si esegue: Account.setBalance()
  - Questo chiama check(), si cerca come prima
  - Quindi si cerca prima su SavingAccount: trovato
  - Si esegue SavingAccount.check()

## Tipi di variabili e tipi a runtime(casting)

Il tipo si può distinguere a compiletime o a runtime.

```
Persona p = new Studente();
Studente s = p; //errore. p è di tipo Persona e non c'è compatibilità fra Studente e Persona
ma fra Persona e Studente
```

Per sistemare forzo il compilatore e dirgli che p è di tipo studente:

```
Studente s = (Studente) p;
```

- Il cast **non cambia il tipo a runtime**
- Viene comunicato al compilatore che a runtime i tipi saranno compatibili

Se invece:

```
Persona p = new Persona();
Studente s = (Studente) p; //è fattibile a compiletime ma a runtime c'è errore
```

In questo caso l'eccezione lanciata è `ClassCastException` e non si può usare nemmeno un `try catch` perché tale eccezione non si può "catturare"

Per questo motivo il **casting** va sempre **evitato** e si fa solo in **rari casi**.

## Design Pattern

I **DESIGN PATTERN** sono **strutture software** (ovvero microarchitetture) per un piccolo numero di classi che descrivono soluzioni di successo per problemi ricorrenti

- Tali micro-architetture specificano le **diverse classi ed oggetti coinvolti** e le loro **interazioni**
- Si mira a **riusare un insieme di classi**, ovvero la soluzione ad un certo problema ricorrente, che spesso è costituita da più di una classe
- Durante la progettazione, le conseguenze sulle classi di varie scelte potrebbero non essere note, e le **classi potrebbero diventare difficili da riusare** o non esibire alcune proprietà. In questo caso viene data anche la soluzione a tali problemi
- Esistono *tanti cataloghi di design pattern*, per vari contesti
- Sistemi centralizzati, concorrenti, distribuiti, real-time, etc

Un design pattern descrive un problema di progettazione ricorrente che si incontra in specifici contesti e presenta una **soluzione collaudata** generica ma specializzabile

- Documentano soluzioni già applicate che si sono rivelate di successo per certi problemi e che si sono evolute nel tempo
- Se il **sistema sviluppato NON** presenta varianti di design pattern allora esso non è la scelta migliore per il proprio sistema ma diventa una **scelta pessima**. *Si cerca di risolvere un problema NON PRESENTE nel software*
- Aiutano i principianti ad agire come se fossero esperti
- Supportano gli esperti nella progettazione su grande scala
- **Evitano di re-inventare concetti e soluzioni**, riducendo il costo
- Forniscono un vocabolario comune e permettono una comprensione dei principi del design
- Analizzano le loro proprietà non-funzionali: ovvero, come una funzionalità è realizzata, es. affidabilità, modificabilità, sicurezza, testabilità, riuso

## Descrizione di un pattern

Un design pattern nomina, astrae ed identifica gli aspetti chiave di un problema di progettazione, le classi e le istanze che vi partecipano, i loro ruoli e come collaborano, ovvero la distribuzione delle responsabilità  
La descrizione include cinque parti fondamentali:

- **Nome**: permette di identificare il design pattern con una parola e di lavorare con un alto livello di astrazione, indica lo scopo del pattern
- **Problema** (Motivazione + Applicabilità): descrive il problema a cui il pattern è applicato e le condizioni necessarie per applicarlo
  - Si parla di **forze** (come in fisica). Se si vuole sapere la traiettoria di un corpo bisogna capire prima le forze che agiscono su questo corpo. Serve per capirne il movimento e la **forza risultante**.
  - Si hanno tante possibili spinte verso le soluzioni del problema e solo l'**analisi di tutte le forze** permette il **raggiungimento di una soluzione**
- **Soluzione**: descrive gli elementi (classi) che costituiscono il design pattern, le loro responsabilità e le loro relazioni
- **Conseguenze**: indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern

## Organizzazione

Ci sono 3 categorie di Pattern:

- Creazionali: riguardano la creazione delle istanze. Per esempio (Singleton, Factory Method, Abstract Factory, Builder, Prototype)
- Strutturali: indicano la struttura del sistema ad oggetti e quali sono le classi per risolvere bene il problema. Per esempio: Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy
- Comportamentali: indicano il comportamento di una struttura. Qual è il metodo migliore per rappresentare il comportamento di una classe. Per esempio Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visitor.
  - Il pattern Template Method l'abbiamo visto con il codice di setBalance() e check(). Si usa un metodo non ancora analizzato a compile time. Quindi check() cambia a runtime che può essere trovato nella stessa classe oppure nella superclasse. Viene chiamato una **PARTE VARIABILE** di codice

# Design pattern Factory Method

C'è un **metodo** in una classe che **produce un'istanza**

## Intento

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

## Problema

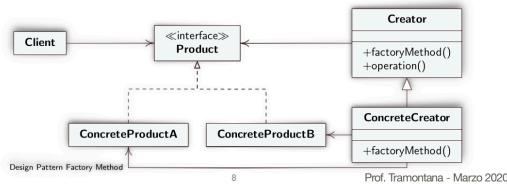
Un **framework usa classi astratte** per definire e mantenere relazioni tra oggetti. Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare

- Non ci sono eseguibili ed è il programmatore che deve scrivere tali programmi.
- Le classi astratte devono lavorare con delle classi che un altro user implementerà e quindi che non si possono istanziare

Un metodo responsabile per l'istanziazione (detto factory, ovvero fabbricatore) incapsula la conoscenza su quale classe creare. Tale metodo verrà usato da un altro sviluppatore per istanziare l'oggetto che a lui interessa.

## Soluzione

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

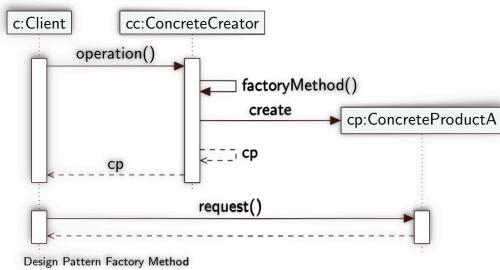


Gli obiettivi di ogni ruolo delle classi del metodo possono essere più di uno per raggiungere lo scopo e la soluzione del problema.

In questo caso ci sono diversi ruoli e ogni classe ne ha uno

- **Product** è un'interfaccia degli oggetti da creare. Il **ruolo Product** serve per capire il tipo di Product e assumerà **varie forme in base all'uso**
- **ConcreteProduct** è un'implementazione concreta di *Product* in base al comportamento da fargli assumere
- La classe *Creator* dichiara il metodo *factoryMethod()* e lo lascia **astratto** e la sottoclasse *ConcreteCreator* implementa questo metodo e quindi decide quale istanza creare
- Il problema parla di framework. Se si vuole incapsulare su quale istanza creare allora è bene usare questo design pattern con la classe *ConcreteCreator* e qui si sceglie se istanziare *ConcreteProductA* oppure *ConcreteProductB*

- Soluzione: diagramma UML di sequenza, che illustra le interazioni fra i vari ruoli



```

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

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

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

```

10 Prof. Tramontana - Marzo 2020

## Considerazioni

- Il Client conosce solo l'interfaccia Product e quindi non conosce se l'istanza è di tipo ConcreteProduct1 oppure ConcreteProduct2 o altro
- Il Client non sa niente riguardo le sottoclassi e nemmeno sa quante ce ne sono
- I ConcreteProduct sono facilmente intercambiabili
- Se nel ConcreteCreator istanzia solo un ConcreteProduct1 allora per istanziare ConcreteProduct2 serve un ConcreteCreator2 con un proprio factoryMethod()
  - Altrimenti se ConcreteCreator è solo uno, nel factoryMethod() si può verificare se c'è una condizione e in base a quella istanziare ConcreteProduct1 oppure ConcreteProduct2

### Esame:

- Vantaggi di factoryMethod()?**
- Se ho l'interfaccia Product che implementa solo un ConcreteProduct, ha senso usare il factoryMethod?**
  - No, si esagera con la progettazione. Ho usato il design pattern factoryMethod che non affronta il problema che tale design affronta

- Varianti
  - Il ruolo Creator e ConcreteCreator sono svolti dalla stessa classe
  - Il factoryMethod() è un metodo static
  - Il factoryMethod() ha un parametro che permette al client di suggerire la classe da usare per creare l'istanza
  - Il factoryMethod() usa la *Riflessione Computazionale*, quindi Class.forName() e newInstance(), per eliminare le dipendenze dai ConcreteProduct, la classe istanziata sarà nota a runtime

```

try {
 Class<?> cls = Class.forName("Studente"); // Il nome della classe da istanziare e' una stringa
 Constructor<?> cnstr = cls.getConstructor(new Class[] {});
 return (IStudente) cnstr.newInstance();
} catch (InstantiationException | IllegalAccessException | IllegalArgumentException | InvocationTargetException | NoSuchMethodException | SecurityException | ClassNotFoundException e)
 e.printStackTrace();
}

```

- Conseguenze
  - Il codice delle classi dell'applicazione conosce solo l'interfaccia Product e può lavorare con qualsiasi ConcreteProduct. I ConcreteProduct sono facilmente intercambiabili
  - Se si implementa una sottoclasse di Creator per ciascun ConcreteProduct da istanziare si ha una proliferazione di classi

- Si passa un parametro al costruttore Creator e in base a questo il factoryMethod() istanzia la classe necessaria
  - **Riflessione Computazionale:** può essere usato dal factoryMethod() ed è una tecnica che consente *di avere codice che prende decisioni sul codice stesso.*
    - Java permette di usare alcune operazioni: indicare, tramite un tipo apposito, "una classe esiste" all'interno di un software. Esiste il tipo "Class" che permette di rappresentare qualsiasi classe. In questo modo si accede al codice del mio software attraverso il tipo `Class`.
    - Posso avere una variabile `Class c;` e su di essa posso usare `Class.forName(" ");` E' statico e restituisce un'istanza di tipo `Class`. Il parametro di `forName` è la stringa che indica il nome della classe che voglio rappresentare.
    - Quindi avrò: `Class c = Class.forName("Studente");` e in questo caso la variabile `c` si comporta come una classe di tipo `Studente`
    - Devo capire se la classe che voglio rappresentare **ha un costruttore** e in tal caso lo posso usare per fare l'istanza.
    - Quando cerco il costruttore devo scegliere i parametri da passargli per decidere quali costruttore invocare. Nell'esempio sto cercando il costruttore che non ha parametri in ingresso. `new Class[] {}`.
  - I parametri vengono specificati fra parentesi graffe. Si userà un array di tipo `Class[]`
  - Sul costruttore `cnstr` si chiama il metodo `newInstance()` che genera **l'istanza**
  - Si fa un **cast** perchè il tipo restituito da `newInstance()` è di tipo `Object` e quindi, essendo programmatore del software, visto che sono partito da `Studente` sono sicuro che mi riferisco all'interfaccia `IStudente`
- Quindi riassumendo si è fatto:
- **Rappresentazione di una classe** a partire da una stringa
  - **Ispezione del codice** di una classe (cerco il costruttore, riflessione computazionale)
  - Creo istanza con `cnstr.newInstance();`
  - Se la classe non è presente nel filesystem ("Studente") allora verrà lanciata un'eccezione (`ClassNotFoundException`) che deve essere gestita

Nel `factoryMethod()` potrebbe non conoscere affatto le sottoclassi di `IStudente` e possiamo verificare tramite questi metodi si potrebbe verificare se determinate classi esistono oppure hanno determinati costruttori.

# Hash Map

HashMap è una mappa associativa, formata da coppie di valori: il **primo valore** è la **chiave di accesso** per il secondo valore

## Operazioni

- Inserimento in tabella: `put(chiave, valore)`
- Trovare il valore in base a una chiave: `valore= tabella.get(chiave)`
- Rimuovere un valore dalla tabella: `valore=tabella.remove(chiave)`

Aver separato Creator e ConcreteCreator introduce il meccanismo Dependency Injection: La classe MainEsami crea istanza di StCreator e la fa conoscere al Client. Il Client si lega a Creator e non a StCreator. In particolare:

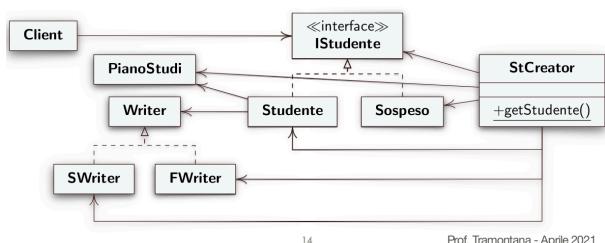
```
public class MainEsami{
 public static void main(String[] args){
 Creator crea = new StCreator();
 Clienc c = new Client(crea);
 }
}
```

- Creo una sola istanza di StCreator e la passo al Client
- Si separa la decisione sulla costruzione dall'uso di quell'istanza
- Il client non deve sapere le istanze create ma deve conoscere solo l'interfaccia. (incapsulamento)
- Le dipendenze sono iniettate al client per mezzo di parametri nel suo costruttore. Questo permette di evitare complicazioni derivanti da metodi setter e da controlli per verificare che le dipendenze non siano null, di conseguenza il codice è più semplice
- L'oggetto che fa Dependency Injection si occupa di connettere (*fa wiring di*) varie istanze. In un unico posto vediamo le connessioni fra gli oggetti.

## Esempio

- Si abbiano Writer e PianoStudi che sono dipendenze per Studente
- Studente riceve nel suo costruttore le istanze di Writer e PianoStudi
- Studente conosce solo il tipo Writer non i suoi sottotipi

La classe non crea internamente le dipendenze di cui ha bisogno ma le acquisisce esternamente (gli vengono iniettate dall'esterno) per esempio come parametri formali



# Abstract Factory

## Intento

Fornire un'interfaccia per creare famiglie di oggetti che hanno qualche relazione senza specificare le loro classi concrete

- Si hanno più gerarchie di Product e ConcreteProduct.
- Product è un'interfaccia totalmente diversa da factory method
- Quindi ho **più interfacce** di Product:
  - un'interfaccia **ProductB** che implementa **ConcreteProductB1** e **ConcreteProductB2**
  - un'interfaccia **ProductA** che implementa **ConcreteProductA1** e **ConcreteProductA2**
- Quando seleziono **ConcreteProductA2** è giusto avere una **corrispondenza** con **ConcreteProductB2** (o anche altre relazioni)
  - Quando uso **B2** mi relaziono solo a **A2** o viceversa e non relazionarsi a **A1** o **B1**

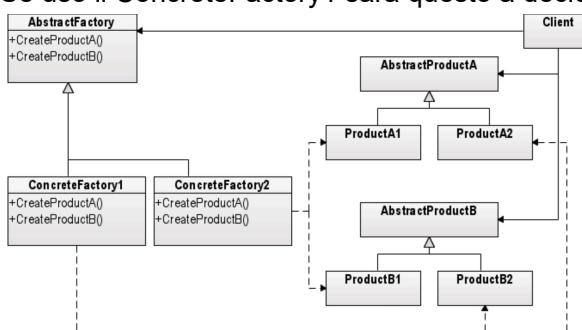
## Problema

- Il sistema complessivo dovrebbe essere indipendente dalle classi usate, così da essere configurabile con una di varie famiglie di classi. Le classi di una famiglia dovrebbero essere usate insieme (in modo consistente)

*| Es. lo strato di interfaccia utente permette vari tipi di look-and-feel*

## Soluzione

1. Creare delle gerarchie con interfacce **AbstractProductB** e **AbstractProductA**
2. Gestire la parte di creazione di istanze **AbstractFactory** e **ConcreteFactory**
3. Se uso il **ConcreteFactory1** sarà questo a decidere la consistenza.



Usare una **serie di ruoli**: **AbstractFactory**, **ConcreteFactory**, **AbstractProduct**

- Si possono creare istanze di famiglie di classi in maniera consistente.
- Le famiglie di classi sono facilmente intercambiabili.
  - Se voglio eliminare una famiglia di **AbstractProduct** e cambiare (Per esempio un bottone cliccabile) in un bottone che scompare, allora posso inserire una nuova famiglia con un'interfaccia. In questo caso devo solo modificare il **ConcreteFactory** far istanziare le nuove classi create.

Maggiori informazioni su [Wiki](#)

*| Se volessi, quindi, **AbstractProductC** basta fare le operazioni sopra citate, quindi creare una nuova gerarchia.*

- **Uso di classi di famiglie diverse corrisponde a diverse famiglie di classi.**

## Object Pool

Si ha bisogno di tenere le istanze (in una lista) create a run-time per riusare la stessa istanza in circostanze diverse. Questa tecnica è detta **Objcet Pool**

*Quando si finisce di usare un'istanza, potrebbe venir restituita (venendo liberata) e riutilizzata.*

Utile per:

- Creazione di istanze può essere **pesante a livello computazionale**.
- Quando le istanze sono "libere" allora possono essere fornite a chi vuole una nuova istanza
- Si evita di creare altri spazi di memoria

Serve:

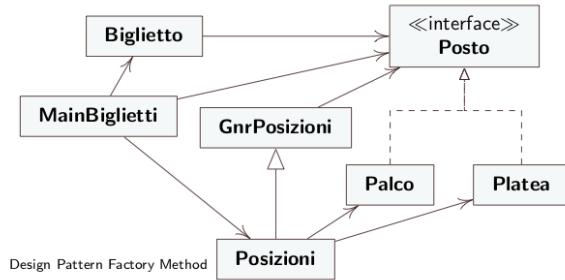
- Un posto per tenere le istanze (lista delle istanze create) messa in una classe che ha appunto tale ruolo (**all'interno del factoryMethod**).

```
import java.util.LinkedList;
// CreatorPool è un ConcreteCreator e implementa un Object Pool
public class CreatorPool extends ShapeCreator {
 private LinkedList<Shape> pool = new LinkedList<Shape>();
 // getShape() è un metodo factory che ritorna un oggetto prelevato dal pool
 public Shape getShape() {
 Shape s;
 if (pool.size() > 0) s = pool.remove();
 else s = new Circle();
 return s;
 }
 // releaseShape() inserisce un oggetto nel pool
 public void releaseShape(Shape s) {
 pool.add(s);
 }
}
```

E: Tramontana - Pool, Dependency - 12 May 10 | 2

- Queste tecnica può **non avere limiti** oppure **può averne** (in base ai requisiti)
-

## Esempio di Factory Method



```
// Codice Java che implementa il design pattern Factory Method
```

```
// Posto e' un Product
public interface Posto { // è tipo una matrice che avrà la coppia (lettera,riga) oppure
 (numero)
 public String getPosizione();
 public int getCosto();
 public String getSettore();
}
```

```
// Palco e' un ConcreteProduct
import java.util.Random;

public class Palco implements Posto {
 private final int numero;

 public Palco() {
 numero = new Random().nextInt(20) + 1;
 }

 @Override
 public int getCosto() {
 if (numero > 10) return 50;
 return 40;
 }

 @Override
 public String getPosizione() {
 return Integer.toString(numero);
 }

 @Override
 public String getSettore() {
 if (numero == 20) return "Centrale";
 if (numero > 10) return "Verde";
 return "Blu";
 }
}
```

```
// Platea e' un ConcreteProduct
import java.util.Random;
```

```

public class Platea implements Posto {
 private final String[] nomi = { "A", "B", "C", "D", "E", "F" };
 private final int numero;
 private final int riga;

 public Platea() { //indica un posto ben preciso.
 numero = new Random().nextInt(10) + 1;
 riga = new Random().nextInt(5) + 1;
 }

 @Override
 public int getCosto() { //fasce di costo diverso a seconda della posizione
 if (numero > 5 && rigaMax()) return 100;
 if (numero > 5 && rigaMin()) return 80;
 return 60;
 }

 @Override
 public String getPosizione() {
 return nomi[riga].concat(Integer.toString(numero));
 }

 @Override
 public String getSettore() {
 if (riga == 0) return "Riservato";
 return "Normale";
 }

 private boolean rigaMax() {
 return (riga >= 1 && riga <= 4);
 }

 private boolean rigaMin() {
 return (riga == 0 || riga == 5);
 }
}

```

```

// GnrPosizioni e' un Creator
import java.util.Set;
import java.util.TreeSet;
public abstract class GnrPosizioni { // versione 1.1
 private static final int MAXPOSTI = 100;
 private final Set< String > pst = new TreeSet< >(); // set di posti -> Il controllo
è più veloce rispetto a una List. L'ordine non è importante e NON CI SONO DUPLICATI

 // metodo che rimanda alla sottoclassse l'istanziazione della classe
 public Posto prendiNumero(int x) {
 if (pst.size() == MAXPOSTI) return null;
 // il chiamante dovrebbe controllare se null
 Posto p;
 do { // fino a quando non trova un posto libero
 p = getPosto(x); // chiama metodo della sottoclassse
 } while (pst.contains(p.getPosizione()));
 pst.add(p.getPosizione());
 return p;
 }
}

```

```

public void printPostiOccupati() {
 for (String s : pst)
 System.out.print(s + " ");
}

// il metodo factory e' dichiarato ma non implementato
public abstract Posto getPosto(int tipo);
}

```

```

// Posizioni e' un ConcreteCreator con un metodo factory
public class Posizioni extends GnrPosizioni {
 // metodo factory che ritorna una istanza
 @Override
 public Posto getPosto(int tipo) {
 // crea l'istanza di un ConcreteProduct
 if (1 == tipo) return new Palco();
 return new Platea();
 }
}

```

```

// Biglietto e' un client del Product Posto
public class Biglietto {
 private String nome;
 private final Posto pos;

 public Biglietto(Posto p) {
 pos = p;
 }

 public void intesta(String s) {
 nome = s;
 }

 public String getDettagli() {
 return nome.concat(" ").concat(pos.getPosizione());
 }

 public String getNome() {
 return nome;
 }

 public int getCosto() {
 return pos.getCosto();
 }
}

```

```

// Classe con il main che usa il ConcreteCreator
public class MainBiglietti {
 private static Posizioni cp = new Posizioni();

 public static void main(String[] args) {
 Posto pos = cp.prendiNumero(0);
 Biglietto b = new Biglietto(pos);
 b.intesta("Mario");
 }
}

```

```

 System.out.println("Costo " + b.getCosto());

 new Biglietto(cp.prendiNumero(0));
 new Biglietto(cp.prendiNumero(0));
 cp.printPostiOccupati();
 }
}

```

`@Override` è detta **ANNOTAZIONE**. Devono stare in posti ben precisi: subito prima della dichiarazione/implementazione di un *metodo/classe/passaggio di parametri nei metodi*. Sono:

- predefinite da JAVA
- create dall'utente

`@Override` è **predefinita**. Serve per evitare errori in fase di pre-compilazione.

A run-time le annotazioni potrebbero essere state eliminate dal compilatore. Servono solo come informazioni aggiuntive ma non servono a run-time. Servono per fare in modo che il compilatore faccia **un controllo in più** o evita di segnalare errori dove effettivamente non ce ne sono. Assicura che effettivamente sto facendo un override e sto definendo un metodo della superclasse.

*Vuol dire che il metodo deve essere stato dichiarato nell'interfaccia che è super-tipo della classe che si sta implementando*

## Design pattern Adapter (object adapter e class adapter)

Questo è un design pattern **STRUTTURALE**. I precedenti visti erano **CREAZIONALI**.

### Intento

Converte l'interfaccia di una classe in un'altra interfaccia che si aspetta il client. Il client si aspetta una certa interfaccia (un certo nome di metodo con un certo tipo di parametro in ingresso). Quindi c'è **INCOMPATIBILITÀ** fra la richiesta del client e l'effettiva implementazione.

### Problema

Si è progettato un sistema software facendo riferimento a una certa interfaccia da usare.

Si è implementato, per esempio, un metodo `getCosto()` ma effettivamente serve un `getCosto( parametro )`.

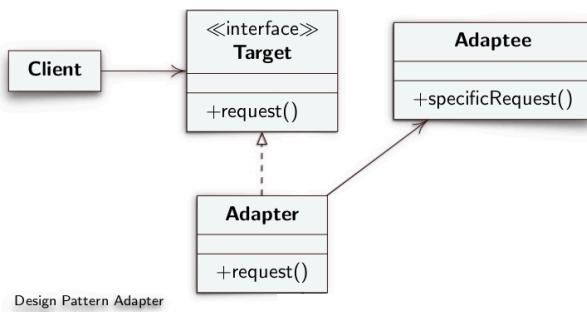
- La parte che deve fornire il risultato (il chiamante) **NON può essere modificata**.
- Cambiare l'interfaccia in modo da adattarsi ai requisiti potrebbe creare problemi se tali metodi sono usati anche altrove.
  - **Se modifico un metodo**, questo stesso deve venire modificato in modo da attenersi all'attività che deve svolgere.

### Esempio

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

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

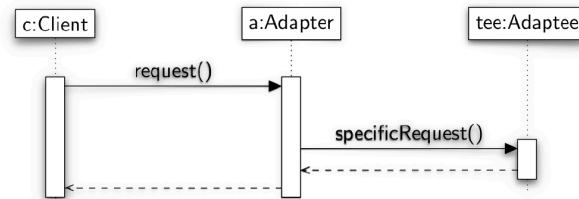
## Soluzione



Crea una nuova classe Adapter che farà da adattatore fra invocazioni del client e server.

- I metodi saranno **conformi al client (chiamante)**.
- La classe che mette a disposizione i metodi saranno conosciuti all'interno della classe Adapter. Quindi da Adapter si chiamano i metodi effettivi.
- Il client si interfaccia con **Target** che è la classe che si aspetta
- Adaptee sarà la classe di libreria che mette a disposizione il servizio che risulta **INCOMPATIBILE** con il Client.

**(ESAME): Pronunciare bene il nome delle classi per evitare fraintendimenti. -> Adapter e Adaptee (Adaptii).**



```

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

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

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

```

```

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

<<interface>>
Label
+getNextLabel()

LabelServer
+serveNextLabel()

```

```

Client -->> Label : +main()
Client -->> Label : create >> s:Label
Label -->> LabelServer : +getNextLabel()
Label -->> LabelServer : create >> ls:LabelServer
LabelServer -->> Label : +serveNextLabel()

```

Prof. Tramontana - Marzo 2019

Si crea un **Adapter** per ogni classe che devo adattare.

Se ho 2 librerie da adattare allora ho 2 **Adapter** diversi.

- opt sta a significare che, quando si deve istanziare ls non è detto che essa venga fatta perché c'è un'istruzione condizionale e quindi potrebbe non venire fatta.
- Questo è un **Object Adapter** perchè si ha l'**invocazione fra oggetti**

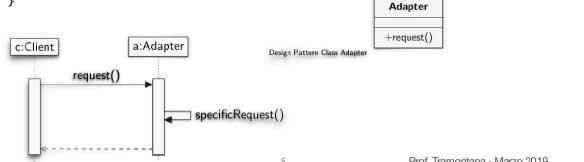
## Variante Class Adapter

- Adapter è sottoclasse di Adaptee.
- Adapter implementa l'interfaccia Target (*come prima*)
- Si possono liberamente invocare i metodi della superclasse e quindi non serve più un riferimento ad Adaptee (un oggetto Adaptee)
- Quando devo invocare un metodo di Adaptee, non serve più l'istanza ma invoco il metodo sulla **stessa istanza**

- Soluzione Class Adapter

- Adapter è sottoclasse di Adaptee

```
public class Label extends LabelServer implements ILabel { // Adapter
 public Label(String prefix) {
 super(prefix);
 }
 public String getNextLabel() {
 return serveNextLabel();
 }
}
```



Design Pattern Class-Adapter

Prof. Tramontana - Marzo 2019

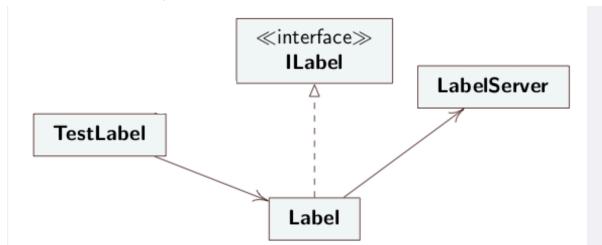
## Differenze Object Adapter e Class Adapter

- Dipendono dai requisiti, ovviamente, e dalle scelte di progettazione
- si crea un'unica istanza di Adapter che **contiene tutto**. Se la superclasse richiede molta memoria allora si inizializza anche Adaptee allo stesso momento e quindi si spende quel tempo di inizializzazione (*a prescindere*)
- In Object Adapter, invece, **rimando l'inizializzazione fino a poco prima dell'invocazione del metodo**. (quindi il più tardi possibile, in modo da non far perdere tempo al Client)
  - Adaptee potrebbe non essere mai inizializzato (in base all'uso del software) e quindi si **risparmia tempo**. Questo *modo di fare* è detto **LAZY INITIALISATION** (cioè **inizializzazione in ritardo, pigra**)
- **Ci sono MOMENTI DIVERSI di inizializzazione**

In Java non è concessa l'ereditarietà multipla per semplificare la scrittura del codice e per evitare incompatibilità fra metodi ereditati da una classe **A** o **B** (**aventi la stessa signature**)  
**Si può, invece, ereditare da una classe A e da un'interfaccia B.**

## Variante Adapter a due vie

Fornisce sia l'interfaccia di Target e sia l'intefaccia di Adaptee. Se un client si riferisce a due interfacce. Usando il object adapter uso questa versione e al suo interno istanzio Adaptee (*LabelServer*)



- Il **Client** deve invocare i **metodi dell'intefaccia Target e i metodi dell'interfaccia Adaptee**
- **Client** si interfaccia con **Adapter** essendo più "ricca" di informazioni rispetto a **Target**

## Conseguenze

- Conseguenze del design pattern Adapter
- Client e classe di libreria Adaptee rimangono indipendenti. L'Adapter può cambiare il comportamento dell'Adaptee
- Può aggiungere test di precondizioni e postcondizioni [Precondizioni: cosa si deve soddisfare prima di eseguire. Postcondizioni: cosa è verificato se tutto è andato bene]
- L'Object Adapter può implementare la tecnica di Lazy Initialization
- Il design pattern Adapter aggiunge un livello di indirezione. Ogni invocazione del client ne scatena un'altra fatta dall'Adapter. Possibile rallentamento (trascurabile), e codice più difficile da comprendere

6

Prof. Tramontana - Marzo 2019

- **PRECONDIZIONE:** deve verificarsi prima dell'invocazione di un metodo per consentire al metodo di essere **BEN ESEGUITO**.
  - Se queste precondizioni possono essere scritte, **evito** di fare chiamate di metodo che molto probabilmente **incontra errori di esecuzione** o non ritorna il dato voluto.
  - Le precondizioni devono essere individuate durante la progettazione. Vengono messe nel Adapter per non farlo fare al Client.
  - Se si vuole semplificare l'Adaptee, tali controlli si fanno sempre nell'Adapter
- **POSTCONDIZIONI:** si devono verificare **se la condizione è andata a BUON FINE**.
  - *Per esempio, un valore di ritorno non deve essere mai nullo oppure deve essere di un certo tipo.*

L'Adapter introduce un livello di dipendenza: ogni volta che viene usato, esso chiama un metodo da Adaptee. Non si passa da chiamante->chiamato ma si aggiunge qualcosa **in mezzo**. In questo caso, a livello di leggibilità di codice, essa viene meno perché magari Adapter non è nemmeno documentato perchè fa solo da "tramite".

# Design Pattern Facade

## Intento

Fornire un'interfaccia **unificata** e **semplificata** al posto di un **insieme di classi** (*che si vogliono nascondere*), detto **SOTTOSISTEMA** (*di norma si usano al massimo 10 classi*).

I vari Client useranno il Facade piuttosto che il sottosistema

## Concetto di interfaccia

Per interfaccia si possono intendere 2 significati:

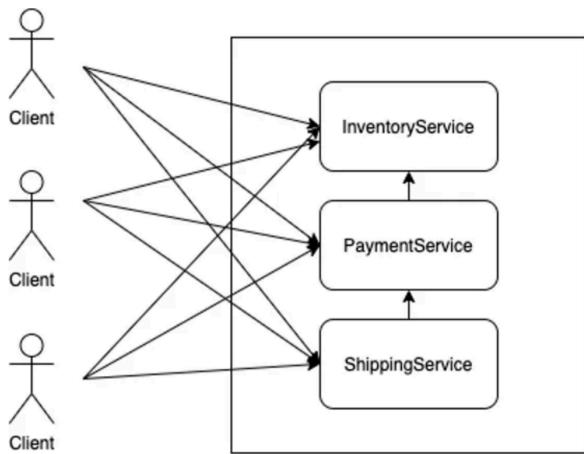
- [Concetto in Facade] **punto di contatto** fra una parte e un'altra. Una componente software ne vuole usare un'altra
- [Concetto Generico] un tipo di Java che non offre implementazione di metodi)

Per *Facade* si intende *Facciata*, cioè il punto che si osserva all'esterno rispetto a tutto quello che c'è all'interno.

## Problema

Spesso ci sono varie classi che, poichè devono essere **piccole** (*ognuna ha un proprio ruolo*), e il *Client* deve incorrere a più classi per ottenere il risultato voluto. Quindi **interagisce con MOLTE classi** e ognuna di esse non fornisce informazioni *corpose*.

- In Client si devono dichiarare almeno un **tot di variabili** per ogni classe
- Per poter **comprendere il Client**, bisogna **comprendere tutte le classi con le quali interagisce**



I Client interagiscono con molte piccole classi (InventoryService, PaymentService, ShippingService)

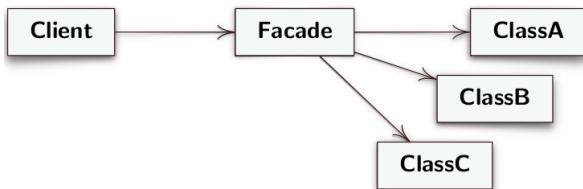
La stessa cosa avviene anche se c'è un Client2 (anche se fa cose diverse rispetto a Client)

Si ha **meno comprensibilità del codice** e **meno modularità** di esso

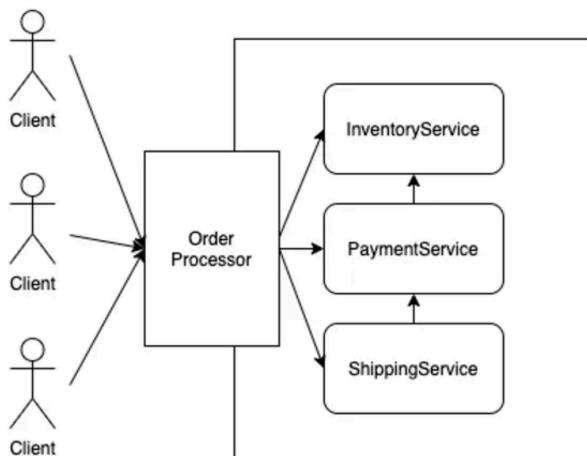
## Soluzione

Mettere *in mezzo* un *Facade*.

I vari client conosceranno solo *Facade* ed esso conoscerà il sottosistema di classi



- Con questa soluzione il codice **acquista modularità** e viene **ridotta la complessità** (il *Client* non usa più tutte le classi del sottosistema come variabili ma ne utilizza solo una di *Facade*)
- Il *Facade* deve offrire un'interfaccia **semplificata** quindi non ricopia le sottoclassi
- Il *Facade* offre un metodo **più di alto livello** rispetto alle sottoclassi. Si va a capire cosa serve al *Client* e si crea un **metodo apposito** per fargli avere quello che effettivamente desidera
- Viene **SEMPLIFICATO** anche in *Client*
- Il *Facade* ha **nomi di metodi DIVERSI** rispetto ai metodi del sottosistema, appunto perché si adatta alle richieste del *Client*



*I Client interagiscono solo con il Facade(OrderProcessor)*

Lo **stesso discorso** vale per un eventuale *Client2*: si progetta un'interfaccia più potente rispetto al sottosistema apposito per *Client2*.

## Conseguenze

- Il *Client* non si deve preoccupare del sottosistema ma fa un'unica richiesta di più alto livello.
- La **dipendenza** fra sottosistema-client diventa **inesistente** visto che il *Client* dipenderà solo da *Facade*
- Il sistema è più **MODULARE** perché se si dovesse sostituire *ClassA* con un'altra classe, la **sostituzione** fa sì che l'unica classe che dovrà essere cambiata sarà *Facade* visto che è l'unica che interagisce con essa
- Le **INTERAZIONI** sono **ridotte** e la propagazione di modifiche viene meno, appunto
- A *livello di compilazione*, se si **modifica una classe**:
  - *senza Facade*: devo ricompilare **tutto il sistema** che interagiva con quella classe
  - *con Facade*: devo ricompilare **solo il Facade** e la *classe modificata*

E' possibile che esista un *Client* (come eccezione) che voglia accedere solo a una determinata classe definita in un *Facade*. Per cui interagisce con *Facade* e con la classe del sottosistema che gli interessa. Questo accade in maniera rara visto che, altrimenti, verrebbe reintrodotto il problema che questo design pattern tende a risolvere.

- A livello di codice nulla vieta a *Client* di usare classi del sottosistema visto che le classi sono `public` e si trovano nella stessa cartella del progetto.
  - Per risolvere e nascondere questa problematica si possono usare le **CLASSI ANNIDATE**, cioè *classi dentro le classi*. A questo punto le classi del sottosistema non potranno più essere utilizzate visto che faranno parte di *Facade*.
  - Tale operazione è consentita fino a un certo numero di linee di codice (*senza esagerare*)
  - Quindi, le classi che usa il *Facade* vengono implementate direttamente al suo interno così che non vengano viste dal *Client*

## Esempio di Facade (classi annidate)

```

public class Client {
 public static void main(String args[]){
 // English e; Errore di compilazione
 // Italian i; Errore di compilazione perchè la classe non è visibile
 Translator t = new Translator();
 t.addEnglish("Hello");
 t.multiPrinting();
 }
}

public class Translator{// Ruolo Facade
 private English e = new English();
 private Italian i = new Italian();
 public void addEnglish(String s){
 if (e.test(s)){
 e.add(s);
 i.add(e.getIndex(s));
 }
 }

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

 //classe annidata
 public class English {
 private String text = " ";
 private List<String> d = Arrays.asList("Alright", "Hello", "Understood",
 "Yes");

 public boolean test(String s) {
 return d.contains(s);
 }

 public void add(String s){
 text = text + " " + s;
 }

 public String getText() {
 return text;
 }
 }
}

```

```

 }

 public int getIndex(String s) {
 return d.indexOf(s);
 }

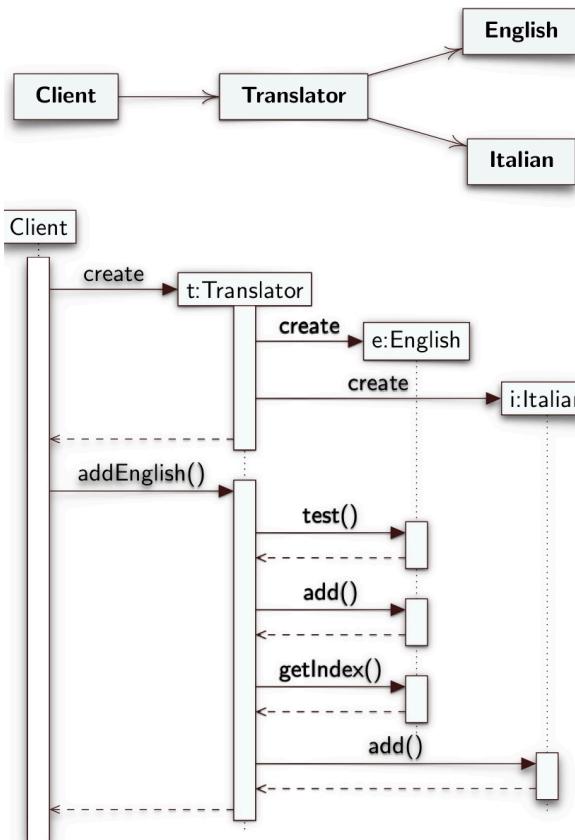
 public void printText(){
 System.out.println(text);
 }
 }

//classe annidata
public class Italian {
 private String text = " ";
 private List<String> d = Arrays.asList("Va bene", "Ciao", "Capito", "Si");

 public void add(int i){
 text = text + " " + d.get(i);
 }

 public void printText() {
 System.out.println(text);
 }
}
}

```



**(esame)Disegnare un PRIMA-DOPO in seguito all'applicazione di Facade**

Secondo esempio di Facade

# Design Pattern State

## Intento

Fornire una struttura in modo tale che, quando si hanno **oggetti che devono modificare il loro comportamento in base al loro stato**, è come se si cambiasse l'intera classe. Lo **STATO** è rappresentato dai valori degli attributi (o da una loro **combinazione**).

Se si ha uno **stato** si deve fare una cosa. Se si ha uno **stato diverso** deve fare altro. Quindi il comportamento della classe si basa sui valori che assumono gli attributi.

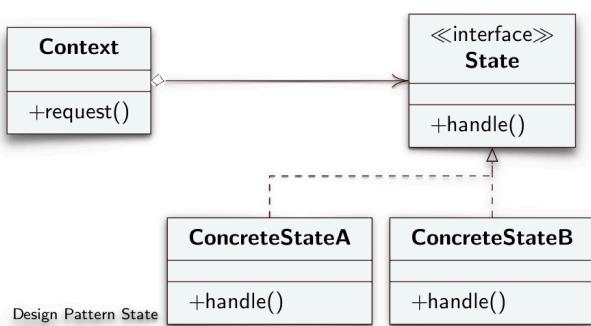
Si deve, inoltre, trattare tutto in maniera **MODULARE**.

## Problema

- Il comportamento di un oggetto **dipende dal suo stato** e si vuole che esso **cambi a run-time**. Quando ci sono **MOLTI rami condizionali** (da mettere su **ciascuna operazione che si implementa**) e ogni ramo ci sono varie operazioni, allora in tal caso **si evidenzia il problema** che questo *Design Pattern* tratta
- Quando si controllano le condizioni in maniera costante, *le variabili che rappresentano lo stato* si possono anche considerare **porzioni di variabili**, cioè uno stato non è sempre rappresentato da tutte le variabili.
- Spesso **varie operazioni** contengono la **STESSA** struttura condizionale

## Soluzione

- Si separa ogni ramo condizionale in una classe vera e propria, generando il `ConcreteState` che definisce il comportamento di uno stato specifico in base allo *stato*.
- Serve un'interfaccia `State` che definisce le operazioni di ogni `ConcreteState`.
- Context conosce quale `ConcreteState` si devono usare, tiene il riferimento a un'istanza di `ConcreteState` che rappresenta lo **stato attuale**. Se si deve cambiare stato, allora è il Context che lo cambia e istanzia un altro `ConcreteState`.
  - Inoltre definisce anche l'interfaccia che interessa ai *Client* e tutte le condizioni per ogni possibile cambiamento di stato.
- Ogni `ConcreteState` non è a conoscenza del fatto che ci possono essere più di uno `ConcreteState`



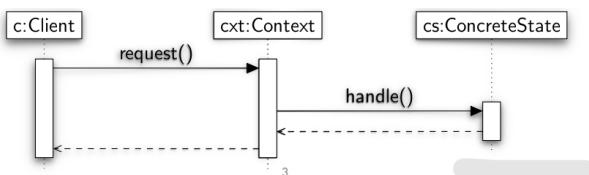
- `request()` è sempre nel *Context* e riguarda sempre i requisiti del Client ed è spesso usato questo nome di metodo visto che riguarda proprio le richieste previste dai requisiti

- La *request* viene **implementata** in una serie di richieste ad uno specifico *ConcreteState* con il metodo `handle()`

Nella freccia che collega **Context -> State**, c'è un **DIAMANTE** (un rombo) che indica l'**aggregazione**: per poter definire bene l'oggetto che si sta aggregando (*Context*) allora servono vari oggetti che offrono dei servizi e che sono aggregati (*ConcreteState*)

I vari *ConcreteState* **NON** devono essere conosciuti dal *Client*

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

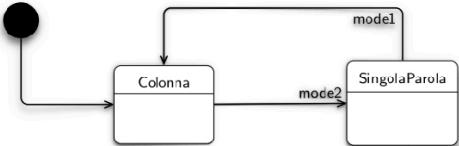
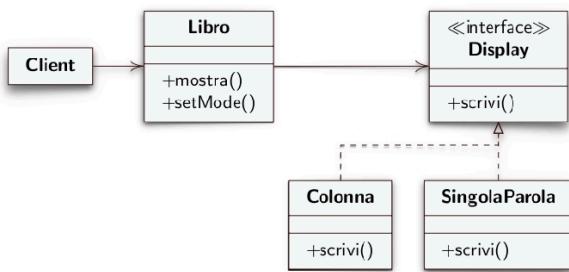


## Conseguenze

- Il comportamento di ogni singolo stato è **ben separato** in una classe **distinto da altri comportamenti**
- Il partizionamento permette di vedere ogni classe **più comprensibile e più semplice da modificare o correggere gli errori.**
  - In caso di **più operazioni** in un **unica classe**, risulta **molto più facile confondere l'implementazione delle varie operazioni che non sono distinte in stati diversi** visto che si devono aggiungere degli **if addizionali**
- L'**operazione di transizione** di stato è delegato a *Context* con i metodi `modeColonna` e `modeParola`.
- Se serve un **nuovo stato** basta **aggiungere** un nuovo *ConcreteState*
- Il codice **si può testare facilmente** e molto riusabile (**principio di singola responsabilità**)

## Esempio (e DIAGRAMMA UML DEGLI STATI)

Si vuole visualizzare un libro con **visualizzazione a colonna** oppure a **singola parola** e ciò dipende dalla decisione del *Client*. *Libro* può passare da un modo di visualizzazione ad un'altra.



Il diagramma nella parte sottostante è detto **Diagramma UML degli STATI**:

- Il rettangolo con bordi arrotondati indica lo **STATO**
- La freccia che collega gli stati indica una **TRANSIZIONE** fra 2 stati diversi e la freccia indica lo **stato di partenza** e lo **stato di arrivo**. L'**evento** che si deve verificare (`mode2`) per passare da uno stato ad un altro è scritto sopra la freccia. La transizione verrà fatta da `Context`
- La **Palla Nero** indica da quale stato si parte, quindi indica **LO STATO INIZIALE**
- Sotto la linea orizzontale di ogni stato si possono scrivere le **azioni da fare** quando si **arriva nello stato (entry)**, quando si **persiste nello stato(do)** e quando si **lascia quello stato(exit)**
  - Se queste **azioni non sono presenti (non richieste)** allora si fa il disegno in questo come in figura oppure è possibile omettere la linea orizzontale (che è disegnata sopra) e scrivere il nome dello stato **AL CENTRO DEL RETTANGOLO**

```

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

public class Colonna implements Display { // ConcreteState
 private final int numCar = 38;
 private final int numRighe = 12;
 public void scrivi(List<String> testo) {
 int riga = 0;
 int col = 0;

 for (String p : testo) {
 if (col + p.length() > numCar) {
 System.out.println();
 riga++;
 col = 0;
 }

 if (riga == numRighe) break;
 System.out.print(p + " ");
 col += p.length() + 1;
 }
 }
}

public class SingolaParola implements Display { // ConcreteState
private int maxLung;
public void scrivi(List<String> testo) {
}

```

```

System.out.println();
mettiSpazi(30);
trovaMaxLung(testo);

for (String p : testo) {

int numSpazi = (maxLung - p.length()) / 2;
mettiSpazi(numSpazi);
System.out.print(p);

if (p.length() % 2 == 1) numSpazi++;
mettiSpazi(numSpazi);

aspetta();
cancellaRiga();
}
System.out.println();
}

private void mettiSpazi(int n) {

for (int i = 0; i < n; i++) System.out.print(" ");
}

private void cancellaRiga() {

for (int i = 0; i < maxLung; i++) System.out.print("\b");
}

private void trovaMaxLung(List<String> testo) {

for (String p : testo) if (maxLung < p.length()) maxLung = p.length();
}

private static void aspetta() {

try {
Thread.sleep(300);
} catch (InterruptedException e) { }
}
}

public class Libro { // Context
private String testo = "Darwin's _Origin of Species_ persuaded the world that the "
+ "difference between different species of animals and plants is not the fixed "
+ "immutable difference that it appears to be.";
private List<String> lista = Arrays.asList(testo.split("[\\s]+")); //da una stringa "testo"
si può avere una lista di stringhe, dove ognuna di esse è divisa secondo un'espressione
regolare regex: in questo caso l'elemento separatore fra stringhe è uno o più spazi
([\\s]+)
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;
}
}
}

```

*Il client vede la stessa Classe Libro ma al suo interno le classi cambiano in base alla modalità selezionata*

Non sarebbe sbagliato avere più di un metodo per cambiare modalità in context visto che si potrebbero avere necessità diverse per client diversi (*non bisogna esagerare e quindi non usare troppi metodi*)

## Serie di divieti da seguire in ingegneria del software

- Non avere metodi lunghi (*righe/istruzioni > 15 circa*)
- Il client non deve usare `new` ovunque, cioè non deve essere a conoscenza dell'implementazione dei vari codici. Le **dipendenze** devono essere verso **le interfacce e non verso le implementazioni**
- **Non usare condizioni** (come nello *State*) o non usarne *TROPPE* e al loro posto implementare il **polimorfismo**
- Non usare i cicli: si fanno cose alternative

## Refactoring verso State

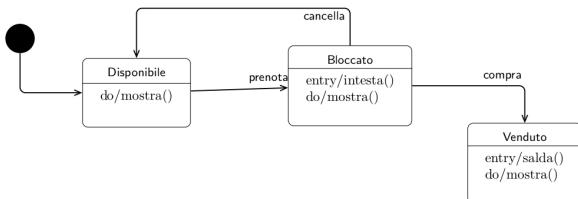
### Requisiti

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

### Progettazione

- Il sistema software dovrà fornire la possibilità di **prenotare** e **acquistare** un **biglietto** (per un viaggio). Si potrà **annullare** la prenotazione, ma non l'acquisto. Ogni biglietto ha un **codice**, un **prezzo**, una **data** di acquisto, il **nome** dell'intestatario (e i dettagli del viaggio). Per la prenotazione si dovrà dare il nome dell'intestatario.
- **Classi:** Biglietto
- **Attributi:** codice, prezzo, data, nome
- **Operazioni:** prenota, acquista, annulla
- La classe Biglietto si può trovare in uno degli stati: disponibile, bloccato (ovvero prenotato), venduto

### Diagramma degli stati



- Quindi, la classe Biglietto dovrà implementare i metodi: prenota, cancella, compra, mostra.
- Ciascuna operazione controllerà lo stato in cui si trova il biglietto prima di eseguire le azioni necessarie

```
// Codice che implementa i suddetti requisiti (prima versione)

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

 // ogni operazione deve controllare in che stato si trova il biglietto
 public void prenota(String s) {
 switch (stato) {
 case DISP:
 System.out.println("Cambia stato da Disponibile a Bloccato");
 nome = s;
 System.out.println("Inserito nuovo intestatario");
 stato = StatoBiglietto.BLOC;
 break;
 case BLOC:
 nome = s;
 System.out.println("Inserito nuovo intestatario");
 break;
 case VEND:
 System.out.println("Non puo' cambiare il nome nello stato Venduto");
 break;
 }
 }
}
```

...>Vedi slide>...

- Ogni metodo ha 3 rami condizionali che dipendono dallo stato del biglietto (*venduto*, *disponibile* o *bloccato*)

\*Si trasforma il codice presente in codice che usa il design pattern State, facendo **refactoring**\*

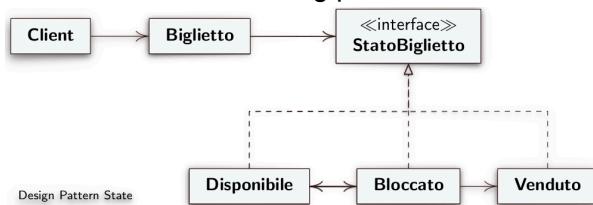
# Refactoring

- Dopo aver scritto il codice, si **riguarda il codice** e si **aggiungono dettagli alla descrizione** o si va a capire meglio la descrizione presente.
- Si va a **perfezionare la progettazione** in modo da poter aggiungere nuove cose in maniera semplice: si **cambiano nomi, metodi ecc lasciando invariato il comportamento**.
- La **nuova versione risulta più appropriata** per incorporare i nuovi requisiti aggiunti

Esiste una tecnica di refactoring chiamata **Replace Conditional with Polymorphism** che è una tecnica usata anche fuori dallo State, quindi **per altri scopi**.

Quando si hanno dei **tipi enumerativi**, è **molto probabile** che serva il **design pattern State**

L'idea è di fare refactoring per arrivare ad una struttura del tipo:



**(Esame)** dati alcuni frammenti di codice significativi quindi un piccolo sistema software, individuare il design pattern e completarlo. Oppure ->Dato un codice con un design pattern, completare il design pattern.

Dal codice, fare una rappresentazione UML con l'ereditarietà

Si vedono i vari casi condizionali e, nei vari ConcreteState si vanno a mettere solo quei metodi dove vengono controllati i casi che cambiano.

- Nel caso della classe Disponibile, si vanno a prendere quei **pezzi di codice** che **coinvolgono lo stato DISP** del Biglietto.
- Il metodo `mostra()` in Disponibile si deve implementare forzatamente perché si usa una classe astratta, anche se esso risulta vuoto.

```
// Codice Java che implementa il design pattern State
```

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

```
// Disponibile e' un ConcreteState
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;
}
}

```

```

// Bloccato e' un ConcreteState
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();
 }
}

```

```

// Venduto e' un ConcreteState
import java.time.LocalDateTime;
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("VEND Non puo' cambiare stato da Venduto a Disponibile");
 return this;
 }
}

```

```

// Biglietto e' un Context
public class Biglietto {
 private String codice = "XYZ";
 private int prezzo = 100;

 private StatoBiglietto sb = new Disponibile();

 public void mostra() {
 System.out.println("Prezzo: " + prezzo + " codice: " + codice);
 sb.mostra();
 }

 public void prenota(String s) {
 sb = sb.intesta(s);
 }

 public void cancella() {
 sb = sb.cancella();
 }

 public void compra() {
 sb = sb.paga();
 }
}

```

```

// Classe con il main che usa il Context
public class Client {
 private static Biglietto b = new Biglietto();
 public static void main(String[] args) {
 usaBiglietto();
 }

 private static void usaBiglietto() {
 b.prenota("Mario Tokoro");
 b.compra();
 b.mostra();
 }

 private static void nonUsaOk() {
 b.compra();
 b.cancella();
 b.prenota("Mario Biondi");
 }
}

```

```

 }
}

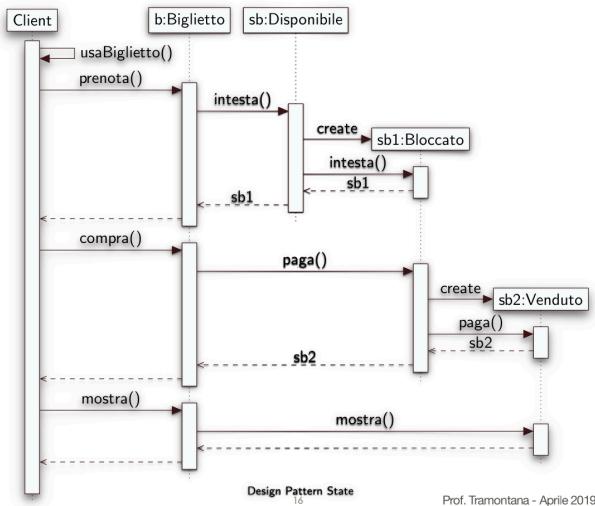
```

-- Output dell'esecuzione

```

DISP Cambia stato da Disponibile a Bloccato
BLOC Inserito nuovo intestatario
BLOC Cambia stato da Bloccato a Venduto
VEND Pagamento effettuato
Prezzo: 100 codice: XYZ
VEND Nome: Mario Tokoro

```



## Conseguenze nell'adozione di State e nel fare Refactoring

- Si sono eliminati i rami condizionali eccessivi e ripetitivi visto che **non si devono testare più i vari stati**
- Il **codice** di ogni classe è **molto più semplice**
- Il **PASSAGGIO DI STATO** è fatto **all'interno** dei singoli *ConcreteState*
  - Le frecce nell'UML indicano le dipendenze fra i vari *ConcreteState* e **NON** è inserita nel *Context* che, appunto, **non è a conoscenza dei vari stati**
  - **VARIANTE:** La logica del **passaggio di stato** potrebbe **trovarsi** anche nel *Context* e questo sarebbe **ANCHE** corretto
- Per **FLUENT** si intende la seguente: si tiene **fissa l'interfaccia** di riferimento e le **invocazioni** avvengono in **cascata**
  - return new Venduto(nome).pagina();
- ➡ • Le **LOC diminuiscono** e i vari costrutti switch che si presentavano inizialmente suggerivano di usare il polimorfismo

## Progettazione e individuazione delle classi

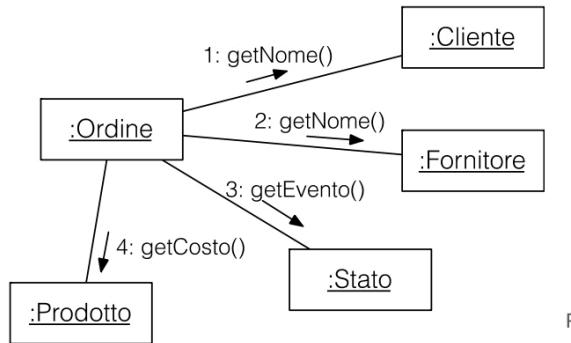
(PDF non presenti oggi 20/04/2023, le carica dopo (tecniche per individuare le classi))

- Fare analisi grammaticale e trovare verbi/sostantivi

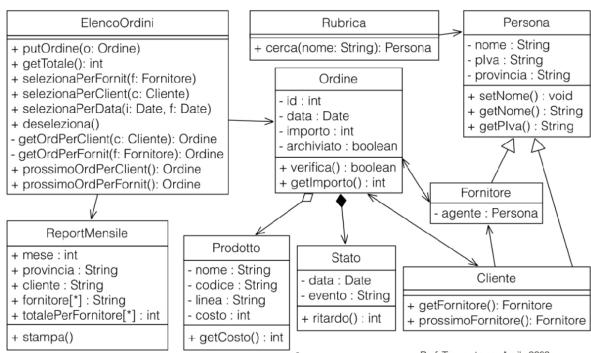
- Per il sistema software trovare **oggetti fisici** menzionati nei requisiti (Biglietto, Auto, ...). Ognuno di essi potrebbe essere potenzialmente **una classe**
- Un **documento dei requisiti è scritto bene** se usa il verbo **DOVRA'**, cioè si deve sapere **cosa dovrà fare il sistema software**
- I numeri che si mettono sulle **frecce delle dipendenze** (relazioni fra classi) vengono detti **MOLTEPLICITA'** e indicano il **numero di istanze**
  - Ordine  $0..* \rightarrow 1$  Cliente: Un cliente può fare 0 o più ordini. *Un ordine è riferito solo ad un cliente*
- L'**AGGREGAZIONE (rombo vuoto)** **Classe1 ◊-> Classe2** si legge Classe2 **AIUTA A DEFINIRE** Classe1. Classe2 al di fuori della classe di Classe1 **potrebbe ancora avere senso**
- La **COMPOSIZIONE (rombo pieno)** **Classe1 ♦-> Classe2** è un'aggregazione più forte. Se Volessi usare Classe2 al di fuori di Classe1 non ha senso. Le due classi sono fortemente legate.

# Diagrammi UML di collaborazione

Mostra le **interazioni fra oggetti**. L'obiettivo è mostrare il comportamento a run-time del sistema software (stesso obiettivo del diagramma UML di sequenza, che è anche di collaborazione).



- Si hanno dei nodi (oggetti che interagiscono fra loro) e archi che rappresentano l'interazione fra oggetti.
  - **La freccia** indica quale metodo è stato invocato sull'oggetto oppure va messa un'**indicazione di interazione**. Il numero sulla freccia indica la **sequenza di interazioni** che avvengono (quindi l'ordine)
    - Preso come esempio il metodo 1: `getNome()` esso potrebbe ritornare qualche valore. In questo caso si avrà il ritorno nella variabile 1 della classe *Ordine* -> 1: `1 := getNome()`
  - Si interagisce sempre con una **SINGOLA ISTANZA**. L'istanza è indicata dai : e dalla sottolineatura -> :Cliente è un'istanza della classe Cliente. -> c:Cliente è l'istanza c di Cliente
  - La **disposizione** degli oggetti è **LIBERA**



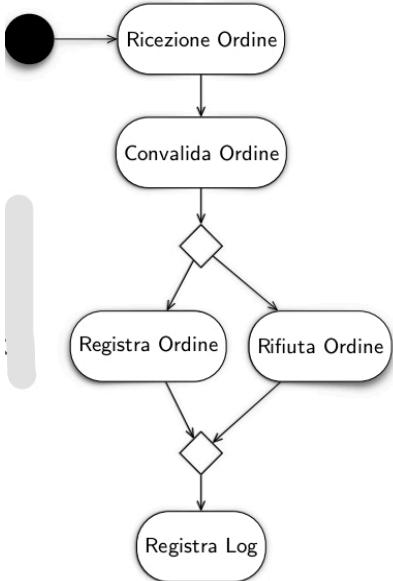
- L'analisi grammaticale sui requisiti **non** ha evidenziato le classi Persona, Rubrica, ElencoOrdini, e Stato. Il progettista deve capire che servono
  - 'Cerca un cliente' è realizzato dal metodo `cerca()` di Rubrica
    - Rubrica può contenere istanze di Fornitore o Cliente
  - 'L'elenco dei fornitori di un cliente' è realizzato dai metodi `getFornitore()` e `prossimoFornitore()` di Cliente
    - La classe Cliente tiene una lista di Fornitore
  - 'Calcolare totale ordini' è nel metodo `getTotale()` di ElencoOrdini
    - L'insieme di Ordini su cui calcolare il totale è generato dai metodi `selezionaPerClient()`, `selezionaPerData()`

- I metodi `selezionaPerFornit()`, `selezionaPerClient()`, `selezionaPerData()` permettono ciascuno di estrarre una parte degli ordini secondo criteri diversi, quindi si possono usare separatamente, e sono metodi brevi
- Si possono richiamare i suddetti metodi separatamente per ottenere selezioni di ordini (e totali) differenti rispetto al requisito
- `prossimoXyz()` permette di scorrere la lista correntemente selezionata dall'esterno della classe `ElencoOrdini` [vedi `List`,  `StringTokenizer`, etc.]

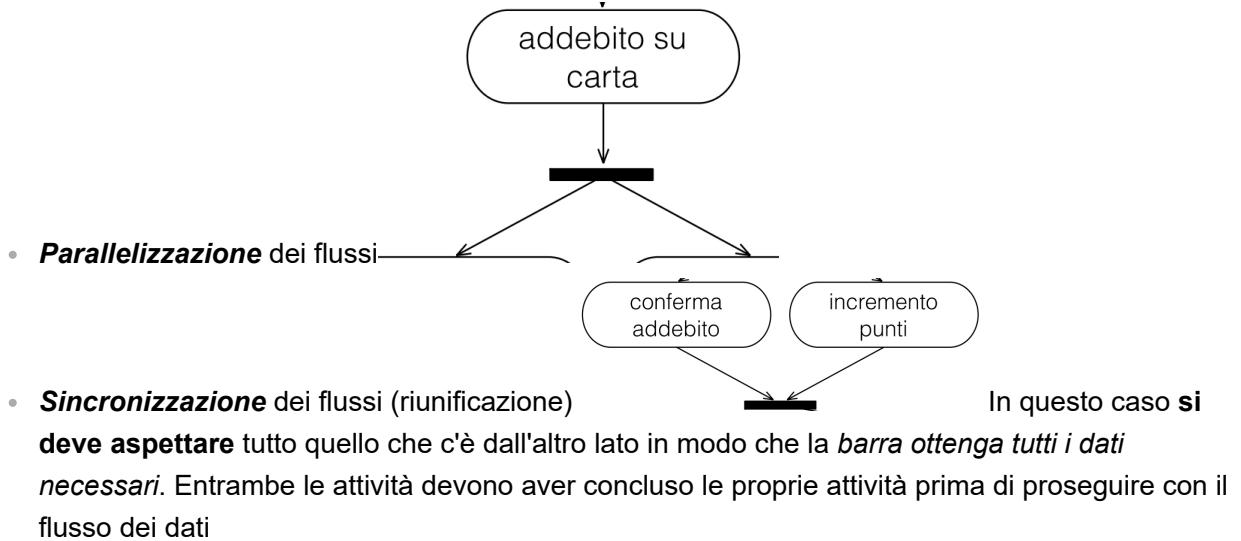
- Serve una **lista di clienti e fornitori** (anche se **NON** è presente nei requisiti) che permette di tenere le varie istanze di *Cliente* e *Fornitore*
  - Determinate scelte per gli attributi/metodi ***non sono scritte nei requisiti*** ma dovranno essere ricavate dai progettisti

## Diagramma UML delle attività

Mostra le attività e i **passi di esecuzione del software**.

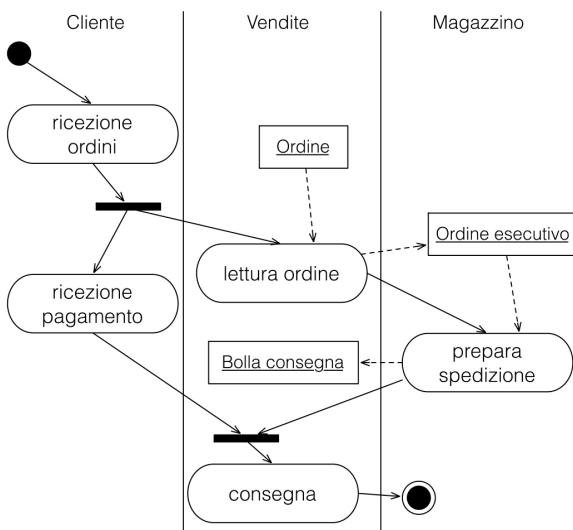


- Evidenzia le **operazioni più importanti** del software
- L'**attività** è un **rettangolo MOLTO arrotondato ai bordi** (`Ricezione Ordine`)
  - Le linee che collegano le attività sono delle frecce che indicano quale attività viene prima e quale dopo, quindi il **flusso delle attività**
- La **PRIMA ATTIVITA'** è indicato da un **cerchio annerito**
- Il **ROMBO** indica se bisogna seguire un flusso o un altro in base a delle condizioni indicate dei **FLUSSI ALTERNATIVI**. (si usa anche per **UNIFICARE I FLUSSI**)
- Lo **STATO FINALE** si può *indicare* (o *non indicare* se non necessario) con il seguente simbolo:
  - È utile per *chiarire dubbi* sulle attività finali del software
- Una **barra annerita** si vuole mostrare che il flusso di esecuzione si rammifica e diventa *parallelo* e si seguono entrambi i flussi attraverso **2 attività dedicate** con **2 Thread Paralleli**. È rappresentato dal seguente simbolo:



## Diagramma UML con corsie

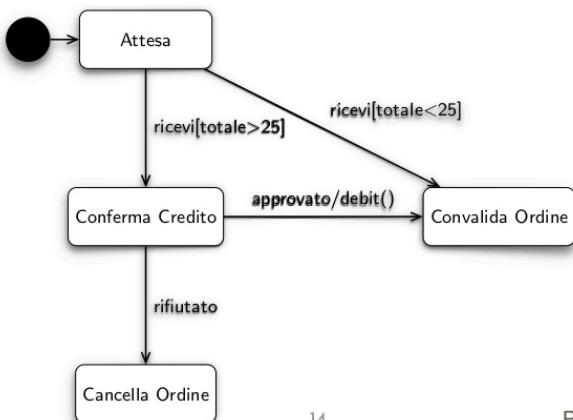
Si guardano solitamente le **MACRO-attività**. I **responsabili** di determinate **attività** devono essere individuati. Si possono disegnare delle suddivisioni orizzontali/verticali, dove ogni corsia ha un'**intestazione** che indica il **nome del responsabile** di quell'attività.



*Ordine* è un **dato** ed è **IN INGRESSO** (freccia entrante nell'attività) o **RISULTANTE** dall'attività (freccia uscente dall'attività).

## Diagramma UML degli stati

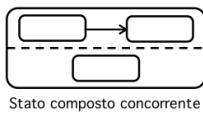
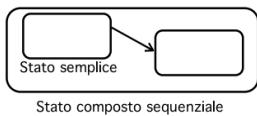
Sono rappresentati da rettangoli con i soli **ANGOLI ARROTONDATI**.



- La freccia è indicata con l'evento che viene chiamato che fa **transitare** in uno stato ulteriore
- La transizione potrebbe essere **subordinata ad un evento** oppure **ad una condizione** che può essere **soddisfatta oppure no**. La condizione viene scritta accanto l'evento -> `evento [n>5]` .
  - Se la condizione **non è soddisfatta**, allora la **transizione non avviene**
  - Le condizioni devono essere **ESCLUSIVE** in modo da non avere un blocco delle transizioni fra stati
  - `evento[n>5]/debit()` si legge -> se  $n > 5$  allora effettua l'attività `debit()` e transita nello **stato successivo**
- Le attività da fare pertinenti allo stato servono:
  - `entry / cerca()` -> L'attività in ingresso nella transizione è `cerca()`
  - do -> attività da fare durante la permanenza di uno stato
  - `exit / finish()` -> l'attività da fare prima di uscire da una transizione

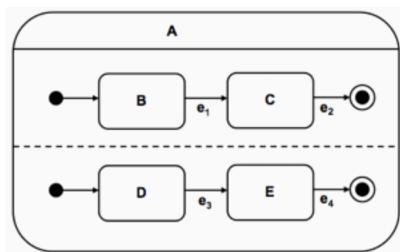
## Stati composti

All'interno di uno stato ci sono altri stati che a loro volta, ovviamente, possono essere semplici o composti:



L'**etichetta** si può mettere sempre in alto (come negli stati semplici) oppure **si può omettere** senza annotarlo nel diagramma stesso ma da qualche altra parte.

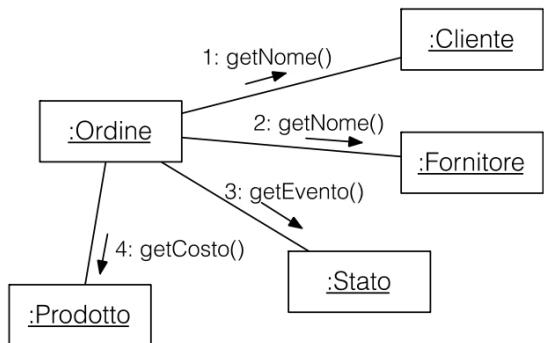
- Occorre specificare **uno stato iniziale** (*non rappresentato nel disegno soprastante*) e si usa sempre **un pallino nero** collegato con una **freccia** ad uno stato che viene considerato iniziale.
- La **transizione fra gli stati interni** si **indica allo stesso modo** con una freccia etichettata dal **nome dell'evento**



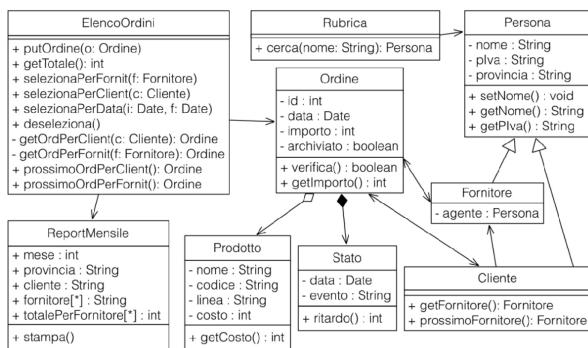
Gli stati composti possono essere **sequenziali** o **paralleli** (*linea tratteggiata orizzontale*).

- Vengono assunti allo stesso tempo (diversamente dal sequenziale -> prima uno stato S1 e poi uno stato S2)
- In questo caso, quando si entra nello stato generale (*il più grande*) **si entra contemporaneamente** negli stati sopra e sotto# Diagrammi UML di collaborazione  
Mostra le **interazioni fra oggetti**. L'obiettivo è mostrare il comportamento a run-time del sistema

sotware (stesso obiettivo del diagramma UML di sequenza, che è anche di collaborazione).



- Si hanno dei nodi (oggetti che interagiscono fra loro) e archi che rappresentano l'interazione fra oggetti.
- La freccia indica quale metodo è stato invocato sull'oggetto oppure va messa un'**indicazione di interazione**. Il numero sulla freccia indica la **sequenza di interazioni** che avvengono (quindi l'ordine)
  - Preso come esempio il metodo 1: `getNome()` esso potrebbe ritornare qualche valore. In questo caso si avrà il ritorno nella variabile 1 della classe `Ordine` -> 1: `l := getNome()`
- Si interagisce sempre con una **SINGOLA ISTANZA**. L'istanza è indicata dai `:` e dalla sottolineatura -> `:Cliente` è un'istanza della classe `Cliente`. -> `c:Cliente` è l'istanza `c` di `Cliente`
- La **disposizione** degli oggetti è **LIBERA**



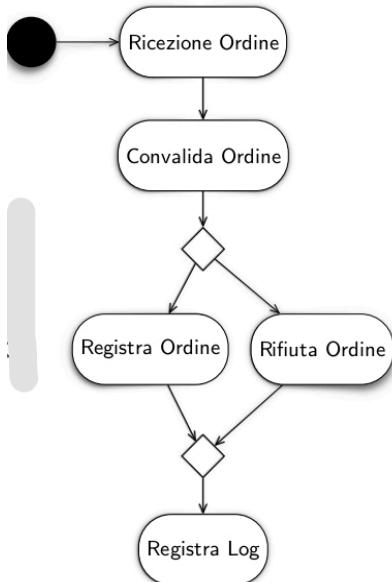
- L'analisi grammaticale sui requisiti non ha evidenziato le classi `Persona`, `Rubrica`, `ElencoOrdini`, e `Stato`. Il progettista deve capire che servono
  - 'Cerca un cliente' è realizzato dal metodo `cerca()` di `Rubrica`
    - `Rubrica` può contenere istanze di `Fornitore` o `Cliente`
  - 'L'elenco dei fornitori di un cliente' è realizzato dai metodi `getFornitore()` e `prossimoFornitore()` di `Cliente`
    - La classe `Cliente` tiene una lista di `Fornitore`
  - 'Calcolare totale ordini' è nel metodo `getTotale()` di `ElencoOrdini`
    - L'insieme di `Ordine` su cui calcolare il totale è generato dai metodi `selezionaPerClient()`, `selezionaPerData()`
  - I metodi `selezionaPerFornit()`, `selezionaPerClient()`, `selezionaPerData()` permettono ciascuno di estrarre una parte degli ordini secondo criteri diversi, quindi si possono usare separatamente, e sono metodi brevi
    - Si possono richiamare i suddetti metodi separatamente per ottenere selezioni di ordini (e totali) differenti rispetto al requisito
  - `prossimoXYZ()` permette di scorrere la lista correntemente selezionata dall'esterno della classe `ElencoOrdini` [vedi `List`,  `StringTokenizer`, etc.]

- Serve una **lista di clienti e fornitori** (anche se **NON** è presente nei requisiti) che permette di tenere le varie istanze di `Cliente` e `Fornitore`

- Determinate scelte per gli attributi/metodi ***non sono scritte nei requisiti*** ma dovranno essere ricavate dai progettisti

# Diagramma UML delle attività

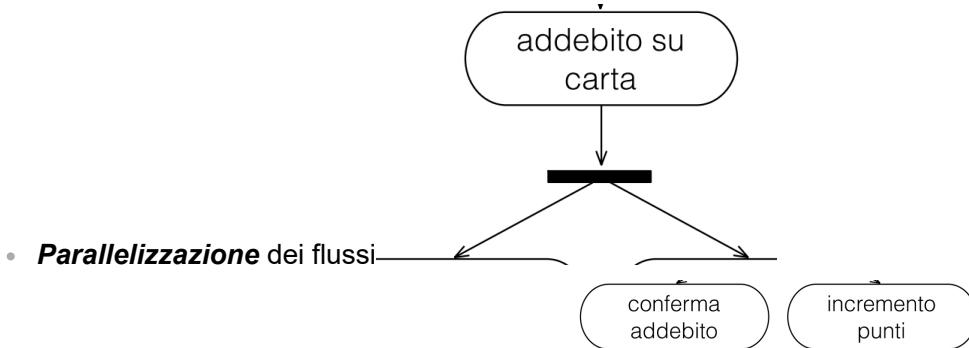
Mostra le attività e i **passi di esecuzione del software**.



- Evidenzia le **operazioni più importanti** del software
- L'**attività** è un **rettangolo MOLTO arrotondato ai bordi** (*Ricezione Ordine*)
  - Le linee che collegano le attività sono delle frecce che indicano quale attività viene prima e quale dopo, quindi il **flusso delle attività**
- La **PRIMA ATTIVITA'** è indicato da un **cerchio annerito**
- Il **ROMBO** indica se bisogna seguire un flusso o un altro in base a delle condizioni indicano dei **FLUSSI ALTERNATIVI**. (si usa anche per **UNIFICARE I FLUSSI**)
- Lo **STATO FINALE** si può *indicare* (o non indicare se non necessario) con il seguente simbolo:



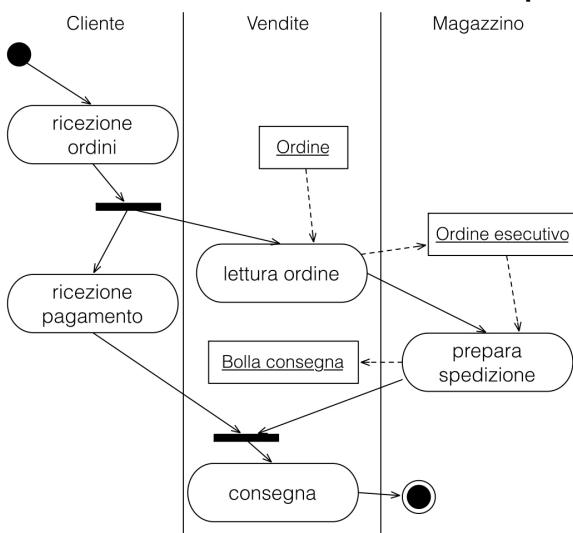
- E' utile per *chiarire dubbi* sulle attività finali del software
- Una **barra annerita** si vuole mostrare che il flusso di esecuzione si rammifica e diventa *parallelo* e si seguono entrambi i flussi attraverso **2 attività dedicate con 2 Thread Parallel**. E' rappresentato dal seguente simbolo:



- Parallelizzazione** dei flussi
  - Sincronizzazione** dei flussi (riunificazione)
- In questo caso **si deve aspettare** tutto quello che c'è dall'altro lato in modo che la *barra* *ottienga tutti i dati necessari*. Entrambe le attività devono aver concluso le proprie attività prima di proseguire con il flusso dei dati

# Diagramma UML con corsie

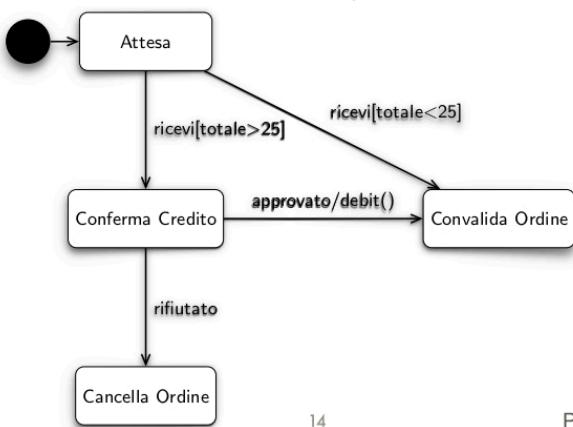
Si guardano solitamente le **MACRO-attività**. I **responsabili** di determinate **attività** devono essere individuati. Si possono disegnare delle suddivisioni orizzontali/verticali, dove ogni corsia ha un'intestazione che indica il **nome del responsabile** di quell'attività.



**Ordine** è un **dato** ed è **IN INGRESSO** (freccia entrante nell'attività) o **RISULTANTE** dall'attività (freccia uscente dall'attività).

# Diagramma UML degli stati

Sono rappresentati da rettangoli con i soli **ANGOLI ARROTONDATI**.



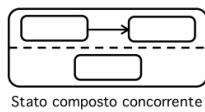
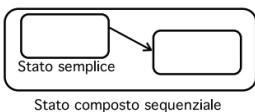
14

Pro

- La **freccia** è indicata con l'**evento che viene chiamato** che fa **transitare** in uno stato ulteriore
- La transizione potrebbe essere **subordinata ad un evento** oppure **ad una condizione** che può essere **soddisfatta oppure no**. La condizione viene scritta accanto l'evento -> **evento [n>5]** .
  - Se la condizione **non è soddisfatta**, allora la **transizione non avviene**
  - Le condizione devono essere **ESCLUSIVE** in modo da non avere un blocco delle transizioni fra stati
  - **evento[n>5]/debit()** si legge -> se  $n > 5$  allora effettua l'attività **debit()** e transita nello **stato successivo**
- Le attività da fare pertinenti allo stato servono:
  - **entry / cerca()** -> L'attività in ingresso nella transizione è **cerca()**
  - **do** -> attività da fare durante la permanenza di uno stato
  - **exit / finish()** -> l'attività da fare prima di uscire da una transizione

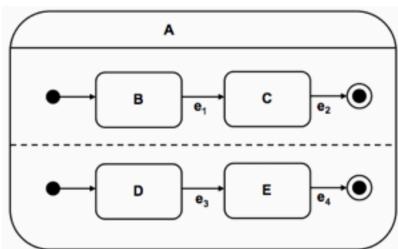
# Stati composti

All'interno di uno stato ci sono altri stati che a loro volta, ovviamente, possono essere semplici o composti:



L'**etichetta** si può mettere sempre in alto (come negli stati semplici) oppure **si può omettere** senza annotarlo nel diagramma stesso ma da qualche altra parte.

- Occorre specificare **uno stato iniziale** (*non rappresentato nel disegno soprastante*) e si usa sempre **un pallino nero** collegato con una **freccia** ad uno stato che viene considerato iniziale.
- La **transizione fra gli stati interni** si *indica allo stesso modo* con una freccia etichettata dal **nome dell'evento**



Gli stati composti possono essere **sequenziali** o **paralleli** (*linea tratteggiata orizzontale*).

- Vengono assunti allo stesso tempo (diversamente dal sequenziale -> prima uno stato S1 e poi uno stato S2)
- In questo caso, quando si entra nello stato generale (*il più grande*) **si entra contemporaneamente** negli stati sopra e sotto

# Processi di sviluppo del software

Sono delle **descrizioni** per lo sviluppo di un sistema software.

Ce ne sono molti che hanno in comune delle attività tipiche (che, in base ai processi, possono variare):

- Analisi dei requisiti (**specifiche**)
- Progettazione (**design**) -> si determinano quali componenti servono per realizzare il software (con i vari diagrammi UML e le proprie descrizioni)
- Codifica o implementazione (**codice**) -> si fa corrispondere del codice rispetto al design
- Convalida o testing (**approvazione**) -> si testa il software con vari input per vedere se i *requisiti sono soddisfatti*
- Manutenzione

## Analisi dei requisiti

Si va a definire quello che il software dovrà fare e questo rappresenta proprio una specifica.

**Specifiche** e **Requisito** sono sinonimi ma, per la precisione:

- Requisito -> **Caratteristica singola** che il software deve avere (anche *informale*, con pochi dettagli)
- Specifica -> **Descrizione RIGOROSA** di una caratteristica del software (viene usato un *formalismo* particolare e ci sono *molti dettagli scritti bene*)

## Feature è una caratteristica del software

Per mettere a punto i requisiti si fanno i seguenti passi:

- **STUDIO DI FATTIBILITÀ** -> ascoltare le esigenze del cliente e capire, prima di scrivere in modo dettagliato i requisiti, se si è **propensi e capaci** a poter costruire il software oppure no (*esempi di rifiuto -> il cliente non ha le idee chiare, non si è in grado di sviluppare una data tecnologia richiesta, non è possibile soddisfare il cliente in termini di costo/tempi*)
- **ANALISI DEI REQUISITI**
- **SPECIFICA DEI REQUISITI** -> Si ha un documento dettagliatissimo dei requisiti con una buona organizzazione dei dati
- **CONVALIDA DEI REQUISITI** -> Rilettura dei requisiti, anche da parte del cliente, dove si vanno a correggere gli eventuali errori. Si possono realizzare dei **PROTOTIPI** che simulano l'interazione con l'utente

Per i requisiti si distinguono:

- **requisiti funzionali** -> **COSA** il software dovrà fare
- **requisiti non funzionali** -> **COME** il software lo dovrà fare (affidabilità, efficienza, prestazioni, sicurezza dei dati ecc..)

## Progettazione dettagliata del sistema

In questa fase si **determinano** le **classi** che servono, **interfacce**, **relazioni fra classi** e questa conclude la fase di progettazione.

E' possibile anche arricchire la progettazione e implementare altre classi.

Spesso si passa da progettazione a implementazione in modo tale da incrementare i dettagli del sistema

## Implementazione

Si devono **ricavare degli algoritmi** sulla base delle basi della progettazione e **rimuovere i difetti** dovuti a malfunzionamento. Il programma scritto va anche **testato** per analizzare bene il comportamento

## Convalida e test

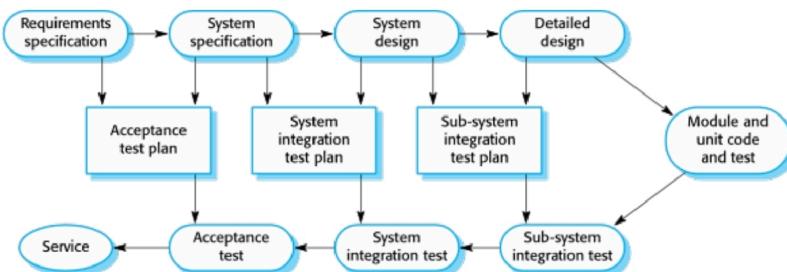
Si rileggono i requisiti e si prova il sistema software secondo questo documento. In alcuni casi bisogna ridefinire e riscrivere codice per migliorare e per adattarsi ai requisiti.

Si fanno vari tipi di test:

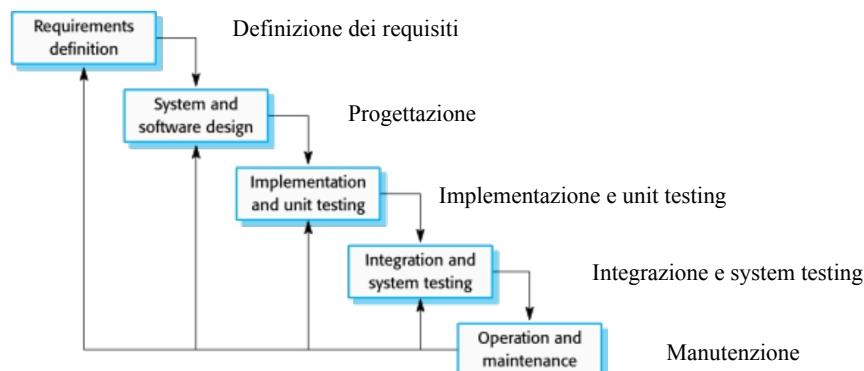
- **Test di unità (unit test)** -> si verificano i singoli metodi/classi e vengono scritti durante l'implementazione
- **Test di sistema** -> Si devono testare le interazioni fra classi/metodi realizzando test complessivi dando speciale importanza alle proprietà emergenti
- **Test di accettazione** -> si simula il funzionamento del sistema con i dati del cliente, anche con una certo arrotondamento della loro quantità
- **Beta test** -> viene fornito il software a pochi clienti e si fa lavorare il cliente con il software per vedere se il software funziona su diversi sistemi. Si vanno a tracciare le varie operazioni e vengono rilevati i risultati

- Versione **ALFA** (*con difetti e parti mancanti*) -> versione quasi finale per poter eseguire test di accettazione ma si deve ancora testare
- Versione **BETA** (*con difetti*) -> il software viene fornito a diversi clienti dove vengono fatti beta test
- Versione **GOLD** -> il software ha superato tutti i test e quindi è la versione finale

Riassumendo:



## Processo di sviluppo a cascata



Questo è il *primo processo di sviluppo* documentato

Date le fasi descritte prima, vengono eseguite *nella maniera elencata*, in modo **ORDINATO**.

- **Solo quando una fase è completa si può passare alla fase successiva**
- Eventuali modifiche si possono effettuare **solo dopo essere arrivato alla manutenzione**, quindi alla fine della cascata.

*Si affrontano le singole fasi e solo quando si è deciso che la fase è finita, si passa alla fase successiva e si esegue tutto in modo ordinato.*

Un **punto di forza** di questo sistema è proprio la fase iniziale (*analisi dei requisiti*) che, se vengono scritti bene e sono stabili, allora il processo riesce a realizzare un **software di qualità** in tutte le fasi successive (*implementazione, test ecc..*)

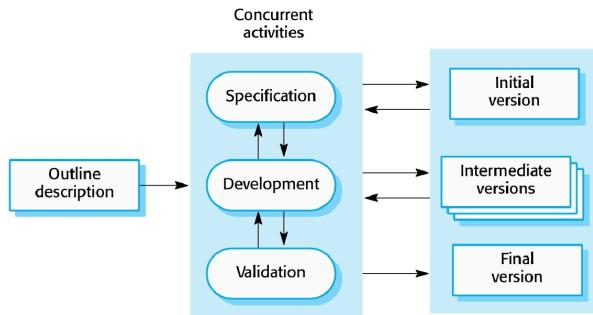
Questo modo di procedere può portare vantaggi e svantaggi e quindi bisogna valutare quando adottare (o *non adottare*) questo sistema:

- **Vantaggi:**
  - Viene usato per sistemi grandi, complessi, **critici** (cioè quelli da cui dipende la vita umana -> *tecnologia di un aereo*) per garantire un'alta qualità del prodotto)
  - Vi è un'**ampia documentazione**
  - Utile se i requisiti sono stabili e chiaramente definiti
  - C'è un **grande team di sviluppo** e, ogni volta che viene prodotto del codice, le varie componenti vengono *integrate nel sistema principale*
- **Svantaggi:**

- La **durata** della raccolta dei requisiti (*con la progettazione e implementazione*) è **lunga**. *L'intero processo dura, in media, un anno*
- Si hanno **poche interazioni con i clienti** visto che esse avvengono solo durante la fase della raccolta dei requisiti
- **Non è facile introdurre i cambiamenti richiesti dal cliente** (in caso di ripensamenti da parte sua). Di conseguenza si devono *ignorare le richieste* di cambiamento e andare avanti.
- Questo tipo di sviluppo non è "forte" in caso di rilasci anticipati in versioni *di prova* del software durante la fase di sviluppo. Quindi il prodotto viene consegnato solamente alla fine della sua realizzazione senza anticipi.

# Processi di sviluppo evolutivi

E' una **famiglia di processi di sviluppo**. Si chiamano così perchè si procede nello sviluppo per evoluzione, per modifiche successive a quello che si realizza.



## Per esplorazione

Il processo di sviluppo **PER ESPLORAZIONE**:

- Si lavora a **stretto contatto con il cliente** per tutta la durata del **processo di sviluppo** dove si hanno delle prime specifiche iniziali
- Su queste ultime si progetta e si produce il codice
- Questa prima parte del codice viene mostrata al cliente e lui stesso verifica la correttezza e aggiunge ulteriori dettagli

Le **SPECIFICHE** inizialmente raccolte sono **CHIARE**. Vengono sviluppate delle singole parti delle specifiche e quindi, inizialmente, non si ha una visione globale del processo di sviluppo che, invece, sarà più chiara solo più avanti.

## Build and Fix

- Si inizia a progettare, implementare e, nel frattempo, si devono aggiustare *i problemi*
- In particolare, quello che si sviluppa scaturisce da **REQUISITI NON CHIARI** all'inizio. Nonostante questo *disagio* si **tenta a sviluppare del codice** e si va a **modificare finché il cliente è soddisfatto**
- Si ha una **comprendizione limitata** del sistema da produrre e la **fase di design** è pressoché **inesistente**
- I requisiti non chiari possono derivare da possibili *difficoltà del progettista o del cliente nello descrivere i requisiti necessari*
- Il codice prodotto è di **BASSA QUALITÀ**

*Non si è ben capito cosa si deve realizzare ma si inizia a realizzare*

- Questo sistema può essere consigliato **SOLO** per **eventuali prototipi** (che non saranno i sistemi finali) che rappresentano delle **versioni non complete** che mostrano al cliente che si può arrivare a un **determinato** sistema software. La **realizzazione di prototipi è veloce** dove si può adottare *build and fix*.

## Problemi e applicazioni

*Problem:*

- I tempi sono **lunghi**
- Il codice è di **bassa qualità**
- Il **costo** è **dificilmente stimabile** inizialmente  
*Applicazioni:*
- Sistemi di **piccole dimensioni**
- Singole parti di sistemi grandi (es. interfaccia utente)
- Sistemi con **vita breve** (es. prototipi)

## Incrementale

Vengono fatte aggiunte successive a qualcosa che si è già realizzato, in particolare:

- A differenza di prima, si raccolgono i **requisiti** dai quali **si scelgono quello che permettono di realizzare componenti che fanno da fondamenta** per il sistema software. A queste ultime vengono **aggregate al codice principale**.
- Si ha una **visione globale** di quello che si vuole realizzare
- **Non si modificano le parti già sviluppate ma a esse vengono integrate nuove parti**

## CBSE (Component-Based Software Engineering)

Realizzazione software basata su componenti. Si usano componenti già esistenti e realizzati. Si fa un **RIUSO DEI COMPONENTI** per un successivo sistema software da realizzare.

*Si dice che è basato su COTS (Components Off The Shelf), cioè componenti già realizzati, appunto.*

- Si **raccolgono i requisiti** dal cliente e si cerca di **trovare una corrispondenza fra i requisiti e i requisiti che avevano fatto sorgere eventuali componenti già realizzate**
- Si propone al cliente una **variazione dei requisiti** per vedere se è possibile integrare componenti già usate in modo da arrivare ad un accordo bilaterale.
  - Per tutte le **variazioni accettate dal cliente** si devono **solo integrare** le componenti che si vogliono usare che già esistono
- Si cerca di essere **VELOCI** nello sviluppo del sistema software nuovo (nonostante l'integrazione) e di rendere **MENO COSTOSA** la realizzazione (*grazie al riuso*)

*Questo processo viene usato se l'azienda ha sviluppato molti software e comunque vuole usare componenti passate*

## A spirale

Inventato da Biehm nel 1988.

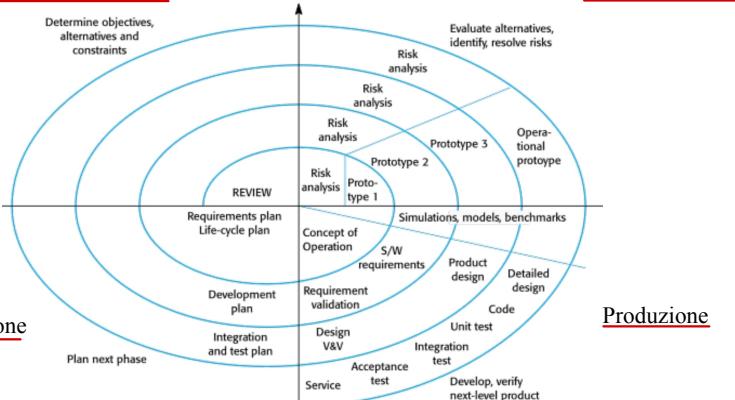
Tutti i processi di sviluppo prendono esempio da altri processi utilizzati da altre aziende e si valuta se aggiungere o togliere dei passaggi

Si realizza un prodotto in più fasi che sono *un giro di una spirale* quindi tutto è organizzato secondo questa forma geometrica (*dall'interno verso l'esterno*). I vari passaggi vengono fatti **tramite dei cicli** (dette **FASI**) e un ciclo si conclude dopo 360° in **SENSO ORARIO**.

Tra i vantaggi del processo di sviluppo software a spirale ci sono:

- 1) Gestione dei rischi: la fase di analisi dei rischi permette di individuare e gestire i rischi già nelle prime fasi del progetto, riducendo il rischio di fallimento del progetto.

## Pianificazione degli obiettivi



Ciò che si deve fare in ogni quadrante viene descritto dal processo di sviluppo.

- Nel primo quadrante (*in alto a sinistra*) si fa una **PIANIFICAZIONE DEGLI OBIETTIVI**:
  - In questa fase si decidono gli obiettivi **di un solo ciclo** (quello in corso)
  - Si valutano le priorità
  - In questa fase si può pianificare anche un'eventuale raccolta dei requisiti
- Nel secondo quadrante si fa la **VALUTAZIONE DEI RISCHI**:
  - Si analizza *quello che potrebbe non essere soddisfatto* quando si procede con le due fasi successive e questi rischi devono essere valutati e, in questo caso, si possono prendere delle **precauzioni**
- Nel terzo quadrante si fa la **PRODUZIONE** e qui si fanno le attività per gli obiettivi preposti durante la fase di pianificazione.
- Nel quarto quadrante si fa la **REVISIONE**:
  - Si verifichino che tutti gli **obiettivi stabiliti sono stati raggiunti** durante la fase di produzione
- **Un ciclo di spirale dura parecchio** (in media *6+ mesi*) ma non tutti i cicli hanno la stessa durata (*ovviamente variano*)
- Le interazioni con il cliente avvengono abbastanza spesso (non giornalmente ma *prevede un buon dialogo*)
- **Il tempo di realizzazione è ampio**.
  - *Se si fanno 2 cicli è come un processo a cascata ma spesso se ne fanno più di 2.*

## Processo di Extreme Programming

XP è un manifesto di un processo di sviluppo che contiene i dettagli:

- Si deve porre l'accento su certe cose piuttosto che su altro
- Quando si sviluppa un software la priorità non è la scrittura dei documenti o la pianificazione sistematica del tempo ma lo è **l'importanza degli individui** (*rispettare le persone che partecipano allo sviluppo*)
- Bisogna **collaborare con il cliente** piuttosto che mettere contratti che fissano le cose in modo stabile e preciso (ci vogliono, sì, ma non con così alta priorità)
- La **priorità** tende alla capacità di essere **AGILI** nello sviluppo del software (capacità di incorporare nuove richieste) che comportano cambiamenti senza sconvolgere il modo di operare o le date di consegna. Questo è il punto focale del manifesto **XP**

*Questo processo di sviluppo lo si fa se le persone che lavorano sono persone motivate, ben istruite e che mirano a un'eccellenza e che hanno a cuore il prodotto da produrre piuttosto che pensare al solo lavoro\*\**

**2) Flessibilità:** il modello a spirale è flessibile e può essere adattato alle esigenze del progetto. Le iterazioni successive permettono di apportare modifiche e miglioramenti in base ai feedback ricevuti.

**3) Coinvolgimento del cliente:** il modello a spirale prevede un coinvolgimento attivo del cliente, che può fornire feedback e suggerimenti in ogni fase del progetto.

**4) Miglioramento continuo:** il processo a spirale prevede una valutazione continua del prodotto, permettendo di apportare miglioramenti e correzioni in ogni iterazione.

**5) Riduzione dei costi:** la gestione dei rischi e il miglioramento continuo del prodotto permettono di ridurre i costi complessivi del progetto.

Il processo di sviluppo XP ha le seguenti **caratteristiche**:

- **Poco tempo** per lo sviluppo incrementale
- **Piccolo team** per ogni incremento
- **Poca documentazione**
- **Costante miglioramento** del codice
- Molta **comunicazione** fra le persone
- Tanta **interazione**
- Tanti **test** del codice (quasi sempre)

*Ciò permette di avere tante piccole release di ottima qualità.*

## Principi di XP

- Si ha un **commento rapido** di quello che si sta sviluppando da parte del cliente
- Si fanno **cose semplici**
- Si hanno dei **cambiamenti graduali** supportati
- Si produce **codice di qualità**

Le **PRATICHE XP** sono 12 che consentono di produrre codice di qualità, di essere *AGILI* ed è adottato da tutte le aziende.

- |                                                 |                                    |
|-------------------------------------------------|------------------------------------|
| • Gioco di pianificazione                       | • Design semplice                  |
| • Piccole release                               | • Possesso del codice collettivo   |
| • Metafora                                      | • Integrazione continua            |
| • Testing                                       | • Settimana di 40 ore              |
| • Refactoring                                   | • Usare gli standard per il codice |
| • Pair Programming<br>(programmazione a coppie) |                                    |
| • Cliente in sede                               |                                    |

## Gioco di pianificazione

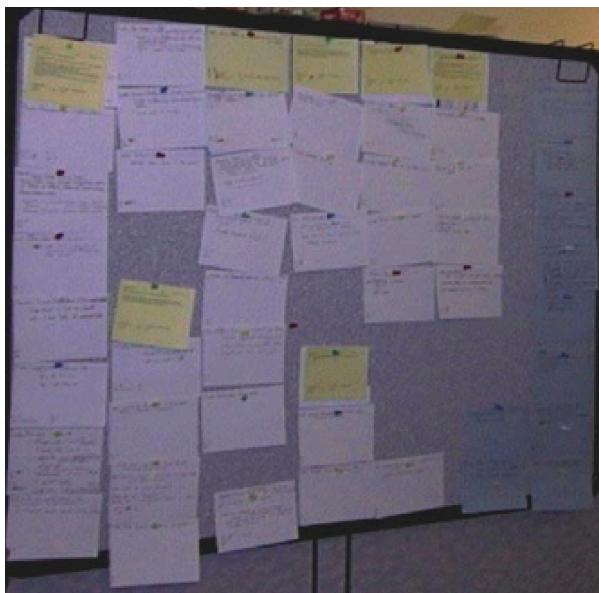
Questa fase produce dei requisiti **scritti bene**, sono **granulari** e **comprendibili**.



Si fa quando si vuole sviluppare **codice di alta qualità**. Si raccolgono i requisiti e si categorizzano dando loro una valutazione. Tale raccolta viene svolta nel seguente modo:

- I clienti si riuniscono con gli sviluppatori (\*che sono **pochi**\*) al fine di raccogliere i requisiti
- Il cliente scrive un **singolo requisito** (detta *storia*) sulla **STORYCARD** (una scheda piccola, 12x7 cm). Quest'ultima è piccola affinché il cliente si focalizzi su una **singola cosa**
- La scheda viene **passata** ai sviluppatori che si fanno un'idea sul requisito
- Nel frattempo il cliente scrive altri requisiti. Alla fine si raccolgono un certo numero di schede

- Si fa una stima per la realizzazione di un requisito, quindi di una sola scheda.
- Le schede **incomprensibili e che richiedono troppo tempo (oltre ai due giorni) vengono strappate**. Se il tempo è troppo lungo allora sta a significare che il requisito comprende anche altri **sotto-requisiti** al suo interno
- Si scelgono delle priorità alle storie da parte del cliente (*in base ai propri requisiti*) e degli sviluppatori (*in base alla loro esperienza*) e quindi si cerca di arrivare a un  **compromesso**.
- Si fa una **stima generale del tempo necessario** per sviluppare tutte le storycard \* cercando di non superare le **3 settimane\***
  - Durante questo periodo di sviluppo è possibile far subentrare qualche modifica al codice visto che il tempo di sviluppo è molto poco
- Si fa una pianificazione **più precise nelle prime 2 settimane**. La terza settimana è per quello che deve essere fatto dopo (**meno prioritario** e con una **stima grossolana**).
- Si fornisce una **piccola release dopo le 2 settimane**, chiedendo dei commenti al cliente e dopodiché si rifà il gioco di pianificazione.
  - Quello che era stato pianificato per la terza settimana possono diventare priorità per il prossimo ciclo di 2 settimane oppure possono essere cambiate durante il nuovo gioco di pianificazione
- Bisogna scegliere cosa fare all'inizio **in base ai rischi**. Se si tratta di una cosa difficile da realizzare o comunque si tratta di algoritmi sofisticati allora deve avere **priorità maggiore** e quindi realizzata durante il primo sviluppo per avere tempo per fare i dovuti fix.
- Inoltre, *ogni story deve poter essere testata* in modo da *validare* ciò che si produce
- Le StoryCard vengono messe nella **Storyboard**



- Le storyboard ha 3 colonne:
  - Ogni colonna rappresenta una settimana
  - Le storycard da realizzare si mettono in alto mentre le storycard realizzate si mettono in basso



## Pair programming

Consiste nel **programmare in coppia**, cioè da due programmatore alla volta e ogni coppia avrà una storycard. Si scrive molto codice, quasi bypassando la fase di progettazione. Nella coppia si segue la seguente organizzazione:

- Si ha un unico PC
- Un programmatore ha l'uso della tastiera del mouse che scrive del codice (**DRIVER**)
- L'altro programmatore (**NAVIGATOR**) osserva il collega e dà dei suggerimenti all'occorrenza, quindi fa **CODE REVIEW** in modo da produrre codice di alta qualità e trovare soluzioni migliori. Egli pensa anche ai test da realizzare prendendo degli appunti a penna che poi convertirà in codice.
  - I ruoli vengono scambiati entro qualche ora
- Alla realizzazione della storycard, la coppia ha realizzato il codice ed essa viene completata **solo dopo la scrittura dei dovuti test**. La storycard completata viene depositata nella storyboard.
- Alla fine della storycard, la coppia si scioglie e ci si aggrega ad altri programmatore per sviluppare altre stoycard.
- Ogni coppia *conosce una parte del sistema software* e queste piccole conoscenze vengono *passate anche alle prossime coppie che si formeranno*. Questa, insieme alla **presenza del cliente in sede**, è una **contromisura alla scarsa documentazione** presente in questo processo di sviluppo

Se la coppia incontra qualche problema durante lo sviluppo (*perchè quel requisito è effettivamente incomprensibile*) allora ci si rivolge al cliente (*che è in sede*) che può dare ulteriori spiegazioni.

I due programmatore devono essere **ESPERTI** e **MOTIVATI** affinchè la coppia sviluppi bene:

- se **uno dei due è meno esperto** allora possono nascere perdite di tempo.
- Se **entrambi sono principianti** passano più tempo a discutere piuttosto che scrivere codice e quindi usano male il proprio tempo.

## Piccole release

Dopo 2 settimane bisogna ottenere qualcosa di funzionante e in questo piccolo tempo è **possibile accorgersi di eventuali cose che non funzionano**.

- La **progettazione** deve essere *piccola* e non si va a valutare l'opzione "Ma questo sistema crescerà? Ci saranno altre cose da aggiungere?". Al momento dell'aggiunta di altre feature, si può fare **refactoring semplice**.
- Realizzare delle piccole parti funzionanti, inoltre, garantisce **incoraggiamento e motivazione** al team
- Si fanno possibili aggiustamenti e non si vanno a buttare mesi e mesi di lavoro

## Metafora

- Crea una sorta di **slogan** in modo che tutti **gli sviluppatori si ispirano** ad esso quando devono prendere delle decisioni. Per esempio: "*La UI è un desktop!*".
- Il **cliente deve essere partecipe** di questa metafora
- La **metafora deve rappresentare l'architettura** del sistema software che si sta sviluppando

"**Il nostro sistema software è un edificio**".

In questa metafora, il software viene rappresentato come un edificio, con le sue fondamenta, le pareti, i piani e i tetti. Le fondamenta rappresentano le componenti di base del sistema, come le librerie e i framework, che supportano il sistema software. Le pareti rappresentano le classi e i moduli del sistema, che proteggono il software dalle interferenze esterne e garantiscono la sicurezza e la stabilità del sistema. I piani rappresentano le funzionalità e le caratteristiche del sistema, organizzate in modo gerarchico e strutturato, mentre il tetto rappresenta l'obiettivo finale del sistema, ovvero la soluzione completa e funzionante del problema che il software deve risolvere.

## Design semplice

Quando la coppia di programmatore legge la storycard pensano alle dovute classi che servono e agli algoritmi da usare.

Si pensa alle **classi e algoritmi più semplici possibili**. Non si fa più progettazione del dovuto ma viene fatta *al volo*.

*Caratteristiche:*

- Il codice deve essere di qualità
- Classi **piccole e modulari**
- Quello che si produce deve superare i test
- Non ci sono parti duplicate
- Si deve **esprimere**, nel codice, **l'intento in maniera chiara**
- Non ci si preoccupa molto di quello che si implementa inizialmente (*quindi si prende una decisione e si applica*) perchè comunque in futuro potrebbero essere necessarie delle modifiche
- Le **scelte della progettazione** (*rapida*) sono **documentate nel CRC**(*Class Responsibility Collaboration*) che è una scheda (*circa grande quanto la storycard*) e contiene:
  - Il **nome** della classe
  - Le sue **responsabilità**
  - Le **interazioni** con le altre classi

Class Cartella	
Responsibility	Collaboration
Legge i file immagine da una cartella Filtra i file in base ad un criterio di selezione Tiene il numero di immagini presenti Tiene il numero di immagini selezionate	Img File (da java.io)

**In XP non si fa uso di diagrammi UML.**

## Testing

Bisogna **testare sempre tutto**. I test vengono eseguiti appena si pensa che il codice è completo e vengono fatti **in locale**. Dopodiché vengono **depositati** su sistemi remoti dell'azienda (*Git*).

Successivamente vengono scritti ed eseguiti i **test di sistema** (o *di integrazione*) che vengono eseguiti **sull'intera applicazione**, quindi test delle classi che interagiscono fra loro.

**(ESAME) Quando si eseguono i test nel processo XP?** Più volte al giorno:

- Quando la coppia di programmatore finiscono di implementare la storycard o nel frattempo
- Quando il codice è condiviso sulla repository (push)

I **test** rappresentano la **specifiche dei requisiti** (in formato eseguibile).

Unit test: test delle singole unità o singolo metodo

Test funzionali: test dell'integrazione fra più parti del codice

I test possono essere **scritti prima di scrivere il codice**, ovvero applicare la **TDD**(*Test Driven Development*)

## Possesso collettivo del codice

- Chiunque partecipi al progetto può leggere e modificare il codice sorgente.
- Ogni membro del team è responsabile per l'intero sistema
- Visto che il codice viene spesso modificato, i test **proteggono** le funzionalità del sistema

## Integrazione continua

Man mano che si sviluppa del codice non ci si deve preoccupare solo di quello che si scrive ma **si deve integrare con il codice dell'intero software**.

- Quando la coppia fa l'integrazione può capitare che questa non vada a buon fine. Quindi si potrebbe decidere di buttare il codice prodotto e ricominciare
- Se le integrazioni (e quindi lo sviluppo del codice) si fanno con una durata di 6-8 ore e poi il codice si deve buttare, allora vuol dire che si sono spurate solo quelle poche ore di lavoro

## 40 ore a settimana

- Si lavora **8 ore** al giorno
- Visto che ci sono molte cose si riconosce il fatto che **se si è stanchi non si lavora bene** e quindi non si trovano errori in maniera semplice
- Se ci si accorge che per andare avanti nella scrittura del codice per le storycard bisogna **lavorare delle ore extra** (*superando il tempo stimato sulla storycard*) vuol dire che la **stima non è stata corretta** perchè si sono verificati degli imprevisti (*cambio piattaforma di sviluppo, librerie non disponibili ecc..*).
  - In questo caso ci si riunisce e si prendono le eventuali decisioni per affrontare i problemi sorti

## Cliente in sede

**Il cliente è insieme agli sviluppatori** quindi si trova in sede. In questo modo può **rispondere in qualsiasi momento** alle eventuali domande dei programmatore e questo **compensa la poca documentazione**.

*Il dialogo quindi spiega eventuali dubbi sul codice scritto*

- In vari momenti durante la giornata **il cliente è disponibile per chiarire dubbi** ai programmatore (di **qualsiasi ruolo**) e, nel *tempo libero*, può **svolgere il proprio lavoro**.
- La presenza del cliente in sede, quindi, rappresenta complessivamente **un vantaggio molto strategico** per lo sviluppo del software.
- **Stabilisce priorità** e fornisce il contesto per le decisioni dei programmatore

## Standard di codifica

- Quando si sviluppa in team bisogna stabilire delle convenzioni per la codifica (*parentesi graffe nella stessa linea oppure subito sotto, posizione dei commenti, scelta dei nomi delle classi/metodi, CamelCase o snake\_case, spaziatura*) e questo **facilita la comprensione del codice** da parte dei programmatore e del cliente.
- Il team che partecipa allo sviluppo del software **si mette d'accordo sugli standard da adottare** durante tutto il processo di sviluppo
- Costruzioni complicate (per il design) **NON** sono permesse

[Convenzioni linguaggio Java](#)

## Refactoring

- Refactoring vuol dire **ristrutturare il codice senza cambiarne la funzionalità**.
- Si ha un **miglioramento della STRUTTURA del codice** e quindi una maggior **modularità**
- Se si aggiungono molte linee di codice, allora si fa di nuovo refactoring
- I **test** possono essere scritti **in maniera più semplice** visto che i metodi conterranno *poche linee di codice*
- Dopo il refactoring bisogna **verificare che il comportamento del codice sia identico** alla versione precedente e, per questo motivo, **si scrivono dei test prima di fare refactoring**.
- Si punta a **codice senza ripetizioni**

## Considerazioni di XP

- **Si focalizza sul codice:** alta qualità, facile da comprendere, molti test che documentano quello che si deve fare
- **Si orienta sulla gente** e mette le persone al primo posto e la loro **collaborazione** (anche perchè il cliente è in sede)
- **E' leggero:** si **pianifica** il lavoro per il **breve termine** e \***NON** si spreca tempo per il lungo termine\*. Se si spreca del tempo per una storycard esso è un **tempo breve**. **Rimuove** anche **costi aggiuntivi** e **crea un prodotto di qualità** tramite **TEST RIGOROSI**
- I principi di XP non sono nuovi perchè queste pratiche erano già adottate nel passato

XP è stato descritto nel 1999 e, di seguito, sono elencati gli **aggiornamenti più significativi**:

- *Documentazione:* nel 2008 StackOverflow
- *Metodologia:* nel 2014 Mob Programming (*ci sono molte persone che lavorano allo stesso codice e solo uno di loro scrive mentre gli altri guardano e ogni 15 minuti si scambiano i ruoli*)
- *Metodologia:* dal 2020 circa, è possibile instaurare una sessione di *Remote Pair Programming*, per esempio usando Live Share di VSCode (*la produttività si abbassa*)
- *Metodologia:* dal 2023 circa, *ChatGPT* (*assitente di capacità enormi*) -> *Pair Programming* (il programmatore esperto deve essere la *PERSONA FISICA* per **EVITARE** di essere **rimpiazzati da ChatGPT**)

## Scrum

E' un processo di sviluppo (inventato nel 2010) che, se usato da solo, non risponde a molte domande/esigenze (tipo la frase "*Usiamo Scrum*"). Un'azienda che usa solo Scrum non vuol dire nulla (a differenza di "Sto usando XP e Scrum" che ha più senso).

- Da una terminologia e una direzione per orientarsi nella programmazione

XP e Scrum affrontano aspetti diverse dell'organizzazione del lavoro e **non affrontano lo stesso problema**.

- Il **PRODUCT OWNER** è il proprietario del software che dovrà riceverlo. Prende scelte sul prodotto e sulle caratteristiche e i requisiti. *Quindi elenca il lavoro da realizzare*. I requisiti da realizzare vengono raccolti nel **PRODUCT BACKLOG** che viene **riempito più volte**.
- **SCRUM TEAM** è il team di sviluppatori che adottano le tecniche Srum che prende i requisiti e produce il software.
- Il prodotto viene realizzato in **più iterazioni** (*più passate di produzione*) e ognuna di esse viene chiamata **SPRINT** che **dura 4 settimane** (*in XP durava 2 settimane*).
- Il team valuta il da farsi per i prossimi sprint insieme al Product Owner e insieme ad **altre persone (STAKEHOLDER)** che *hanno interesse nello sviluppo dello stesso prodotto*.

## Principi di Scrum

E' semplice ma incompleto. Si basa sui seguenti termini:

- **EMPIRISMO:** si deve **osservare la realtà** e prendere **scelte basate su quello che avviene nella realtà**. Nelle aziende non sempre si ha la capacità di osservare la realtà (*per esempio nella stima dei tempi di consegna del prodotto -> si stima un tempo brevissimo ma in realtà ne serve molto di più*)

- **LEAN:** letteralmente vuol dire *snello/agile* e significa **ridurre lo spreco e concentrarsi sull'essenziale** (si produce solo quello che serve subito).

Nella guida di Scrum non vi è la parola *agile* perchè compare *lean*

## Pilastri di Scrum

Scrum si sostiene sui seguenti pilastri (*devo avere la prima per la seconda e devo avere la seconda per avere la terza*: trasparenza -> ispezione -> adattamento):

- **TRASPARENZA:** Tutto il codice prodotto derivato dal Product Backlog (che *può venire* prodotto in *qualsiasi modo*, in genere con le storycard) deve essere **visibile a tutti** e di **facile consultazione**. *Se tale pilastro è soddisfatto, allora posso avere il successivo.*
- **ISPEZIONE:** Si vanno a cercare i problemi molto frequentemente per scoprire potenziali problemi. Scrum fornisce la cadenza delle ispezioni tramite **5 eventi**.
- **ADATTAMENTO:** Si vanno a cercare le soluzioni per un determinato problema ispezionato. L'adattamento si applica subito per evitare ulteriori divergenze dei problemi

## Valori di Scrum

Il successo nell'uso di Scrum dipende da 5 valori :

- **Commitment:** il team si impegna a **raggiungere gli obiettivi** e a supportarsi a vicenda
- **Focus:** concentrazione sul lavoro da fare **senza farsi distrarre** da nuovi avvenimenti durante lo sviluppo
- **Openness:** il team può **cambiare direzione durante lo sviluppo** e il lavoro può essere riprogrammato in base alle nuove idee che vengono fuori
- **Respect:** ogni componente del team **rispetta gli altri** (modo di procedere, punto di vista)
- **Courage:** il team **prende con coraggio delle decisioni importanti di comune accordo**. Il team lavora su problemi difficili. Il team non si lascia scoraggiare dalla difficoltà del problema da risolvere.

## Eventi di Scrum

Ci sono degli eventi importanti:

- **Sprint:** iterazioni del lavoro. Si ha un Product Backlog e da esso si selezionano le cose da fare subito e si fa il primo sprint.
- **Sprint Planning:** Prima dello sprint si fa una pianificazione del lavoro (*8 ore al massimo*) per un mese (4 settimane). Partecipano tutti i membri del team e il Product Owner
- **Daily Scrum:** giornalmente c'è un **incontro** che dura *15 minuti* e si fa all'inizio della giornata. Serve per scambiarsi informazioni tecniche sul lavoro svolto e il lavoro da svolgere. Si programma un po' il da farsi durante la giornata. Si affronta **in piedi** per avere la **sensazione che il tempo stia scorrendo** e quindi per sbrigarsi (*se ci si siede si ha la sensazione di poter perdere molto più tempo*).
  - **SPRINT BACKLOG** -> si estraggono, dal Product Backlog, i **requisiti da portare a termine nel singolo Sprint**. Lo Sprint Backlog include anche **attività che non sono relative per ottenere nuove funzionalità** e quindi sono solamente **attività da svolgere**
- **Sprint Review:** si fa dopo la conclusione dello sviluppo delle 4 settimane e dura al massimo 4 ore. Serve per capire se si sono raggiunti i risultati posti come obiettivi all'inizio dello Sprint. Si aggiungono eventuali cose da fare al Product Backlog

- **Sprint Retrospective:** si ragiona su come si è operato per il singolo sprint e si va a valutare l'organizzazione adottata per tentare di migliorarla. (si sono incontrate difficoltà nell'utilizzo di una libreria)
  - **Definition of Done:** si deve capire bene e **DEFINIRE** quando si reputa che **effettivamente si è finito il lavoro** (è *PERSONALE* e può essere: "ho finito quando ho completato i test", "ho completato i test senza errori", "ho consegnato il codice al cliente")

## Artefatti di Scrum

Ci sono varie cose che possono essere prodotte (*Artefatti*):

- **Product Backlog:** Lista dei requisiti del prodotto che servono per migliorarlo. E' la sorgente del lavoro del team
- **Product Goal:** obiettivo e si trova nel backlog e descrive lo stato futuro del prodotto
- **Sprint Backlog:** obiettivo complessivo del singolo sprint.
  - **Sprint Goal** è l'obiettivo dello Sprint
- **Increment** è qualcosa che porta verso il Goal. Quando il Product Backlog soddisfa la Definition of Done allora si ha un nuovo incremento
- **Scrum Master:** è una figura che fa da **allenatore per il team di Scrum** che ha **più esperienza** con la tecnica di Scrum e quindi **dà consigli** su come attuare questi principi e di come **rendere più snello il lavoro da fare**.

## Refactoring

- Refactoring è il processo che **cambia un sistema software** in modo che **il comportamento esterno del sistema non cambi**, ovvero i requisiti funzionali soddisfatti sono gli stessi, per far sì che la struttura interna sia migliorata
- Si migliora la progettazione a poco a poco, durante e dopo l'implementazione del codice

**Esempio semplice:** consolidare (ovvero eliminare) frammenti di codice duplicati all'interno di rami condizionali

Vantaggi del refactoring:

- Spesso la **dimensione del codice si riduce**
- Si **comprende meglio il codice**
- Le **strutture complicate** si trasformano in **strutture più semplici** da capire e mantenere
- Si evita di introdurre un *debito tecnico* all'interno della progettazione

## Tecnica Estrai metodo

```

public void printDovuto(double amount) {
 printBanner();
 System.out.println("nome: " + nome);
 System.out.println("tot: " + somma);
}

• Diventa

public void printConto(double somma) {
 printBanner();
 printDettagli(somma);
}

public void printDettagli(double amount) {
 System.out.println("nome: " + nome);
 System.out.println("tot: " + somma);
}

```

A livello di progettazione:

- `printDovuto()` si serve di un metodo di più basso livello perchè lo richiama (`printBanner()`) e implementa istruzioni di basso livello
- Si rende di livello più alto se si trasportano le `System.out` in un metodo apposito e il metodo `printConto()` fa due macro-attività.
- `printConto()` ha un livello di astrazione più preciso perchè prima faceva sia cose elementari che non elementari.
- Adesso, se voglio stampare di nuovo i dettagli posso solo richiamare la funzione `printDettagli` piuttosto che chiamare `printDovuto()`

## Motivazioni Estrai Metodo

Si applica:

- Quando si hanno **metodi lunghi** (più di 15 linee)
- Quando il **codice non è comprensibile**
- Quando i metodi si hanno commenti all'interno del codice (avere commenti è sintomo di refactoring). Il codice sotto il commento diventa una nuova funzione e si applica la tecnica *Estrai Metodo*. Ogni commento da anche il nome del futuro nuovo metodo che viene inventato da applicare
- Per avere metodi piccoli: in caso di gerarchia di classi **si fa facilmente OVERRIDE**.

## Meccanismi

Come applicare la tecnica Estrai Metodo: (**ESAME**):

- **Creare un nuovo metodo** il cui nome comunica l'intenzione del metodo
- **Copiare il codice estratto** dal metodo sorgente al nuovo
- Guardare se il codice estratto **ha variabili locali al metodo sorgente** e far diventare tali variabili **parametri del metodo**
- Se alcune variabili sono usate solo all'interno del codice estratto farle diventare variabili temporanee del nuovo
- Se una variabile locale al metodo sorgente è modificata dal codice estratto, vedere se è possibile far sì che il metodo estratto sia una query e assegnare il risultato alla variabile locale del metodo
- Sostituire nel metodo sorgente il codice estratto con una chiamata al metodo nuovo

Se il metodo estratto deve ritornare più cose ci sono diverse alternative:

- **uno di questi valori viene tornato e gli altri vengono assegnati ad attributi**
- **più valori di ritorno li metto negli attributi e il valore di ritorno diventa void**
- **si crea un tipo apposito che contiene tutti i valori di ritorno (coppia <String, Integer> e simili)**

```

public class Ordine {
 private double importo;
 public double getImporto() {
 return importo;
 }
}

public class Stampa {
 private ArrayList<Ordine> ordini;
 private String nome = "Mike";
 public void printDovuto() {
 Iterator<Ordine> i = ordini.iterator();
 double tot = 0;
 // scrivi banner
 System.out.println("-----");
 System.out.println("- Cliente Dare -");
 System.out.println("-----");
 }
 // calcola totale
 while (i.hasNext()) {
 Ordine o = i.next();
 tot += o.getImporto();
 }
 // scrivi dettagli
 System.out.println("nome: " + nome);
 System.out.println("tot: " + tot);
}
}

public class Stampa2 {
 private ArrayList<Ordine> ordini;
 private String nome = "Mike";
 public void printDovuto() {
 printBanner();
 double tot = getTotal();
 printDettagli(tot);
 }
 public double getTotal() {
 Iterator<Ordine> i = ordini.iterator();
 double tot = 0;
 while (i.hasNext()) {
 Ordine o = i.next();
 tot += o.getImporto();
 }
 return tot;
 }
 public void printBanner() {
 System.out.println("-----");
 System.out.println("- Cliente Dare -");
 System.out.println("-----");
 }
 public void printDettagli(double somma) {
 System.out.println("nome: " + nome);
 System.out.println("tot: " + somma);
 }
}

```

Prof. Tramontana - Marzo 2019

# 11-05-2023

## 2° Tecnica sostituisci Temp con Query

Le Temp sono *variabili temporanee*, quindi non sono attributi. Si eliminano le variabili locali a un metodo e al loro posto si può mettere un metodo che tiene il valore che potrebbe tenere la variabile (Query).

```
//Temp
double prezzoBase = quantita * prezzo;

//diventa Query
private double prezzoBase(){
 return quantita*prezzo;
}
```

I **VANTAGGI** nell'uso di questa tecnica sono:

- Quando si ha una variabile locale (temp) *si può usare solo all'interno del proprio scope*, quindi all'interno di un unico blocco, mentre *adesso lo scope aumenta* e migliora la possibilità di usare tale variabile e la **modularità**.
- Ci si libera delle variabili locali che producono codice lungo e quindi si **spezzetta l'algoritmo in più metodi**
- E' utile applicare questa tecnica *prima di Estrai Metodo*
- Se la variabile è **assegnata solo una volta** allora si può pensare di applicare questa tecnica. Se, invece, le **assegnazioni sono diverse** e varie, allora bisogna riflettere sul da farsi.

Per **verificare** che la variabile è **assegnata solo una** volta basta provare a **compilare** assegnando la variabile temp come `final`.

La parte destra dell'assegnazione va dentro il corpo del metodo e nei punti dove la variabile viene usata si mette il nome del metodo creato.

```
private double quantita, prezzo;
public double getPrezzo1() {
 double prezzoBase = quantita * prezzo;
 double sconto;

 if (prezzoBase > 1000) sconto = 0.95;
 else sconto = 0.98;

 return prezzoBase * sconto;
}

//diventa
private double quantita, prezzo;
public double getPrezzo2(){
 return prezzoBase() * sconto();
}

private double prezzoBase() {
 return quantita * prezzo;
```

```

}

private double sconto() {
 if (prezzoBase() > 1000) return 0.95;
 return 0.98;
}

```

## 3° Tecnica dividi variabile Temp

Una variabile temporanea è **assegnata più di una volta**. **NON** è assegnata in un *loop* o usata per *accumulare valori* ma viene **sempre sovrascritta**.

- La variabile temporanea ha un *nome non significativo*
- Se la variabile viene divisa si **rende il codice più comprensibile**
- **Non ci si preoccupa delle prestazioni** (*memoria* in uso) perchè, quando si esce dal metodo, le variabili create vengono distrutte perchè escono dal proprio *scope*
- Le parti divise della Temp diventano `final`

```

double temp = 2 * (height + width); //prima assegnazione
System.out.println(temp);
temp = height * width; //seconda assegnazione
System.out.println(temp);

//diventa
final double perim = 2 * (height + width);
System.out.println(perim);
final double area = height * width;
System.out.println(area);

```

- Se \*l'assegnazione avviene molte volte ma si tratta di **accumulazione\***, allora la tecnica non si applica
- \*Se l'assegnazione avviene molte volte e **Temp ha più responsabilità\***, allora la tecnica si applica.
- Si prova assegnando `final` a Temp: se **ci sono altre assegnazioni**, si verifica che **non sia usata come accumulatore**, allora si applica la tecnica. Il nome della seconda assegnazione (e *anche della prima*) si cambia e tutte le future occorrenze presenti nel codice; altrimenti si lascia così e com'è.
- La **rinominazione** può essere **facilitata rinominando la prima assegnazione e compilando il codice**. Il compilatore segnerà un errore del tipo "*Variabile non dichiarata*" ed è proprio lì che bisognerà dare *dichiarare la seconda variabile*.

### Esempio di utilizzo:

```
private double primaryForce, secondaryForce, mass, delay;

public double getDistanceTravelled1(int time) {
 double result;
 double acc = primaryForce / mass; // prima assegnazione
 int primaryTime = (int) Math.min(time, delay);
 result = 0.5 * acc * primaryTime * primaryTime;
 int secondT = (int) (time - delay);
 if (secondT > 0) {
 double primaryVel = acc * delay;
 acc = (primaryForce + secondaryForce) / mass; // seconda assegnazione
 result += primaryVel * secondT + 0.5 * acc * secondT * secondT;
 }
 return result;
}
```

Diventa

```
public double getDistanceTravelled2(int time) {
 double result;
 final double primAcc = primaryForce / mass;
 int primaryTime = (int) Math.min(time, delay);
 result = 0.5 * primAcc * primaryTime * primaryTime;
 int secondT = (int) (time - delay);
 if (secondT > 0) {
 double primaryVel = primAcc * delay;
 final double secondAcc = (primaryForce + secondaryForce) / mass;
 result += primaryVel * secondT + 0.5 * secondAcc * secondT * secondT;
 }
 return result;
}
```

17

Prof. Tramontana - Marzo 2

# Design Pattern Observer

## Intento

- Le classi devono avere una sola responsabilità e possono contenere molti oggetti e le interazioni possono diventare complicate.
- Si hanno dati sparsi fra le classi e tali dati devono rimanere consistenti fra loro. Il dato su cui deve lavorare una classe viene appena aggiornato da un'altra classe.
- Quindi **il dato aggiornato in una prima classe** deve essere **aggiornato nella seconda classe** così che la seconda classe possa **usarlo in modo consistente**.
- *Un dato viene aggiornato da una classe e deve essere fatto avere da tutte le altre classi.*
- La propagazione dell'aggiornamento del dato deve avvenire in maniera automatica in modo da rendere complicate le dipendenze fra oggetti.

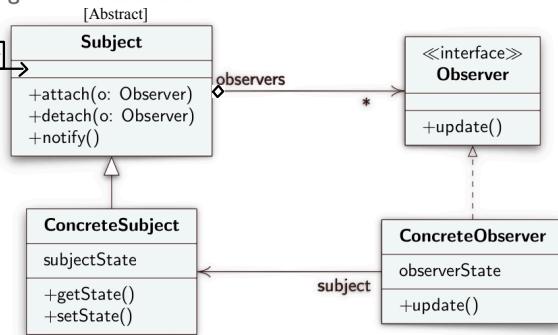
Esempio: *Si ha una tabella di dati che viene aggiornata e il dato che si inserisce in una cella deve aggiornare automaticamente tutte le altre caselle che usano tale dato.*

In questo design pattern vengono descritti:

- **OBSERVER**: osservatore dei dati. Qui ci vanno gli **algoritmi che usano i dati senza aggiornarli**.
- **SUBJECT**: tiene i dati osservati, quindi ha tanti Observer. Qui ci vanno gli **algoritmi che tengono e aggiornano il dato**. *Tale aggiornamento deve essere visto dagli Observer*. Non deve conoscere quante/quali sono gli osservatori e il suo lavoro deve essere il più semplice possibile.

## Soluzione

Diagramma delle classi



**ConcreteSubject** è sottoclasse di **Subject**:

- Tiene i dati ed è pronta a darli aggiornati a chi ha bisogno di conoscerli
- Lo stato `subjectState` può variare e non per forza si deve sovrascrivere
- Sa come **aggiornare il dato** mediante `notify()` della superclasse

**Subject** è una classe non astratta (**ESAME**) ha una lista di osservatori e:

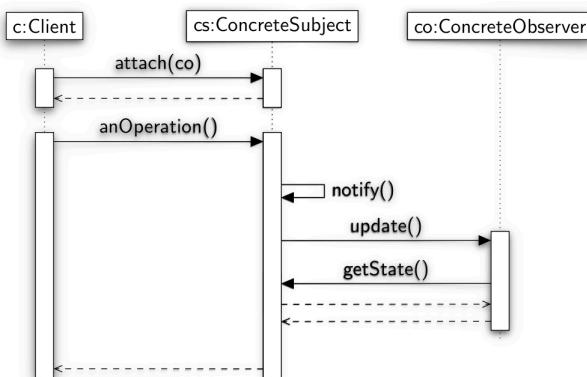
- tiene gli **algoritmi che aggiornano** automaticamente i dati e notifica **Observer**
- `attach(o: Observer)` -> **aggiunge** degli osservatori
- `detach(o: Observer)` -> **rimuove** degli osservatore
- `notify()` -> **notifica** tutti gli osservatori e quindi **chiama metodi ( update() ) sui vari osservatori** utilizzando (scorrendo) la lista di osservatori che ha al suo interno

Ci sono **ConcreteObserver** che sanno fare **determinate** cose differenti dagli altri mentre **Observer** è un'interfaccia (**ESAME**).

Il **ConcreteObserver** conosce il **ConcreteSubject** e ne può usare i metodi.

Quindi ci sono **2 modalità per questo Design Pattern**:

- I dati arrivano con la chiamata ad `update()` e la modalità è detta **PUSH** perchè Subject spinge i dati verso il ricevente **Observer**
- L'altra modalità è **PULL**: il CS avvisa con `notify()` ad `update()` ma non gli passa i dati. Quando gli serve il dato, CO chiede il dato a CS e fa un'operazione di pull del dato mediante il metodo `getState()`



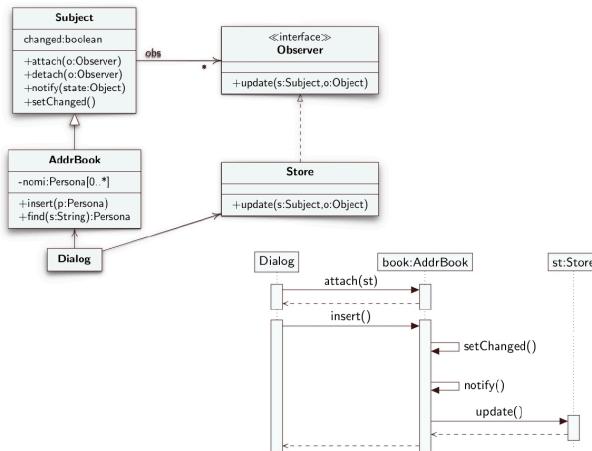
Prima e Dopo l'Uso di Observer



**PRIMA:** in A devo mettere le chiamate di tutte le altre classi passando il proprio stato. Le classi B, C, D costringono A a tenere al suo interno dei riferimenti a B, C, D e le 3 istanze non hanno nulla in comune

**DOPO:** si ha una semplificazione delle relazioni fra le classi. A non conosce le altre classi (con *Observer*) ma solo Subject di cui è sottoclasse. Subject conosce **solo l'interfaccia *Observer*** e quindi non conosce le sottoclassi B, C, D. A runtime riuscirà a invocare i vari metodi update() delle sottoclassi mediante l'interfaccia

## Esempio



- `find()` non modifica i dati e quindi non avrà al suo interno una chiamata a `notify()`
- `changed()` con `setChanged()` serve a Subject per farsi dire dal CS se lo stato è cambiato e, solo quando si vorrà, si notificheranno gli altri.
- Da `insert()` si chiama `notify(nomi)` -> `nomi` passato a `notify()` diventa `Object` -> `update(Object)` e la classe `Store` converte in `List<Persona>`
- `update()` deve avere come primo parametro `Subject` e il **secondo è facoltativo (in base alla modalità scelta)**.
  - Gli viene passato Subject così che, nel CO si può usare `getState()` se previsto per farsi dare lo stato (quindi *PULL*).
  - Subject si mette *obbligatoriamente* perché gli `Observer` potrebbero osservare più CS (quindi **più istanze di sottoclassi di Subject**) e quindi si può **risalire a chi ha fatto l'aggiornamento**.
  - In questo esempio è inutile visto che c'è solo un CS ed è una predisposizione

```

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

 public void notify(Object state) { //state è Object perchè deve essere molto generale perchè non si conosce lo stato del CS (AddrBook)
 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);
 }
}

```

```

}

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 avverrà
 notify(nomi); // notifica i ConcreteObserver
 }

 public Persona find(String cognome) {
 for (Persona p : nomi)
 if (p.getCognome().equals(cognome)) return p;
 System.out.println("AddrBook.find: NOT found");
 return null;
 }
}

public interface Observer { //modalità push visto che viene passato lo stato a update()
 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", "012345", "NY");

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

```

## Conseguenze

- C'è una **completa indipendenza** fra *ConcreteSubject* e *Subject*
- Codici più **semplici da riusare** e modificare
- La **notifica** avviene in **automatico** a tutte le varie istanze di *ConcreteObserver*

- Se update() avviene **troppto spesso**, si può modificare la **tempistica di esecuzione**
- Si possono avere **molti Subject e pochi Observer**: i Subject devono avere una lista di Observer e quindi viene spesso duplicata. Si può avere un'unica tabella che fa i dovuti riferimenti
- Quando si chiama update() si deve conoscere il CS e per questo si passa un riferimento al Subject
- *Subject* chiama `notify()` dopo un cambiamento, oppure *aspetta un certo numero di cambiamenti*, in modo da evitare continue notifiche agli Observer
- Il **CO potrebbe modificare lo stato del CS** e visto che deve essere sempre **consistente** allora può usare `setState()` per farlo passando come **parametro** lo stato aggiornato. Il CS decide se prendere tale parametro e **modifica lo stato**. Con `notify()`, in seguito, *aggiorna tutti gli altri Observer (compreso chi gliel'ha passato)*
- Se il CO ha un attributo CS può invocare `setState()` e `getState()` in qualsiasi momento e **se si volesse eliminare l'istanza di CS**, essa non può essere deallocated perché c'è almeno un CO che ne tiene un riferimento.
  - Se nessun oggetto viene puntato da qualcuno, viene automaticamente eliminata da Java.
  - Si deve eliminare da CO il riferimento di Subject e quindi **aggiungere un metodo che lo faccia diventare NULL** e così l'istanza di CS può essere deallocated in maniera automatica
- Il CS tiene uno stato partizionabile in molte altre cose. `update()` potrebbe non passare lo stato (quindi usando poi pull per farsi dare lo stato) oppure si potrebbe usare: CO, quando usa `attach()` (?) passa anche come parametro lo stato che gli interessa.

Visto che il problema che risolve il design pattern observer Java lo implementa nelle sue librerie:

- Subject e Observer sono implementati con i suoi tipi di libreria predefiniti.
- Le interfacce usate sono `java.util.Observer` e `Observable` (**Subject**)
- Il metodo `notify()` si chiama `notifyObservers()`

Esempio Observer è un tipico problema: un NoteProducer (*ConcreteSubject*) manda messaggi agli osservatori. Uno degli osservatori è **lento** e con il suo tempo di esecuzione **rallenta le notifiche agli altri osservatori**. Visto che è possibile che nascano tali problematiche (un osservatore lento rallenta tutto il flusso del programma) allora l'uso delle interfacce di Java sono deprecate ed è sconsigliato usarle.

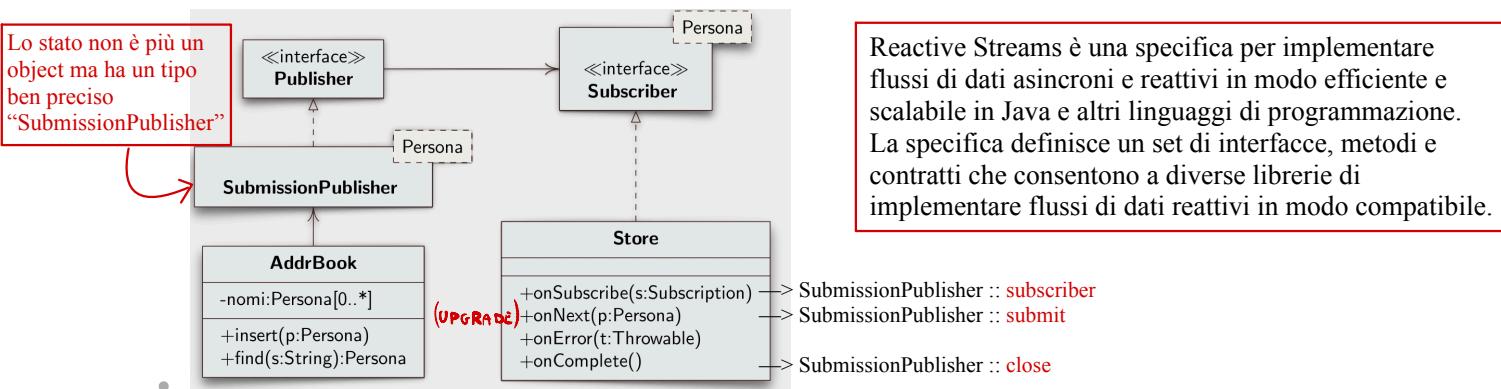
Il problema era presente anche prima di usare le librerie di Java.

Si usa un **parallelismo** fra Thread. Ogni volta che ci sono notifiche (*che magari fa molte cose con un determinato stato*) e notifiche non lente e si usano **più thread di esecuzione**.

## Reactive Streams

Per lanciare più thread di esecuzione si può usare con la versione 9 di Java:

- Anziché usare la classe *Observer* e *Observable* si usa Publisher, Subscriber e tipi di supporto Subscription e SubmissionPublisher.
- **Publisher** e **Subscriber** sostituiscono il Subject e Observer.



- Subscriber può avvisare di vari eventi:
  - `onNext()`: aggiornamento dello stato o un nuovo elemento nello stato
  - `onSubscribe()`: viene chiamato solo una volta al momento della iscrizione nella lista degli osservatori. Quando qualcuno usa l'istanza di Subscriber per agganciarsi a un Publisher (che era `attach()`)
  - `onComplete()`: è l'ultima operazione che invoca un *ConcreteSubscriber* in modo che si avvia un'operazione conclusiva che serve anche per liberare un riferimento e liberare la memoria
  - `onError()`: se capita un errore da parte del chiamante

- Dall'interfaccia Publisher c'è un'implementazione `SubmissionPublisher` serve per la creazione dei thread a servizio della notifica da fare. Scorre la lista degli osservatori e avvia un thread per ognuno di essi.
  - Lo SP può rappresentare lo stato che viene passato poi all'osservatore

*Prima si faceva il cast negli osservatori lo stato era un Object e in questo caso il tipo dello stato non viene perso*

Nel ConcreteSubject si istanzia `private SubmissionPublisher<String> publ = new SubmissionPublisher<>();` che ha i metodi:

- `publ.close()` chiude le operazioni che chiama `onComplete()`
- `publ.submit()` è il notify che chiama `onNext()`
- `publ.subscribe()` chiama `onSubscribe()` dell'Observer e si trova nel metodo `attach(Subscriber<String> sub)`

*Subscriber potrebbe avere molte chiamate per notificare con la chiamata `onNext()`. Se non ha completato il lavoro nell'aggiornamento attuale e arriva il successivo aggiornamento allora, con il parallelismo, c'è anche un sistema per modulare il numero di richieste che può essere configurato in modo da permettere solo un certo numero di notifiche allo stesso tempo.* Per fare questo, nel metodo `onSubscribe(Subscription subscription)` si scrive con `subscription.request(1);` e vuol dire che il numero di richieste contemporanee che si desiderano sono 1. (Serve per dire al `SubmissionPublisher` **manda una successiva notifica solo quando ho finito l'attuale**)

```
public class AddrBook {
 private List<Persona> nomi = new ArrayList<>();
 private SubmissionPublisher<Persona> publ = new SubmissionPublisher<>();

 public void attach(Subscriber<Persona> s) {
 publ.subscribe(s);
 }

 public boolean insert(Persona p) {
 if (nomi.contains(p)) return false;
 nomi.add(p);
 publ.submit(p);
 return true;
 }
}

public class Store implements Subscriber<Persona> {
 private Subscription sub;

 @Override
 public void onSubscribe(Subscription s) {
 sub = s;
 }
}
```

```

 sub.request(1);
 }

 @Override

 public void onNext(Persona p) {
 String nom = p.getNome() + "\t" + p.getCognome();
 System.out.println("Store onNext: "+nom);
 sub.request(1); // numero di richieste contemporanee = 1
 }

 @Override
 public void onError(Throwable throwable) {
 System.out.println("In Store: errore");
 }

 @Override
 public void onComplete() {
 System.out.println("In Store: completato");
 }
}

```

## Model View Controller (MVC)

Fra vari linguaggi di programmazione le logiche si implementano in modi diversi fra loro.

E' un **pattern architetturale** presente in framework e librerie: danno un suggerimento sulla scelta dell'architettura dell'intero progetto, quindi per **MOLTE classi** (*a differenza dei design pattern usati per POCHE classi*).

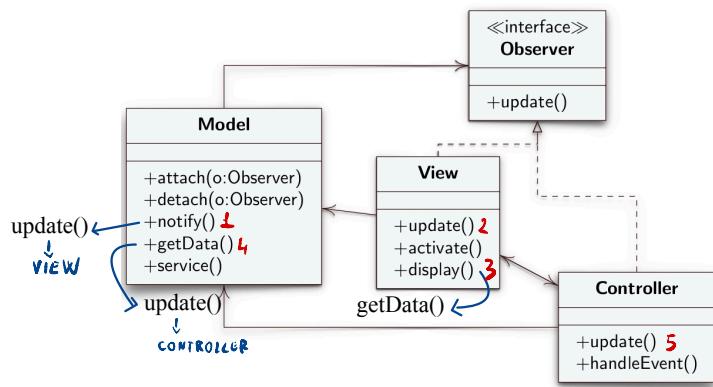
Si hanno **3 ruoli distinti** per l'applicazione:

**Observer = Subscriber = Controller e View  
Subject = Publisher = Model**

- **MODEL**: parte del sistema che è implementata (*da una o più classi*) che realizza le **funzionalità principali** e i **loro dati** vengono modificati e accumulati
- **VIEW**: Sono i dati che devono essere mostrati all'utente e in quale specifico modo. I **dati vengono presi dal Model** ed essi vengono **costantemente aggiornati**
- **CONTROLLER**: parte di programma che **sta in ascolto** delle **interazioni dell'utente (I/O)**.

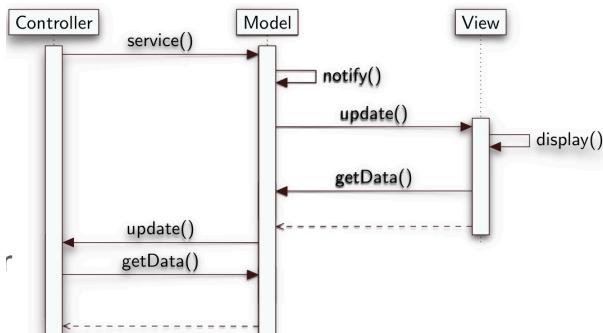
### Motivazioni:

- Le *interfacce utente possono cambiare*, poiché funzionalità, dispositivi o piattaforme cambiano
- Le \*stesse informazioni sono presentate in **finestre differenti**\* (per es. sotto forma di grafici diversi)
- Le visualizzazioni devono subito **adeguarsi** alle manipolazioni sui dati
- I cambiamenti all'interfaccia utente dovrebbero essere **facili**
- Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali



**Model** fa da *ConcreteSubject* mentre **View** e **Controller** fanno da *ConcreteObserver*

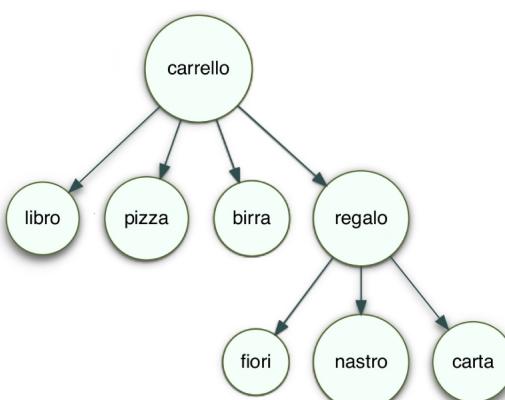
- **Model** incapsula le funzionalità principali e i dati dell'applicazione. E' indipendente dalla rappresentazione degli output e dagli input. Registra View e Controller. Avvisa View e Controller registrati dei cambiamenti di dati
- **View** mostra i dati all'utente. Generalmente ci sono tante View, ogni View è associata a un Controller. View inizializza il proprio Controller, e mostra i dati che legge da Model
- **Controller** riceve gli input dell'utente (da mouse e tastiera) sotto forma di eventi. Traduce gli eventi in richieste di servizio per Model o avvisa View



## Design Pattern Composite

### Intento

Quando si ha a che fare con delle strutture di oggetti che sono organizzate come se fossero un **albero**. Si ha bisogno di una composizione di oggetti e ogni oggetto può contenere altri oggetti. Si vuole poter trattare **oggetti semplici e composti** in maniera **uniforme**. Non si deve adeguare il codice a seconda del tipo di oggetto ma renderlo unico per entrambi i tipi di oggetti.



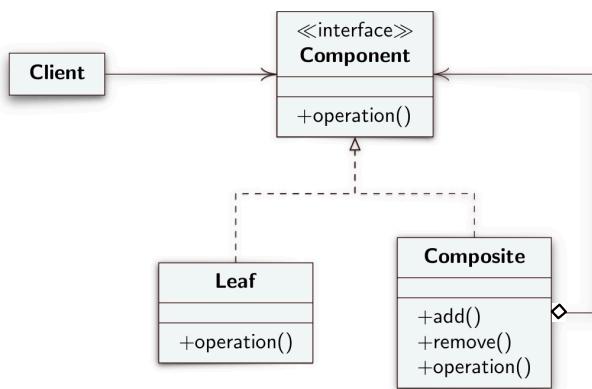
## Esempio File

- Quando si naviga il filesystem, l'oggetto che si trova può essere un file o una cartella. Dentro una cartella si possono mettere altre cartelle oppure file o un mix fra questi due
- Si può leggere/scrivere un file o leggere/scrivere una nuova cartella e il filesystem viene in aiuto sotto questo punto di vista
- Si distingue *cosa si fa* per un oggetto semplice e *cosa si fa* per un oggetto composto.

## Esempio Figure

- In un editor di figure, si possono avere delle figure semplici (cerchio, linea ecc..) ed elementi semplici si possono raggruppare per ottenere elementi composti
- L'operazione di spostamento di una figura composta deve essere fatto su tutte le figure semplici che compongono la figura composta
- In base al tipo di oggetto, l'output di un'operazione deve essere esattamente uguale

## Soluzione



Vi sono le entità:

- **Component**: interfaccia (anche senza implementazioni) e può anche essere astratta (eventualmente) e si **predispongono le operazioni** che serviranno nell'applicazioni e vengono implementate nelle sottoclassi che rappresenteranno rispettivamente gli oggetti semplici e quelli composti
- **Leaf**: classe che rappresenta gli **oggetti semplici**. Qui ci sono i metodi per gli oggetti semplici e viene detta anche *child/children*
- **Composite**: classe che rappresenta gli elementi contenitori per gli oggetti semplici/composti che si mettono al suo interno. Contiene una lista di istanze di tipo Component. A design time sono Component così è compatibile con Leaf e Composite
  - Contiene `add(Component)` e `remove(Component)` che permette di aggiungere o rimuovere elementi dalla lista
  - Questo comporta che si deve verificare che si è dentro Composite visto che è presente solo in questa classe e quindi si **perde un po' di trasparenza**. Per questo motivo ci sono **2 possibili varianti**: una è questa e in questo senso si **guadagna**

**in sicurezza** (quando un codice invoca add, si è accertati prima che si è in un'istanza Composite) oppure **al contrario rinunciando alla sicurezza** (*non si può fare add o remove su un oggetto semplice*) e **aumentando la trasparenza** scrivendo `add()` e `remove()` in *Component* e quindi **implementate poi nelle sottoclassi lasciando il corpo delle operazioni VUOTO**.

Component potrebbe essere anche *abstract* e le add e remove sono messe nell'interfaccia.

(**ESAME**) L'*operation()* dentro *Composite* come si implementa? Invoca *operation()* su tutti gli oggetti presenti in lista. Rappresenta un ciclo che scorre tutti gli oggetti nella lista. Si invoca il metodo su tutti gli oggetti semplici della lista e, si può aggiungere alla fine, delle operazioni più generiche, come la cancellazione della cartella dopo aver cancellato tutti i file che essa contiene.

### Generalmente:

- Collaborazioni
  - I client usano l'interfaccia di Component per interagire con elementi della struttura composita. Se il ricevente del messaggio è Leaf, la richiesta è gestita direttamente. Se il ricevente è un Composite, questo invia la richiesta ai suoi child e possibilmente avvia operazioni addizionali prima e dopo
- Conseguenze
  - Elementi semplici possono essere composti in elementi più complessi, questi possono essere composti, e così via (ovvero composizione ricorsiva)
  - Un client che si aspetta un elemento semplice può riceverne uno composto
  - I client sono semplici, trattano strutture composte e semplici uniformemente. I client non devono sapere se trattano con un Leaf o un Composite
  - Nuovi tipi di elementi (Leaf o Composite) possono essere aggiunti e potranno funzionare con la struttura ed i client esistenti
  - Non è possibile a design time vincolare il Composite solo su certi componenti Leaf (in base al tipo), invece dovranno essere fatti dei controlli a runtime

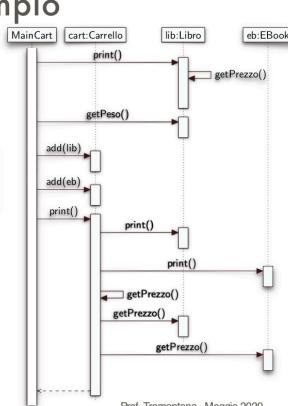
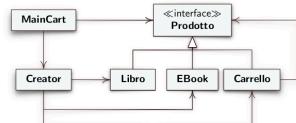
# 23-05-2023

## Conseguenze

- Il Client invoca `operation()` su un'istanza di `Component`.
- Su `Leaf` esegue il metodo totalmente
- su `Composite` esegue la chiamata a `operation()` **su tutti gli elementi** della `List` in `Composite`
- Si ha la possibilità di annidare oggetti in strutture ad albero ed è possibile perchè Composite ha una Lista al suo interno
- Quando si ha un'istanza di `Leaf/Composite` potremmo aver bisogno di sapere il `Composite` (se esiste) che la contiene: In tal caso si può **salvare il riferimento al Composite** che contiene gli oggetti con un metodo `getComposite()`
- Si può avere o **solo trasparenza** o **solo sicurezza** (vedi su con il discorso `add()` e `remove()` implementate nell'interfaccia -> quindi in `Leaf` e `Composite` (in questo caso ci si deve assicurare che si ha un'istanza di `Composite` -> Ho **trasparenza**) oppure implementate solo in `Composite` (**sicurezza**))
- Gestione della cache: In `Composite` si tengono altri attributi che tengono in memoria quello le precedenti operazioni per evitare di invocare le medesime operazioni sui singoli elementi (Come un'operazione `getCosto()` su ogni singolo elemento della lista -> In questo caso potrei salvare un attributo in `Composite` aggiorna il costo totale dei prodotti man mano che aggiungo/rimuovo elementi dalla lista)
  - Se voglio tenere la cache in `Composite` si deve tener conto della possibilità di **invalidare** tali attributi in caso di modifiche delle `Leaf`

### Esempio

- Si vuol gestire un prodotto e un insieme di prodotti (nel carrello) allo stesso modo



## Esempio

```
/**
 * Prodotto svolge il ruolo di Component per il design pattern Composite.
 *
 * I metodi add() e remove() in Prodotto forniscono trasparenza nell'uso di
 * Leaf o Composite, tuttavia non si ha sicurezza.
 */

public interface Prodotto {
 public void print();

 public float getPrezzo();
```

```

 public int getPeso();

 public void add(Prodotto p);

 public void remove(Prodotto p);
 }

 /**
 * Libro svolge il ruolo di Leaf per il design pattern Composite
 */

 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;
 }

 @Override
 public void print() {
 System.out.println("Libro: " + titolo + "\tPrezzo: " + getPrezzo());
 }

 @Override
 public float getPrezzo() {
 return prezzo;
 }

 @Override
 public int getPeso() {
 return peso;
 }

 @Override
 public void add(Prodotto p) {
 }

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

```

```

 /**
 * EBook svolge il ruolo di Leaf per il design pattern Composite
 */

 public class EBook implements Prodotto {
 private String titolo;

```

```

private float prezzo;

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

@Override
public void print() {
 System.out.println("EBook: " + titolo + "\tPrezzo: " + getPrezzo());
}

@Override
public float getPrezzo() {
 return (prezzo * (1 - percSconto() / 100));
}

@Override
public int getPeso() {
 return 0;
}

private float percSconto() {
 return 15;
}

@Override
public void add(Prodotto p) {
}

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

```

```

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

/**
 * Carrello svolge il ruolo di Composite per il design pattern 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("----- ----- ----- ----- -----");
 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) {
 System.out.println("Carrello: add()");
 nestedElem.add(p);
}

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

}

```

```

/**
 * Creator svolge il ruolo di ConcreteCreator per il design pattern Factory Method,
 * fornisce vari metodi factory per istanziare Leaf o Composite
 */

public class Creator {
 private static final String[] libri = { "Gamma Design Pattern", "Fowler Refactoring", "Beck Extreme Programming", "Fowler UML Distilled" };
 private static final float[] prezzi = { 38.5f, 40.0f, 28.5f, 22 };
 private static final int[] pesi = { 650, 700, 300, 200 };

 private static int i = -1;

 public static Prodotto getCarrello() {
 System.out.println("Creator: istanzio Carrello");
 return new Carrello();
 }

 public static Prodotto getLibro() {
 System.out.println("Creator: istanzio Libro");
 aggiornaIndice();
 return new Libro(libri[i], prezzi[i], pesi[i]);
 }

 public static Prodotto getEbook() {
 System.out.println("Creator: istanzio EBook");
 aggiornaIndice();
 return new EBook(libri[i], prezzi[i]);
 }

 private static void aggiornaIndice() {
 if (i < libri.length - 1)

```

```

 i++;
 else
 i = 0;
}
}

```

```

/*
 * L'applicazione che usa il Design Pattern Composite consiste dell'interfaccia
 * Prodotto (Component), e delle classi Libro e EBook (Leaf), della classe
 * Creator che istanzia oggetti, della classe Carrello (Composite) per
 * raggruppare istanze, e del client MainCart che chiama operazioni su Leaf e
 * Composite.
 */

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) {
 System.out.println("\nChiama print su lib");
 lib.print();
 System.out.println("Peso di lib " + lib.getPeso());

 System.out.println("\nInserimento di Libri in Carrello");
 cart.add(lib);
 cart.add(eb);

 System.out.println("\n");
 cart.print();
 System.out.println("\nPeso del Carrello " + cart.getPeso());
 }
}

```

*In questo esempio Creator fa da FactoryMethod ma non è un'alternativa al composite.  
Semplicemente fa comodo averlo*

Con questo esempio:

- si può creare un attributo in Carrello `private float totale = 0;`
- in `getPrezzo()` faccio

```
if(totale>0) return totale; else //faccio le operazioni precedenti
```

- Per **invalidare la cache**:

```
public void prezzoModificato(){
 //invalida la cache
}
```

## Design Pattern Decorator → Strutturale

**Aggiunge o toglie responsabilità** ad un **singolo oggetto** *dinamicamente*: un oggetto ha una responsabilità in più o in meno rispetto a un altro oggetto della stessa classe

- **Responsabilità** sta per "**METODO**", quindi si vuole poter aggiungere/togliere un metodo su ogni singola istanza di una classe

## Intento

In genere i metodi valgono per le classi e non per ogni singola istanza. I metodi implementati sono per tutta la classe e non per ogni singola istanza.

- L'aggiunta di qualcosa rispetto a qualcosa già esistente viene chiamata **WRAP** (che sta per *avvolgimento*).
- Decorator diversi possono essere aggiunti allo stesso oggetto o a oggetti diversi
- Se usassi l'**ereditarietà** sarei **limitato** e quindi **meno flessibile**: si possono **aggiungere metodi con una gerarchia** e quindi estendendo le superclassi
- Si vogliono aggiungere responsabilità quando:

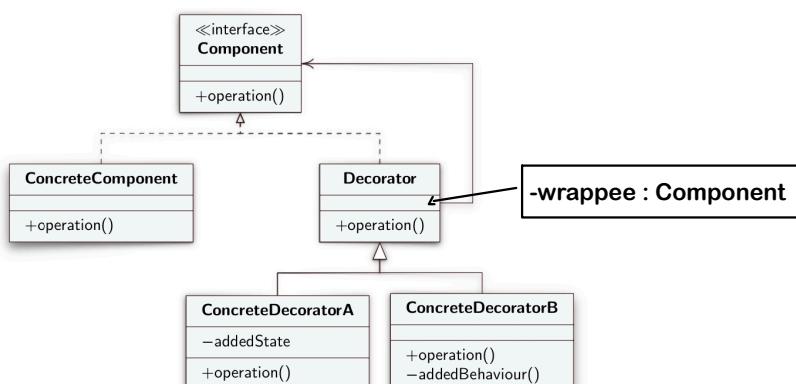
## Esempio

- Si vuol aggiungere la proprietà Bordo ad un componente Testo
- Se si eredita Testo dalla classe Bordo, gli oggetti della sottoclasse avranno questa proprietà, ma non si avrà flessibilità: non si possono avere oggetti senza tale proprietà
- Alternativa flessibile, inserire il componente Testo dentro un oggetto che aggiunge Bordo
- L'oggetto che racchiude, chiamato DecoratorBordo, manda la richiesta al componente Testo e aggiunge altre attività prima o dopo l'invio della richiesta



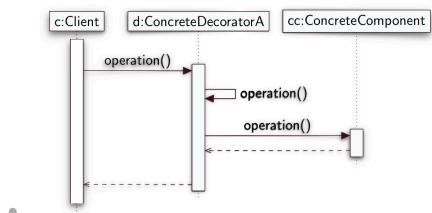
## Soluzione

- Allora si usa Decotator che ha:
  - Component: interfaccia che ha le **operazioni fondamentali da fare**
  - ConcreteComponent: classe che istanzia gli oggetti che ha le **funzionalità base**
  - Decorator: classe che implementa Component e tiene un attributo che è un **riferimento** a Component. Le operazioni che implementa chiamano la stessa operazione sull'istanza che tiene al suo interno. Quindi fa **molto poco**
  - ConcreteDecorator: si può ripetere e implementa la responsabilità aggiuntiva che si desidera.



- `operation()` ha lo stesso nome per tutta la struttura appena descritta e ciò fornisce potenza maggiore, cioè il Client invoca solo `operation()` e quindi le aggiunte sono trasparenti al Client

- Quando si vuole aggiungere un'operazione, si crea un ConcreteDecorator che ha quella singola operazione desiderata. Un'istanza di ConcreteDecorator viene aggiunta a ConcreteComponent e le altre classi non vedono tale aggiunta.



- In ConcreteDecorator si deve chiamare operation() della superclasse per fare in modo che venga richiamato anche il comportamento di base del ConcreteComponent visto che in Decorator ho il riferimento a Component
- Se prima devo eseguire il comportamento base allora, la prima istruzione nel possibile ConcreteDecorator richiama il comportamento base della superclasse.
- 

L'aggiunta viene resa possibile in maniera semplice:

```

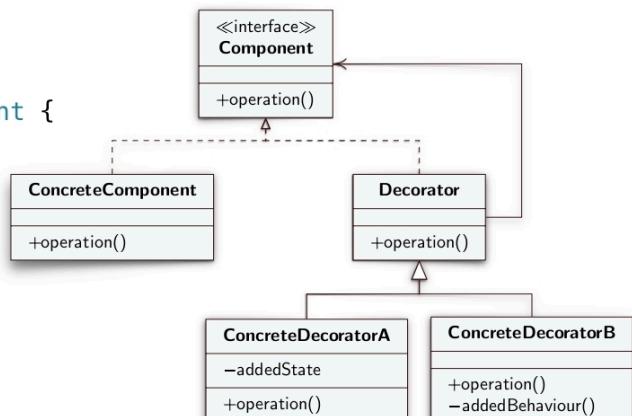
interface Component {
 public void operation();
}

class ConcreteComponent implements Component {
 @Override public void operation() {
 // ...
 }
}

class Decorator implements Component {
 private final Component innerC;
 public Decorator(final Component c) {
 innerC = c;
 }
 @Override public void operation() {
 innerC.operation();
 }
}

class ConcreteDecorator extends Decorator {
 public ConcreteDecorator(Component c) {
 super(c);
 }
 @Override public void operation() {
 super.operation();
 // ...
 }
}

```



```
Component c = new ConcreteDecorator(new ConcreteComponent());
```

```
Component c1 = new ConcreteComponent(); //comportamento base
Component c2 = new ConcredeDecoratorA(new ConcreteComponent()); //comportamento di ConcreteDecorator. Esegue il comportamento operation() di ConcreteComponent e ConcreteDecorator
```

```
Component c3 = new ConcreteDecoratorA(new ConcreteDecoratorB(new ConcreteComponent())); // aggiunge responsabilità più volte sullo stesso oggetto
```

```
Component c4 = new ConcreteDecoratorA(new ConcreteDecoratorA(new ConcreteDecoratorB(new ConcreteComponent()))); //più volte la stessa responsabilità
```

```
Component c4 = new ConcreteDecorator(c1); //fattibile, aggiungo responsabilità dopo aver
```

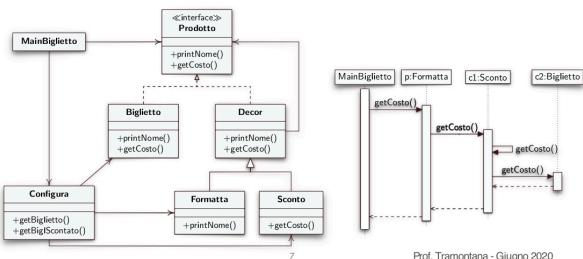
```

definito c1 che ha un comportamento di base
// oppure posso modificare le responsabilità di c1 direttamente
// c1 = new ConcreteDecorator(c1);

```

## Esempio 2

- Vi sono alcuni prodotti, ad es. Biglietto, ognuno con un nome ed un costo; e si hanno diversi modi di calcolare il costo dei prodotti, in base agli sconti e diversi modi di stampare i dettagli
- Si vuol poter combinare a runtime sconti (Sconto) e messaggi (Formatta) sui prodotti, per singole istanze di Biglietto



- Ogni ConcreteDecorator implementa **una sola** operation() dall'interfaccia in modo da avere **più flessibilità**
- Configura crea i vari comportamenti diversi e si comporta da **factoryMethod**

**Se si vuole rimuovere responsabilità ad un oggetto** devo togliere lo **STRATO ESTERNO**:

- per esempio Component c3 = new Bordo(new Bordo(new Testo()));
- chiamo getWrapped() che rimuove lo strato più esterno e quindi si avrà solo una cosa del tipo:  
Component c3 = new Bordo(new Testo());

## Conseguenze

- Si ha maggiore **flessibilità** rispetto all'ereditarietà usata normalmente (quindi si ha *meno ripetizione del codice*)
- Le **responsabilità** possono essere **aggiunte/rimosse dinamicamente**
- Si può aggiungere la **stessa responsabilità più volte**
- Si **spezzettano le responsabilità** senza ingrandire troppo le singole classi (**evitando** anche troppe *istruzioni condizionali*)
- Se si **confrontano i riferimenti** per capire se si tratta la stessa istanza, allora non si ottiene il risultato desiderato.
  - Un **ConcreteComponent** è uguale agli altri ma **non si devono confrontare i riferimenti** (*indirizzi di memoria*).
  - Gli oggetti usati sono gli stessi ma si comportano in maniera diversa perché si aggiungono gli strati esterni.
  - E' più sicuro non confrontare i riferimenti perché a livello di memoria si tratta delle stesse istanze*
- Si avranno tanti oggetti piccoli (ConcreteDecoratorA, ConcreteDecoratorB ecc...) perchè contengono un solo metodo e vengono combinati fra loro per ottenere effetti più complessi.. Il **metodo di partenza deve essere piccolo** e quindi non deve fare troppe cose in modo da poterli ridefinire, cambiare o riusare.
- Le **interazioni fra oggetti** cambiano a run-time in base al *tipo di incapsulamento effettuato*
- Il **codice** che si scrive è **molto semplice** ed è il **Decorator** che **sparge il flusso delle operazioni** chiamando l'operazione stessa sull'attributo che tiene memorizzato al suo interno.

**Questo Design Pattern è uno dei più potenti che vediamo** e magari il più complesso da capire.

#### **Applicazioni :**

- Si utilizza il design pattern Decorator quando c'è la possibilità di assegnare comportamenti extra agli oggetti in fase di esecuzione senza violare il codice che utilizza questi oggetti.
- Si utilizza il design pattern Decorator quando è scomodo o non è possibile estendere il comportamento di un oggetto attraverso l'ereditarietà.

#### **PRO :**

- Puoi estendere il comportamento di un oggetto senza creare una nuova sottoclasse
- Si possono aggiungere o togliere responsabilità ai metodi di un oggetto in fase di esecuzione
- Si possono combinare diversi comportamenti wrappando un oggetto in più decorator
- Single Responsibility : si divide una classe monolitica che implementa molte possibili varianti di comportamento in diverse classi, ognuna per un comportamento specifico

#### **Contro :**

- Difficile rimuovere un decorator specifico dalla pila dei decorator
- Il comportamento di ogni decorator dipende dall'ordine della pila dei decorator
- Codice brutto alla lettura

**26-05-2023**

## Java

- Java è un linguaggio imperativo, permette (*da marzo 2014 circa*) la **programmazione funzionale** e ha introdotto diverse funzionalità.
- La programmazione funzionale è più **espressiva, concisa** e facile da **parallelizzare** rispetto alla programmazione ad oggetti

## Classi anonime

Da Java 1.1 è possibile avere una gerarchia di classi come segue:

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



Questo codice può trasformarsi in un codice corretto che implementa classi anonime. Piuttosto che creare una classe che interfaccia che implementa l'interfaccia (come *Saluti*) si può implementare una **classe anonima**. Non si da un nome alla classe e **non si mette il codice che implementa l'interfaccia in maniera separata.**

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



La classe anonima che implementa Hello è scritta nel main ed è così scritta:

```
//istanzo classe anonima compatibile con Hello
Sera sr = new Sera(new Hello() {
 public void greetings(String s) {
 System.out.println("Ciao, "+s);
 }
});
```

Questa classe la uso solo in quel punto preciso e non può essere usata altrove.

# Espressioni Lambda

E' una funzione anonima, quindi **senza nome**. La sintassi è così formata:

- **Parametri** da passare alla funzione (*a sinistra della freccia*)
- **Corpo della funzione**, cioè il codice che deve essere eseguito quando quella espressione Lambda va in esecuzione (*a destra della freccia*)

Esempio:

```
s ->s=s+1
s -> System.out.println("Ciao " + s);
(p,s) -> //corpo funzione
(x,y) -> x+y;

//se non ci sono parametri in ingresso, allora:
() -> System.out.println("Ciao");

//in caso di insieme di istruzioni a destra della freccia, allora:
() -> {
 System.out.println("Ciao");
 System.out.println("mondo");
}
```

- Solitamente si deve *cercare di evitare di mettere troppe istruzioni* all'interno di una funzione anonima.
- Il **TIPO** non viene specificato fra i parametri e può avere senso solo se si trattano determinati tipi per il corpo della funzione. In pratica il tipo si deduce dal contesto e ha senso solo se è compatibile con le operazioni svolte.
  - I tipi **ci sono** ma sono **SOTTINTESI**
  - *Potrebbero* anche essere *esplicitati* perchè ci sono informazioni derivate dal contesto dove si inserisce l'espressione lambda che definisce i tipi.

I **PARAMETRI IN INGRESSO** sono parametri **FORMALI** (*e non attuali*), per cui non hanno niente a che fare con le variabili scritte prima della dichiarazione della funzione anonima stessa.

## Implementazione interfaccia

Quando si deve istanziare un'interfaccia si usava:

```
Sera sr = new Sera(new Hello() {
 public void greetings(String s) {
 System.out.println("Ciao, "+s);
 }
});
sr.saluti();
```

Usando la funzione anonima diventa:

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

- Si eliminano dei frammenti di codice ridondanti, perchè già si capivano dal contesto (*l'implementazione dell'interfaccia conteneva solo un metodo e quindi non c'è bisogno di ripeterlo*)
- **Ripetere codice inutile** viene detto **BOILERPLATE** (codice ridondante)
- Questa tecnica si è potuta usare perchè l'interfaccia ha **SOLO UN METODO** altrimenti si sarebbero dovute usare le *classi anonime* viste qualche riga prima.
- `greetings(String s)` prende un tipo `String`. `s2` sarà solo di tipo `String` e quindi viene determinato da questo contesto.
  - *Di conseguenza se si passa un parametro non di tipo String, allora ci sarà un errore, ovviamente*

Le interfacce che hanno un solo metodo vengono dette **INTERFACCE FUNZIONALI** perchè possono essere implementate utilizzando proprio le espressioni lambda anonime

## Esempio: ricerca valori in una lista

```
public class Trova {
 private List<String> listaNomi = Arrays.asList("Nobita", "Nobi",
 "Suneo", "Honekawa", "Shizuka", "Minamoto", "Takeshi", "Gouda");

 // in stile imperativo
 public void trovaImper() {
 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");
 }

 // in stile dichiarativo
 public void trovaDichiar() {
 if (listaNomi.contains("Nobi")) System.out.println("Nobi trovato");
 else System.out.println("Nobi non trovato");
 }
}
```

5

Prof. Tramontana - Maggio 2022

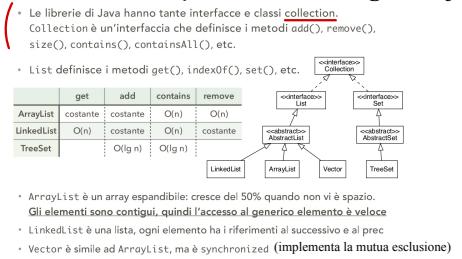
Per controllare *se esiste un elemento all'interno di una lista* si potrebbe usare

- `trovaImper()` che è un metodo *imperativo* che scorre la lista e scorre la lista con un `for` avanzato e, per ogni elemento, viene fatto un confronto.
  - In questo caso si devono scrivere molte cose e, per capire bene il significato, bisogna leggere molto testo.
- Potrei usare un **metodo DICHIARATIVO** `trovaDichiar()` che usa `contains()`. Si sposta la funzionalità che si voleva avere dentro la libreria Java.
  - Il ciclo `for` esiste ugualmente, sì, ma dentro le librerie che *nasconde i dettagli* e quindi il programmatore non deve esplicitare il ciclo `for`

## Stile funzionale

- E' conforme allo stile dichiarativo e aggiunge **funzione di ordine superiore** (in `contains()` si può passare solo il parametro da contenere e non si può passare un pezzo di codice).
- Permette di implementare funzioni che lavorano su altre funzioni che sono passate come parametri in ingresso.
- Quando si implementa un metodo, se c'è bisogno, **si può passare una funzione** (oltre a un singolo dato) a quel metodo.

- I metodi che prendono **in ingresso funzioni**, si parla di funzioni di **ordine più alto**



## Metodi di default

Il metodo `stream()`, a partire da una lista, permette di accedere alla programmazione funzionale.

`stream()` restituisce un tipo `Stream` che mette a disposizione un metodo che prende **in ingresso funzioni lambda** (`filter()` ecc..) e, di base, `stream()` **non fa altro**.

- Serve un modo per trasformare una lista in un qualcosa che si adatti alla prog. funzionale.  
Questo lavoro lo fa proprio `stream(): Stream`
- `stream()` è un metodo di `Collection` ed è implementato in `Collection` stesso e rappresenta un'eccezione per il tipo interfaccia (come regola di base) e si parla appunto di **METODI DI DEFAULT**
- i metodi di default non possono agire sullo stato perché *l'interfaccia non ha uno stato*

Si può eseguire la conta di un certo numero di elementi in una lista in maniera **FUNZIONALE**:

```
List<String> nomi = List.of("Nobita", "Nobi", "Suneo"); // crea la lista
long c = nomi.stream().filter(s->s.equals("Nobi")).count();
```

- `filter(s->s.equals("Nobi"))` seleziona alcuni elementi della lista sulla base della condizione della funzione lambda. Deve tornare `true` o `false`, quindi un **BOOLEANO** e quindi si parla di un predicato, detto **Predicate**
  - Se l'espressione lambda non ritorna un booleano, allora vi sarà un errore di compilazione
  - `s` è di tipo `String` perchè si ha `List<String> nomi;`
  - La funzione viene applicata a tutti gli elementi della lista
  - All'uscita di `filter()` si ha un tipo `Stream` che contiene solo gli elementi presenti nello stream in ingresso e che soddisfano la condizione il *Predicate*
  - `count()` conta gli elementi presenti nello Stream di uscita da `filter()`.
  - Quindi `count()` e `filter()` sono modi messi a disposizione dal tipo `Stream`.

Il pezzo di codice sopra è equivalente al seguente:

```
Stream<String> s1 = nomi.stream();
Stream<String> s2 = s1.filter(s->s.equals("Nobi"));
long c = s2.count();
```

- `s2` non contiene nulla
- Le operazioni terminali (`count()`) fanno finire **FORZATAMENTE** le operazioni intermedie (`filter()`, quindi `filter()` non si applica fino a quando non si conclude con un'operazione terminale)
- Dopo la `filter()` e quindi dopo aver selezionato gli elementi bisogna fare qualcosa altro, per esempio, contare o fare altro. Alcune operazioni sono **TERMINALI**(dette *eager*) e alcune **NON TERMINALI** o **INTERMEDIIE** (dette *lazy*).

- Uno `Stream<T>` non viene riempito fino a quando non si hanno le operazioni terminali

Le operazioni che **tornano uno Stream<T>** sono *operazioni intermedie*. Mentre le operazioni che **tornano un tipo NON Stream<T>**, allora si tratta *operazioni terminale*.

- \*Dopo aver **UTILIZZATO** lo Stream, quindi invocata un'operazione terminale, esso **SI CHIUDE**.
- Per tale motivo uno Stream può essere usato **SOLO UNA VOLTA**

## Esempi con Filter

- Contare quanti elementi della lista `nomi` hanno lunghezza 5 caratteri
  - Si noti che non si può usare il metodo `contains()` di `List` (che è invece utile per verificare se una lista contiene un certo elemento)

```
long c = nomi.stream()
 .filter(s -> s.length() == 5)
 .count();
```

- L'espressione lambda `s -> s.length() == 5` ha in ingresso `s`, ovvero un elemento dello stream; su `s` si invoca `length()` (metodo di `String` che restituisce la lunghezza di `s`), quindi si valuta se è pari a 5
- Contare quanti elementi della lista `nomi` sono stringhe vuote

```
long c = nomi.stream()
 .filter(s -> s.isEmpty())
 .count();
```

- `isEmpty()` è un metodo di `String` (non ha parametri di ingresso e restituisce un boolean). Tale metodo è passato a `filter()`, e sarà chiamato su ciascun elemento dello stream, quindi ciascun elemento dello stream è valutato da `isEmpty()`

## Tipo `Predicate<T>`

Piuttosto che scrivere ogni volta il predicato dell'espressione Lambda, si potrebbe salvare in un'apposita variabile di tipo `Predicate<T>` per poi venire usato in un'altra espressione lambda. Chiaramente `Predicate` avrà esclusivamente un valore booleano.

```
Predicate<String> p = s->s.equals("...");

str.filter(s->s.equals("...."));

//equivale a
str.filter(p); //operazione non terminale
```

`Predicate` è un'interfaccia funzionale e quindi implementa *SOLO UNA FUNZIONE* che si chiama `test(Object obj)`

## Metodo `reduce()`

E' applicato a uno Stream e fornisce in uscita un ***UNICO risultato*** che è **calcolato sull'intero Stream**. Lo **valuta interamente** sulla base dell'espressione lambda passatagli.

```
reduce(T identity, BinaryOperator<T> accumulator)
//identity compatibile con il tipo passato allo Stream. Vale 0 perchè non cambia il valore finale dello Stream (in una somma il valore nullo è 0).

Predicate<Integer> positive = x -> x>=0;
Stream<Integer> result = Stream.of(2,5,10,-1).filter(positive);

reduce(0, (accum,v) -> accum+v); // ritorna 2+5+10=17
// passo 0 -> accum = 0; v = primo valore della lista
// passo 1 -> accum = valore precedente della somma; v = secondo valore della lista
// passo 3 -> accum = valore precedente della somma; v = terzo valore della lista
// passo 4 -> ecc...
```

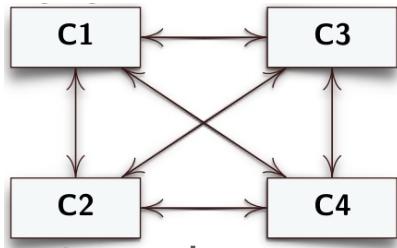
- Identità = non deve cambiare il risultato finale.
  - Nelle somme è 0
  - Nei prodotti è 1
  - Nelle stringhe è " "
  - ecc...

# Design Pattern Mediator

## Intento

Mediator è un design pattern **comportamentale** che consente di ridurre le dipendenze caotiche tra gli oggetti. Il modello limita le comunicazioni dirette tra gli oggetti e li costringe a collaborare solo tramite un oggetto mediatore.

- Gli oggetti vogliono interagire fra loro ma interagiranno tramite un oggetto che rappresenta le interazioni fra oggetti.  
*Gli oggetti sono legati ma non si conoscono fra loro.*
- Si promuove lo scoppioamento degli oggetti che hanno bisogno di interagire fra loro.



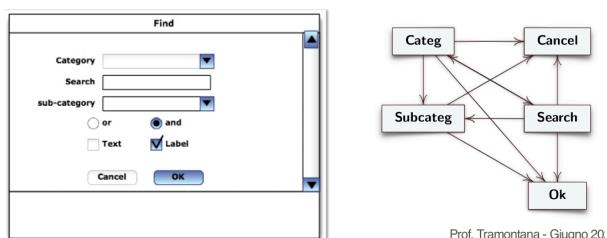
- Un oggetto **C1** comunica un risultato a **C2** e tutti gli altri oggetti. Le **interazioni avvengono fra oggetti**.
- Ogni classe è costretta a conoscere le altre classi
- Se si deve **modificare** **C3**, per esempio, **per via delle dipendenze**, devo **modificare un po'** di **C1**, **C2** e **C4** di **conseguenza**

*Quindi la modifica di una classe comporta la modifica di tutte le altre classi che dipendono da essa o da quale dipende tale classe*

- Il **ri-uso** di queste classi **diventa difficile** perchè esse sono legate fra loro e quindi vi è una **LIMITAZIONE** e si tratta di un **SISTEMA MONOLITICO**
- Si devono **eliminare** le interazioni/dipendenze fra le classi
- Il comportamento complessivo (di interazione) fra le classi, lo esprimo tramite un oggetto a sé stante, cioè tramite il **MEDIATOR**
- Gli oggetti risultano più isolati e \*non più con forti dipendenze e quindi più facili da usare\*

## Esempio che mostra la motivazione

*Una finestra di dialogo ha dei bottoni, caselle per inserire del testo e altre parti..*



Prof. Tramontana - Giugno 20

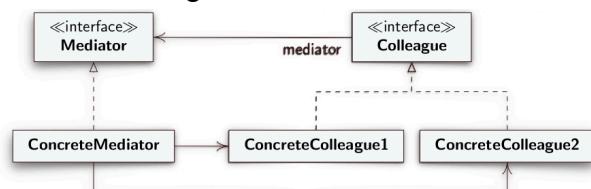
- Ci sono tante classi che vengono istanziate e che rappresentano ogni parte della finestra
- Le istanze sono necessarie perchè, quando l'utente interagisce, il comportamento delle parti di finestra dipendono, appunto, dall'interazione stessa
- Se l'utente sceglie *Category*, la *sub-category* deve essere **riempita opportunamente** in base alla categoria scelta dall'utente
- *Un altro esempio di dipendenza* è: il tasto OK è abilitato solo se le altre parti di finestra sono "state riempite"
- Ogni classe *chiama i metodi* di **TUTTE** le altre classi, quindi vi è una **forte dipendenza fra loro**

Ogni classe implementata deve **comunicare dati a tutte le altre classi** che servono per gestire bene la finestra e aggiornare la visualizzazione

*In questo caso vi è un sistema monolitico: se si vuole un'altra finestra, riusare il codice risulta complicato perchè, magari, nella nuova finestra non vi sono più delle parti che nella vecchia c'erano*

## Soluzione

Si usa il Design Pattern Mediator:



- si usa un componente (`ConcreteMediator`) che implementa le interazioni fra i vari oggetti
- Le **classi NON comunicano più fra loro** ma lo fanno tramite il Mediator
- **Colleague (C)** conosce l'interfaccia **Mediator (M)**
- Il **ConcreteMediator (CM)** conosce i **ConcreteColleague (CC)**
- Quando i CC finiscono il loro lavoro e devono avvisare gli altri, chiamano un metodo definito in M
- CM deve sapere qual è l'avviso dato dai CC e sapere chi deve avvisare fra tutti i CC
- Le interazioni, quindi, vengono messe all'interno di M

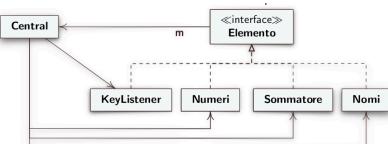
## Conseguenze

- Tutte le dipendenze del gruppo di oggetti che si scambiano i vari risultati, sono gestite dal CM
- Si è tolto parte di codice che serviva alle interazioni fra singoli oggetti e queste parti di codice vengono messe all'interno di CM
- I CC sono più **RIUSABILI** che devono solo conoscere l'interfaccia M

- Ogni classe `CC` non conosce l'esistenza di altre `CC` (`CC1` *non conosce* `CC2`) quindi ogni singolo `CC` si può riusare
- Solitamente i `CM` non sono **RIUSABILI** perchè rappresenta le singole interazioni in **QUELLA DETERMINATA APPLICAZIONE**
- I `CC` sono meno specifici, appunto perchè non conoscono i codici degli altri `CC`

## Esempio applicazione di Mediator

Si ha un'interazione con l'utente (*senza interfaccia grafica*, cioè diversamente dall'esempio di prima). Ogni cosa che l'utente ci dice, può far scatenare eventi a catena.



- KeyListener `KL` legge i dati da tastiera. Entra in azione solo quando si deve leggere dallo standard input
- Se è un dato numerico, esso serve a Numeri `Nu` e a Sommatore `S`
- Il *ConcreteMediaor CM* è `Central` c
- Se è un dato testuale, esso serve a `Nomi` `No`
- `Nu` valuta il numero fornito e decide se chiedere un altro numero in input da `KL` (tramite c)
- `No` valuta la stringa decide cosa fare
- `S` può tenere da parte i numeri dati e può dare un risultato dato dalla somma dei numeri forniti in input

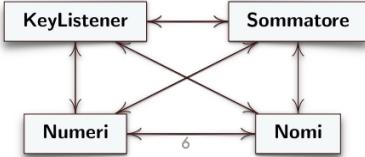
*Generalmente...*

- Si ha bisogno da varie classi e ognuna di esse fa qualcosa di diverso
- Le interazioni fra le classi sono gestite da `Central` c
- Elemento `E` rappresenta il *Colleague*
- c fa da *ConcreteMediator* ed è un'**ALTERNATIVA AL MEDIATOR senza interfaccia**
- Questo design pattern **non suggerisce un nome dei metodi perchè è molto dipendente** da come esso si vuole usare

Lo **scopo dell'applicazione** è:

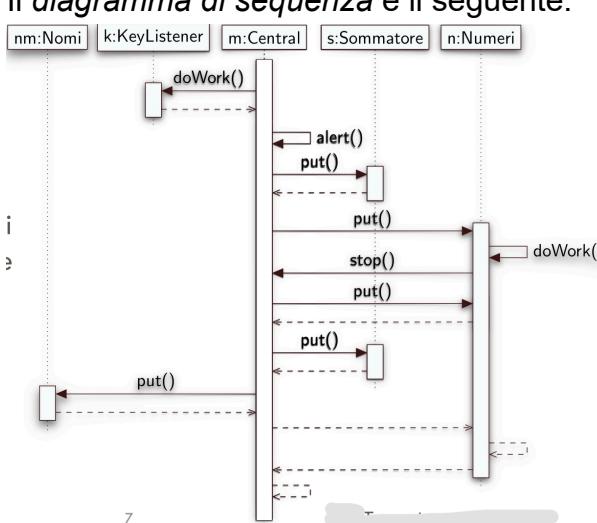
- L'utente deve rispondere a 2 domande: se risponde alla prima deve fornire un numero. Se risponde alla seconda deve fornire una stringa
- Il **Mediator** si occupa **ANCHE dell'istanziazione** degli oggetti delle varie classi
- Ogni `Colleague` hanno un costruttore che prende in ingresso un parametro di riferimento a `Central`

Se non si usasse il design pattern mediator, si avrebbe la seguente situazione:



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

Il *diagramma di sequenza* è il seguente:



[Esempio di codice](#)

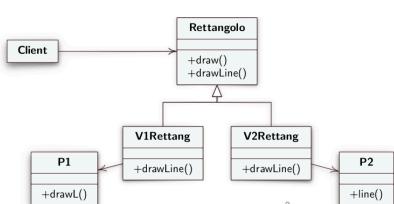
## Design Pattern Bridge

### Intento

Bridge è un design pattern **strutturale** che consente di suddividere una classe di grandi dimensioni o un insieme di classi strettamente correlate in due gerarchie separate, astrazione e implementazione, che possono essere sviluppate indipendentemente l'una dall'altra.

- Un'astrazione può avere diverse implementazioni e in questo caso si usa l'ereditarietà
- Si forniscono algoritmi diversi per quell'unica astrazione che si è pensato
- Si deve disaccoppiare un'astrazione dalla sua implementazione così che le due possano variare indipendentemente

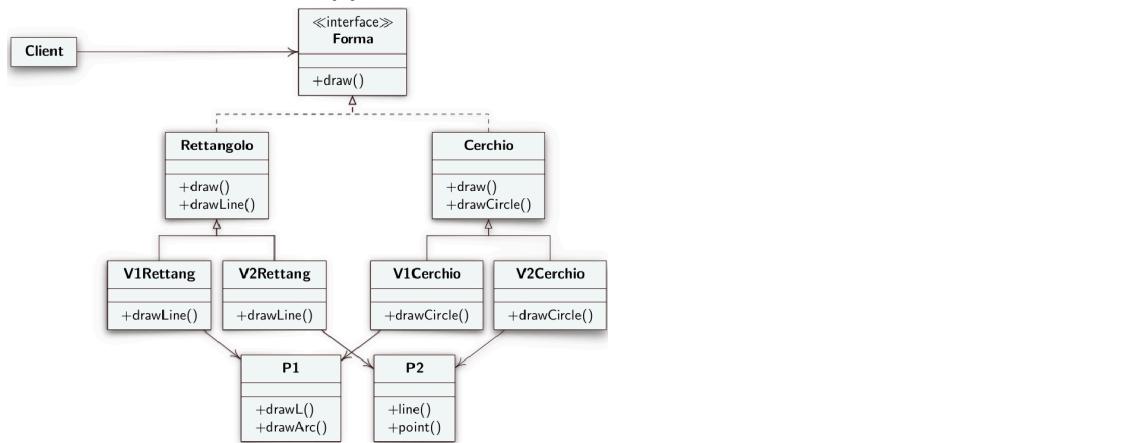
### Esempio



- Si ha bisogno dell'astrazione *Rettangolo* e si vuole disegnare

- Quando si deve disegnare si può avere bisogno di scegliere fra più istruzioni diverse per il disegno
- Quando si cambia *libreria di disegno*, si ha bisogno di un pezzo di codice che chiami i metodi giusti della libreria selezionata
- Una libreria (detta anche *piattaforma P1*) mette a disposizione `drawL()` per disegnare una linea
- Un'altra libreria (*P2*) fornisce `line()` e si deve invocare su un'altra classe diversa.
- Si distingue il codice e fra le 2 librerie non si mischiano fra loro
- Le due classi `V1Rettang` e `V2Rettang` sono particolare implementazioni dell'astrazione `Rettangolo`

Ma se, all'interno dell'applicazione, si vuole anche un Cerchio, si avrà:



In questo caso, il codice è modulare in 2 classi diverse, sia in Rettangolo che in Cerchio  
Se si volesse introdurre un'altra forma geometrica (Triangolo) allora:

- Si deve implementare Triangolo con le 2 possibili classi (o più).
- Per ogni astrazione in più di cui ho bisogno, devo implementare 2 classi in più e così via...
- Se non basta interfacciarsi con 2 librerie e serve una terza libreria `P3`, allora, per ogni forma pensata, serve un'altra classe che implementa i metodi a questa nuova libreria (`V3Rettang`, `V3Cerchio`, `V3Triangolo` che implementano i metodi di `P3`)
- Si ha una ***POLIFERAZIONE DI CLASSI***

Quindi....

e non esponenziale

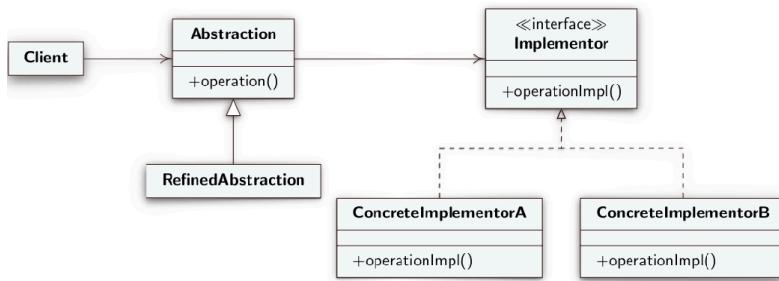
- Per ogni variazione da introdurre, ***si vorebbe un incremento lineare del numero di classi.*** (Aggiunte di nuove astrazioni o di nuove piattaforme (librerie) da usare)
- Ogni classe è legata ad una certa piattaforma in modo permanente, cioè un'istanza di `V1Rettang` non può usare una piattaforma diversa da `P1`

## Soluzione

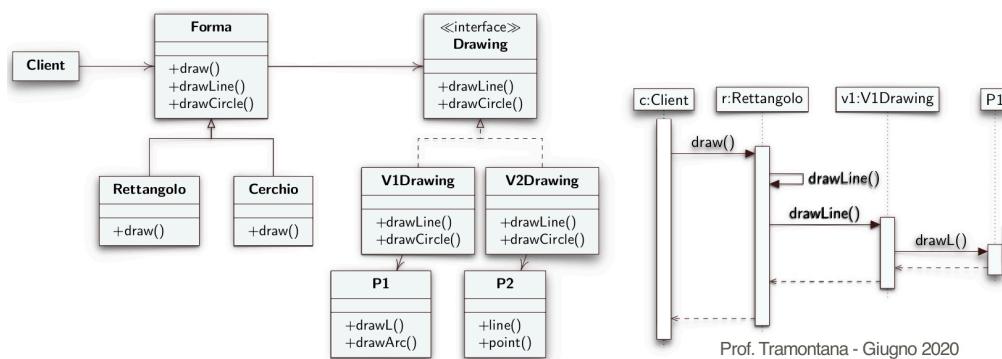
Si usa il Design Pattern Bridge:

**L'astrazione** è un livello di controllo di alto livello per alcune entità. Questo livello non dovrebbe svolgere alcun lavoro reale da solo, dovrebbe delegare il lavoro al livello di **implementazione**.

- Si crea un'Abstraction A generale che tiene in riferimento un oggetto Implementor I e, userà questo riferimento per essere implementata. Qui vengono inseriti i metodi che serviranno essere usati
- Si usano delle sottoclassi di A, chiamate RefinedAbstraction RA che usano i metodi implementati in A.
- Si hanno degli I e dei ConcretelImplementor CI e questi ultimi forniscono le operazioni concrete
- I è un'interfaccia che rappresenta i vari CI
- i CI è uno per ciascuna piattaforma da pilotare. Ogni CI sa chiamare i metodi giusti da usare a run-time
- I fornisce dei metodi usati da A che tiene un riferimento di I, appunto



- Solo A conosce I
- CI si servono delle piattaforme che si vogliono supportare e ognuno di loro **conoscono una sola piattaforma**
- A è una classe che definisce e implementa **operation()** e fa da superclasse per RA
- RA usa **operation** di A
- Il client si lega all'A ma si può legare anche a RA



- Le RA chiamano metodi di A
- Per implementare i **draw()** nelle RA mi servo dei metodi implementati nella classe Forma
- Se si deve supportare una piattaforma aggiuntiva, si deve creare solo una nuova classe V3Drawing che chiamerà metodi della piattaforma P3 e dovrà fornire i metodi utili all'Abstraction
  - Si deve solo fornire all'abstraction il riferimento alla nuova classe V3Drawing
- L'aggiunta di una piattaforma equivale all'aggiunta di una sola classe di Drawing

Se ci serve la classe *Triangolo* che avrà al suo interno `draw()` che chiamerà 3 volte `drawLine()` per le 3 linee da disegnare

## Conseguenze

- Bridge permette a una implementazione di non essere connessa permanentemente a una interfaccia, l'implementazione può essere configurata e anche cambiata a runtime
- Il **disaccoppiamento** permette di cambiare l'implementazione **senza dover ricompilare Abstraction ed i Client**
- Solo certi strati del software devono conoscere *Abstraction* e *Implementor*
- I *Client* non devono conoscere
- Le gerarchie di *Abstraction* e *Implementor* possono **evolvere in modo indipendente**

## Esempio minimale di Bridge

```
// Forma è una Abstraction
public class Forma {
 private Drawing impl;
 public void setImplementor(Drawing imp) { this.impl = imp; }
 public void drawLine(int x, int y, int z, int t) {
 impl.drawLine(x, y, z, t);
 }
}

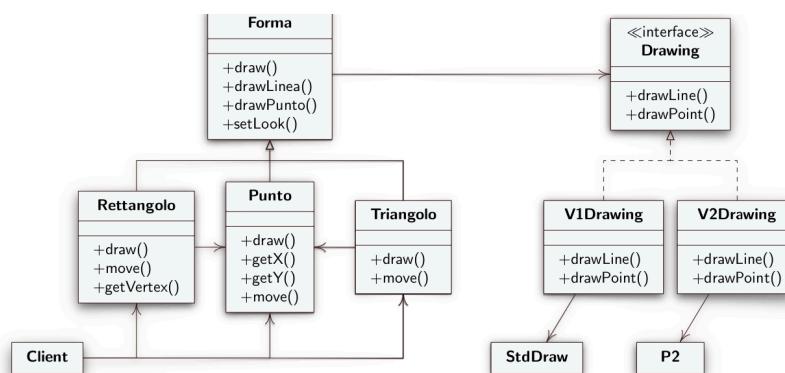
// Drawing è un Implementor
public interface Drawing {
 public void drawLine(int x1, int y1, int x2, int y2);
}

// Rettangolo è una RefinedAbstraction
public class Rettangolo extends Forma {
 private int a, b, c, d;
 public Rettangolo(int xi, int yi, int xf, int yf) {
 a = xi; b = yi; c = xf; d = yf;
 }
 public void draw() {
 drawLine(a, b, c, b); drawLine(a, b, a, d);
 drawLine(c, b, c, d); drawLine(a, d, c, d);
 }
}
```

9

Prof.

## Versione più completa



# Design Pattern Chain of Responsibility

## Intento

Chain of Responsibility è un design pattern **comportamentale** che consente di passare le richieste lungo una catena di gestori. Alla ricezione di una richiesta, ciascun gestore decide di elaborare la richiesta o di passarla al gestore successivo della catena.

- Si deve evitare di accoppiare il mandante di una richiesta con il ricevente.
- Ebbene non si conosca il mandante, la richiesta viene mandata a catena a tutti.
  - *Chi non riesce a gestire la richiesta la manda alla successiva catena (Chain)*

*Quando si tocca un bottone (Conferma) esso cosa fa? Conferma i dati inseriti oppure fa anche altre cose?*

## Esempio

Se si ha un sistema che ha la possibilità di fornire *Aiuto* sugli elementi puntati dal mouse (*classica finestra a comparsa*). La funzionalità di Aiuto è inserita su precisi elementi non subito capibili... Se invece si usa un pulsante Aiuto generico, ci si aspetta che qualcuno risponda a tale richiesta (*può rispondere -> o l'elemento specifico che l'utente ha selezionato col cursore oppure, se l'elemento selezionato dall'utente non ha delle informazioni di aiuto da dare, risponde chi di dovere, seguendo una catena di richieste*)

## Motivazione

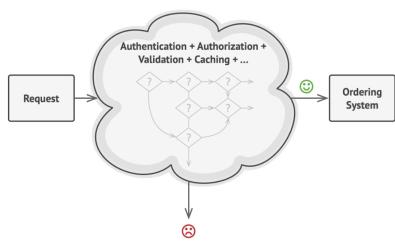
*Formalizzando l'esempio sopra descritto...*

Consideriamo una interfaccia utente dove l'utente può richiedere aiuto su parti dell'interfaccia. Per es. un bottone può fornire informazioni di aiuto.

Se non esiste un'informazione specifica allora il sistema dovrebbe fornire il messaggio d'aiuto del contesto più vicino (ad es. la finestra)

- Il problema: l'oggetto che fornirà il messaggio d'aiuto non è conosciuto dall'oggetto (es. Button) che inizia la richiesta
- Disaccoppiare mandante e ricevente

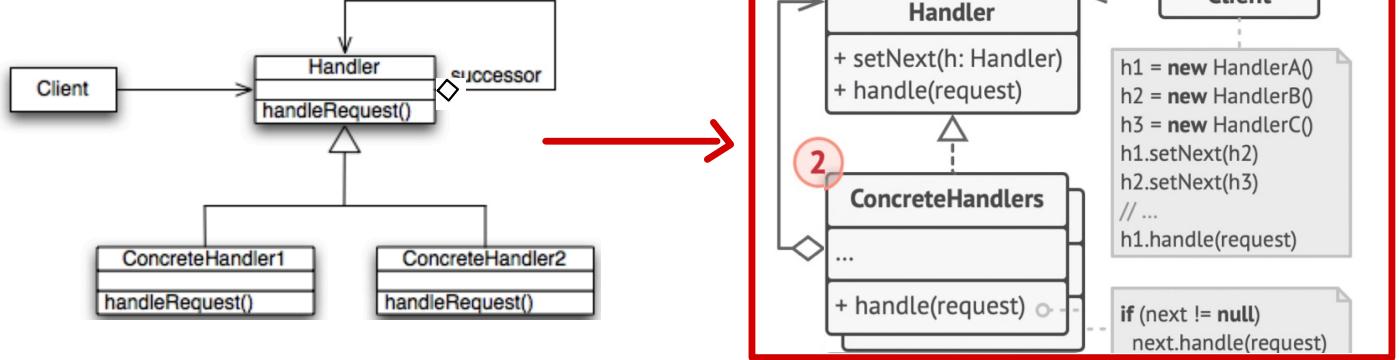
## Esempio 2: autenticazione



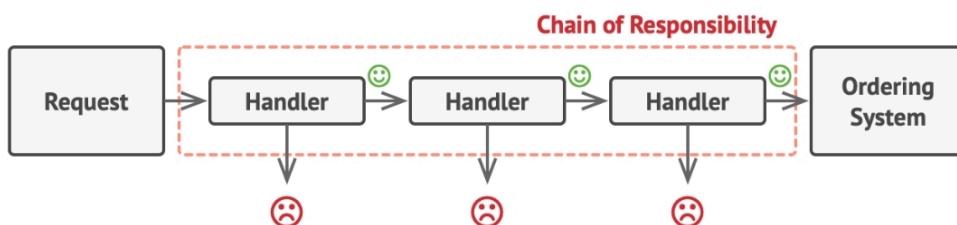
*The bigger the code grew, the messier it became.*

## Soluzione

La struttura del Design Pattern è la seguente:



- **Handler** definisce l'**interfaccia** per gestire le richieste
- La **handleRequest()** in **Handler (H)** potrebbe essere "Mostra aiuto". Questa funzione **gestisce una richiesta**
- Le **ConcreteHandler1 /2 (CH)** implementeranno **handleRequest()**
- Ogni elemento conosce un altro elemento dello stesso tipo.
- A run-time ci sarà un'istanza di **CH** e quest'ultimo conoscerà un altro H (che sarà poi **CH1 o CH2**) che rappresenterà **il successivo elemento della catena**
- Quando **CH1** deve eseguire la **handleRequest()** si controlla se egli stesso può gestirla. Se non può gestirla la rimanda al successivo elemento della catena, richiamando **handleRequest()** proprio su **successor** (*visto che lo conosce*)
- **handleRequest()** potrebbe avere 0+ parametri
- **Il Client non conosce quale CH specifico risponderà alla richiesta.**
  - Egli inizia la catena di richiesta su un CH iniziale specifico
  - non è detto che risponda proprio la classe sulla quale è stata chiamata la richiesta inizialmente



*Handlers are lined up one by one, forming a chain.*

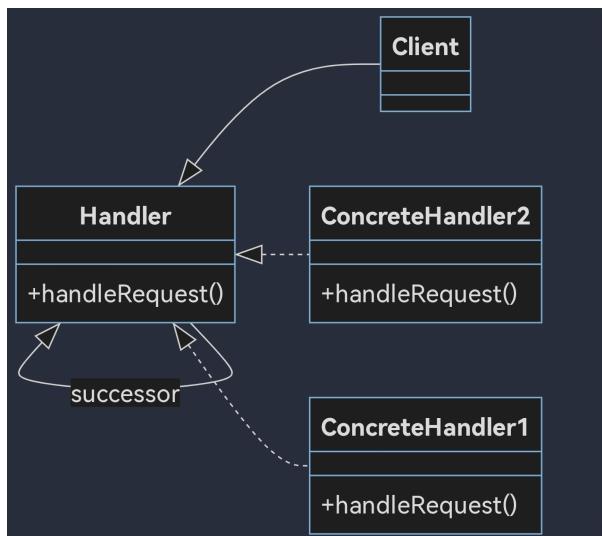
## Conseguenze

- Un oggetto della catena non conosce la struttura COMPLETA della catena ma **esclusivamente il successivo elemento**
- Si possono **aggiungere/rimuovere elementi** dalla catena per aumentare/diminuire la propagazione della la richiesta in questione manipolando la *lista concatenata*
- La propagazione della richiesta può arrivare alla fine della catena. In questo caso nessun elemento sa rispondere e quindi la richiesta potrebbe non essere MAI soddisfatta

## Implementazione

- L'operazione `handleRequest()` definita in H potrebbe essere un'operazione non solo definita ma anche un richiamo all'elemento successivo.
- H potrebbe anche non essere un'interfaccia (ma generalmente è meglio che lo sia) proprio per casi come questi
- Questo Pattern può essere **implementato in una gerarchia già esistente**, come nel *Composite*: Impone ai vari *ConcreteComponent* di implementare un'operazione di risoluzione/propagazione di una richiesta
- Alcune classi facenti parte di Composite, potrebbero implementare diversi Design Pattern in base al proprio ruolo (chiaramente questo è valido per applicazioni non banali)
- Ogni CH segue una **CONVENZIONE** sui dati che determina il tipo di parametro (se definito) passato alla richiesta in questione. Quindi ogni CH devono sapere come interpretare il parametro passato a `handleRequest()` (quindi ci deve essere **coerenza**)

## Diagramma delle classi Mermaid



## Design Patter Prototype

### Intento

- E' un design pattern **CREAZIONALE**
- Quando si creano le istanze, se ne usano altre che fa da base per la creazione di altre istanze. Piuttosto che specificare la classe di cui avere l'istanza, si usa un'istanza già presente, quindi iniziale, che fa proprio da **PROTOTIPO** da cui copiare. La nuova istanza si porta i dati del prototipo

### Motivazioni

- Si ha un'istanza a run-time che ha vari dati.
- Leggo una classe che vuole una nuova istanza (richiedente) alla classe che viene usata per creare l'istanza

Esempio: Se devo istanziare Client, si deve istanziare Student.

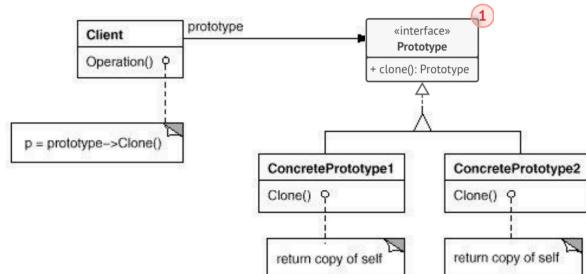
```
public class Client{
 // ...
 v = new Student();
}
```



In questo caso si ha:

\*Si vuole evitare di avere questa dipendenza. Si potrebbe usare un `FactoryMethod()` ma si genererebbe una vasta gerarchia di Creator per slegare il Client dal ConcreteProduct

## Soluzione



La classe **Concrete Prototype** implementa il metodo di clonazione. Oltre a copiare i dati dell'oggetto originale nel clone, questo metodo può anche gestire alcuni casi limite del processo di clonazione relativi alla clonazione di oggetti collegati, al districare dipendenze ricorsive, ecc.

- **Prototype (P)** definisce l'operazione `clone()` che copia l'istanza
- Ogni **ConcretePrototype (CP)** implementa `clone()`
- **Client** si lega a P
- La gerarchia di CP è equivalente alla gerarchia dei *ConcreteProduct* (*in Factory*) e assomigliano ai "prodotti" che ci servono. Qui, però, serve il metodo `clone()`
- P deve includere la definizione delle operazioni per mantenere la compatibilità con i sottotipi CP. Cioè devono esistere altre operazioni (implementate nei sottotipi) che specificano proprio quel P
- `clone()` fa le seguenti operazioni: si **crea** un'istanza e **copio** lo stato iniziale nella nuova istanza
- I dati e gli algoritmi implementati in CP1 sono diversi rispetto a tutti gli altri CP, appunto perchè non *si fa duplicazione di codice*.
  - Ne segue che `clone()`, su ogni CP, è diversa per ogni istanza e quindi la copia è diversa e non può essere unica per tutti i CP
- Il Client conosce il tipo più generale ma a run-time, **tramite clonazione**, avrà il **tipo più specifico** (*quindi senza indicare la classe esplicitamente*). Da questo punto, quindi, verrà creata l'istanza di uno specifico CP ma, comunque, il Client conosce il tipo più generico P

Si evita di legare chi ha bisogno della nuova istanza con la classe che ha quella particolare istanza

# Conseguenze

- Dà la possibilità di registrare nuovi P a runtime
- Il client può usare classi specifiche (sottoclassi) senza conoscerle direttamente
- Si può aggiungere un **REGISTRO** che tiene i prototipi che si possono usare: La classe generica può cercare nel registro (tramite una chiave) e istanziarne uno preciso
- Se diversi oggetti hanno uno stato diverso, rappresentano dei comportamenti diversi. Di conseguenza, pur avendo CP1 (per esempio) la si fa partire da stati diversi. Visto che lo stato cambia, allora clone() sarà diverso (anche per la stessa classe). La programmazione è semplificata per il numero di linee di codice che si scrivono.
  - Si usa (tramite registro o clonazione) da quale stato partire per avere quello stato diverso
  - Si può definire uno **stato iniziale** e conservarlo
- Se in un CP ci sono attributi (tipi riferiti ad altri oggetti) allora si ha una composizione. Un nuovo prototipo può essere usato per tene una struttura diversa di oggetti, appunto

# Implementazione

Si può avere una classe apposita RegistroDiPrototipi che tiene un'associazione (chiave:valore) che permette ai client di immagazzinare e recuperare i prototipi, prima di clonarli

Quando si ricopia lo stato:

- si potrebbero avere dei **riferimenti circolari**: nello stato da copiare potrebbero essere dei riferimenti successivi fino a ritornare al punto di partenza (*loop infinito*)

La copia potrebbe essere superficiale (**shallow copy**) o profonda (**deep copy**):  
(si suppone di avere una classe A con 2 attributi: `int x = 5; B v = new B();` e si vuole clonare. A ha una dipendenza con B)

- Shallow copy: L'attributo di tipo primitivo x, verrà ricopiato nell'istanza clone e, inoltre v si riferirà alla stessa identica porzione di memoria.
  - Questo effetto è indesiderato perchè si vorrebbe che il clone avesse la propria dipendenza con B (separata dalla precedente). In questo caso serve la *Deep copy*, quindi bisogna implementare manualmente la clonazione.
- Deep copy: copia in modo profondo i dati evitando dipendenze "fra classi"  
Nello specifico duplica le aree di memoria a cui fanno riferimento i puntatori.

Command è un design pattern comportamentale che trasforma una richiesta in un oggetto autonomo che contiene tutte le informazioni sulla richiesta. Questa trasformazione consente di passare le richieste come argomenti del metodo, ritardare o accodare l'esecuzione di una richiesta e supportare operazioni annullabili.

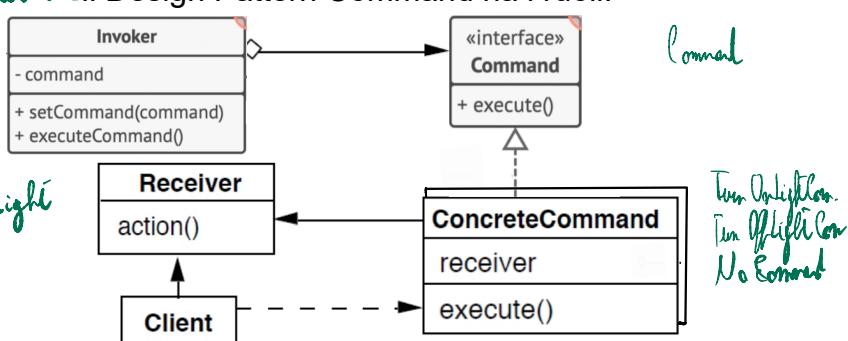
# Design Pattern Command → Comportamentale

## Intento

- Una **richiesta**, di solito, è **una chiamata di metodo**
- La richiesta può diventare un oggetto che incapsula informazioni riguardo a chi potrà portare a termine la richiesta
  - *Quindi la richiesta è fatta tramite un oggetto e qualcun altro saprà come portare a termine la richiesta*
- Si può **SVINCOLARE** richiedente da **chi** effettivamente **esegue la richiesta**
- Si può **eliminare la conoscenza dentro il richiedente** così che egli **non conosca chi può ricevere la richiesta** ma si limita a formularla. Quindi prepara un oggetto Richiesta.
- La **richiesta** può essere **passata ad altri oggetti intermedi** in modo da poter essere **manipolata e modificata**, passando quindi per altri oggetti
- Il richiedente prepara un oggetto di tipo **Command** (quindi vi si lega) e tramite l'interfaccia di Command, si possono invocare delle operazioni su questo tipo.
  - *Il codice di Command è scritta in modo da sapere come gestire il comando in sè*
  - *Command, quindi, conosce i dettagli e i modi di operare su un Target ricevente*
- Ne segue anche che il **ricevente** deve sapere **come trattare la richiesta** e quindi **come soddisfarla**

## Soluzione

Riassunto: Il Design Pattern Command ha i ruoli:



- **Command (C)** rappresenta un'interfaccia che mette a disposizione `execute()`
- **Invoker (I)** conosce un'istanza di C e quando gli serve chiama `execute()` sull'istanza
- **Receiver (R)** è quello che effettivamente svolge il lavoro. Lo fa grazie alle chiamate che gli vengono fatte e che sono scritte dentro un **ConcreteCommand (CC)**. Quindi CC conosce effettivamente come chiamare e utilizzare i R
- Certi oggetti possono rispondere a certe richieste (e sono i R)
- Quando CC deve chiedere una cosa, gli si fa conoscere R che lega le due classi

- *Client (Cl)* inizializza un certo attributo *CC* per far usare la giusta istanza di *R*

## Collaborazioni

- In *CC* è possibile **aggiungere funzionalità** (*tipo*: ripristino di stati precedenti. Per es: eliminare l'ultima operazione fatta su un *R*. Cioè una sorta di `unexecute()`)
- *CC* potrebbe anche avere una **coda di richieste** o dei **log di richieste** (*quindi una cronologia di azioni*)

## Conseguenze

- La parte dell'applicazione che vuole mandare una richiesta è **DISACCOPPIATA** da chi riceverà la richiesta e, inoltre, non conosce tutte le operazioni che effettivamente vengono fatte
- I dettagli della richiesta sono conosciute dal *ConcreteCommand*
- Gli oggetti della gerarchia di *C* possono essere estesi come una normale gerarchia di classi
- I *CC* possono essere anche **composti** (vedi *Composite*) e quindi creando *MacroCommand* che rappresenta una lista di comandi già predisposti
- E' possibile **parametrizzare** i *CC*. Man mano che si cambia quello che c'è all'interno di *CC*, si può cambiare il comportamento totale dell'applicazione. Per questo motivo non è sempre necessario scrivere nuove classi ma basterà **cambiare i parametri**

## Implementazione

Si deve scegliere cosa mettere in *CC*:

- potrebbe essere solo un semplice ***TRAMITE***, quindi semplice *passaggio di chiamate*. Serve solo per disaccoppiare *Invoker* dal *Receiver*
- potrebbe contenere molti altri controlli e/o metodi per far lavorare *R* in base alla situazione

CC potrebbe sostituirsi interamente a *R* ma è una *scelta un po' estrema* che degenera e CC ingloba *R*

- Se si volessero usare azioni di ***UNDO*** e/o ***REDO*** serve conservare una lista di azioni eseguite **Annullamento e Rifacimento**

## Esempio

# Stream pt.2

## Metodo filter()

Prende in un'espressione lambda e ritorna un booleano: se è true, allora l'elemento viene messo nello stream di output che contiene gli elementi valutati dall'espressione lambda.

- Filter non è terminale e restituisce uno Stream
- il metodo `count()` serve per contare gli elementi in un determinato Stream

```
List<String> nomi = "Pippo", "Pappa", "Poppo";

long c = nomi.stream().filter(s -> s.length == 5).count(); // 3
```

## Tipo Predicate<>

Predicate è un tipoc che contiene un'espressione lambda che poi viene passata ad una Filter.

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

Stream<Integer> result = Stream.of(2, -1, -5, 34, 3).filter(positive); // result = [2,34,3]
```

Ne segue che Predicate e Stream devono corrispondere ai tipi **NON PRIMITIVI** (`Integer`, `Long`, `Double`, ... ) passati in input alla funzione `filter()`.

## Metodo reduce()

```
reduce(T identity, BinaryOperator<T> accumulator); // T è uguale al tipo di oggetti dello Stream

reduce(0, (accum,v) -> accum + v); // parte da 0 e vengono ad accum si sommano gli elementi "v" dello Stream
```

Alla prima iterazione:

- `accum` = 0 perchè è il valore di partenza specificato
- `v` corrisponde primo valore dello stream
- L'espressione lambda in `reduce()` prende *due parametri in input* e ne *restituisce un solo valore* (`accum*`).
- `reduce()` è un'operazione terminale e si usa quando si vuole passare da un insieme di valori ad un singolo valore.

## Riferimenti a metodi

Ci si può riferire a metodi dall'interno di una classe.

Il codice si troverà all'interno dentro un metodo specifico di una determinata classe. Il metodo non viene chiamato direttamente ma attraverso il richiamo della classe.

```

reduce(0, Integer::sum); // -> restituisce T
reduce(Integer::sum) // -> restituisce Optional<T>. Invocare isPresent()

```

All'interno di Integer ci sarà il metodo `sum()` e rappresenta un'espressione lambda visto che `reduce()` accetta solo espressioni lambda.

- Ovviamente il risultato non esisterà se lo Stream sarà vuoto. In questo caso, la `reduce` con 2 parametri restituisce 0.
- Se, invece, la `reduce` ha un solo parametro, quindi `reduce(Integer::sum)`, allora restituisce un tipo `Optional<T>` che è un contenitore che **può avere/non avere** un risultato.
  - *Se lo Stream è vuoto allora esso sarà vuoto*
  - Su `Optional<T>` si può invocare `isPresent()` o `isEmpty()` per vedere se il contenitore è pieno o no, oppure se la `reduce` ha restituito un risultato oppure no.
- **Bisogna accertarsi, prima di andare avanti, che la `reduce()` abbia restituito un valore per evitare errori**
- `reduce()` può prendere in **ingresso solo l'espressione lambda**, quindi senza il parametro di inizializzazione della variabile. In questo caso, il metodo prende come parametri iniziali (*run-time*) il primo elemento e il secondo dello Stream

```

// Esempio uso reduce()
public class Pagamenti {
 private List<Float> importi = new ArrayList<>();

 // in stile imperativo
 public float calcolaSommaImper() {
 float risultato = 0f;
 for (float v : importi)
 risultato += v;
 return risultato;
 }

 // in stile funzionale
 public float calcolaSomma() {
 return importi.stream().reduce(0f, Float::sum);
 }
}

```

## Metodo map()

Si applica ad uno Stream e prende in input un'espressione lambda. Produce uno Stream in uscita e li **TRASFORMA** lo Stream.

- Per trasformazione si intende: cambiare il tipo degli elementi di partenza, cambiarne i valori raddoppiandoli ecc...
- Il tipo in uscita di `map()` può, quindi, essere diverso dal tipo di elementi nello Stream iniziale

```
map(Function<T,R> mapper); // T è il tipo in input. R è il tipo in output
```

```

// Esempio
List<Integer> l = List.of(1,2,5);
Stream<Integer> s1 = l.stream().map (x -> x * 2); // risultato = [2,4,10]
List<Integer> s2 = s1.toList(); // converte in List uno Stream

```

```

//-----
List<Persona> l = List.of(new Persona("Pippo", 46), new Persona("Alessio", 18));
Stream<Integer> result = l.stream().map(Persona::getEta); // result = [46,18]
// l.stream().map(p -> p.getEta()); // chiamata equivalente

map(Persona::getEta); // Per ogni elemento dello Stream si chiama il metodo getEta presente
in Persona.

```

- Calcoliamo la somma delle età delle istanze di Persona

```

public class Persona {
 private String nome;
 private int eta;
 public Persona(String n, int e) {
 nome = n;
 eta = e;
 }
 public String getName() {
 return nome;
 }
 public int getEta() {
 return eta;
 }
}

List<Persona> amici = Arrays.asList(
 new Persona("Saro", 24), new Persona("Taro", 21),
 new Persona("Ian", 19), new Persona("Al", 16));

// somma calcolata in stile funzionale con i riferimenti ai metodi
int somma = amici.stream()
 .map(Persona::getEta)
 .reduce(0, Integer::sum);

```

- La funzione passata a map() è il metodo getEta() di Persona

```

// in stile imperativo
int somma = 0;
for (Persona x : amici)
 somma += x.getEta();

// alternativa in stile funzionale
int somma =
 amici.stream()
 .map(ps -> ps.getEta())
 .reduce(0, (s, e) -> s + e);

```

## Stile dichiarativo vs funzionale

```

// Data una lista contenente valori *String*:

List<String> nomi = Arrays.asList("Saro", "Taro", "Ian", "Al");

// Per determinare se la lista contiene un certo valore, in stile dichiarativo:

if (nomi.contains("Saro")) System.out.println("Saro trovato");

// Data una lista contenente istanze di Persona:
List<Persona> amici = Arrays.asList(new Persona("Saro", 24), new Persona("Taro", 21), new
Personna("Ian", 19), new Personna("Al", 21));
// Lo stile funzionale consente di estrarre il campo nome, e inoltre permette di valutare una
funzione ad-hoc, quindi è molto più flessibile e potente

long c2 = amici.stream().filter(s -> s.getName().equals("Taro")).filter(s -> s.getEta() ==
21).count();

```

- `count()` conta quanti elementi ha lo stream prodotto da filter
  - Data la lista di istanze di `Persona`, trovare il nome della persona che è più grande (di età) fra quelli che hanno meno di 20 anni

```
List<Persona> amici = Arrays.asList(new Persona("Saro", 24),
```

```
 new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```

- In versione imperativa, se volessimo scorrere la lista solo una volta

```
Persona pmax = null;
for (Persona ps : amici) {
 if (ps.getEta() < 20) {
 if (pmax == null) pmax = ps;
 if (pmax.getEta() < ps.getEta()) pmax = ps;
 }
}
if (pmax != null) System.out.println("persona: " + pmax.getNome());
```

- Il corpo del ciclo ha varie condizioni, queste rendono il codice più difficile da comprendere

## Ricerca funzionale, v1.0

- Aggiungendo su `Persona` il metodo `getMax()`

```
/** restituisce l'istanza con il valore massimo di eta' */
public static Persona getMax(Persona p1, Persona p2) {
 if (p1.getEta() > p2.getEta())
 return p1;
 return p2;
}
```

- Possiamo implementare la ricerca in modo funzionale

```
Optional<Persona> pmax = amici.stream()
 .filter(x -> x.getEta() < 20)
 .reduce(Persona::getMax);

if (pmax.isPresent())
 System.out.println("persona: " + pmax.get().getNome());
```

- `filter()` è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- `reduce()` è usata per selezionare un elemento: invoca `getMax()` che confronta a due a due

## Ricerca funzionale, v2.0

```
Optional<Persona> pmax = amici.stream()
 .filter(x -> x.getEta() < 20)
 .max(Comparator.comparing(Persona::getEta));
if (pmax.isPresent())
 System.out.println("persona: " + pmax.get().getNome());
```

- `filter()` è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento
- `max()` trova il valore massimo, è un'operazione terminale, prende un `Comparator`, restituisce un `Optional`, `max()` opera in modo simile a `reduce()`
- `Comparator.comparing()` prende una funzione che estrae una chiave e restituisce un `Comparator`
- `Comparator` implementa una funzione di confronto (`compare()`) che controlla l'ordinamento di una collezione di oggetti
- `max()` al suo interno chiama il metodo `compare()` del `Comparator` in input

- `Comparator.comparing(Persona::getEta)`: il parametro per fare il confronto in `comparing()` è l'età della `Persona`.

- `comparing()`, al suo interno, chiama un altro metodo implementato, chiamato `compare()` e fa un confronto fra 2 parametri

## Tipo Comparator

- `Comparator<T>` è una interfaccia funzionale, una sua implementazione permette di stabilire un ordine su una collezione di oggetti. Il metodo che definisce è `compare(T o1, T o2)`
- Se `o1 == o2` restituisce 0
- Se `o1 > o2` restituisce un valore positivo
- Se `o1 < o2` restituisce un valore negativo
- Quindi posso implementare il Comparator nel seguente modo

```
Comparator<Persona> myComp = (p1, p2) -> p1.getEta() - p2.getEta();
Optional<Persona> pmax = amici.stream()
 .filter(x -> x.getEta() < 20)
 .max(myComp);

• Ovvero
 .max((p1, p2) -> p1.getEta() - p2.getEta());
```

Visto che mi occorre fare il confronto fra età, allora implemento un Comparator apposito.

Estrarre il massimo (come in questi esempi). Bisogna ricordare almeno un metodo per farlo (**ESAME**)

## Metodo collect()

- ```
List<Persona> amici = List.of(new Persona("Saro", 24),
    new Persona("Taro", 21), new Persona("Ian", 19), new Persona("Al", 16));
```
- Ricaviamo la lista delle età


```
List<Integer> e = amici.stream()
        .map(x -> x.getEta())
        .collect(Collectors.toList());
```
 - Come prima, `map()` restituisce uno stream con i valori delle età, e la funzione passata dice come trasformare ciascun elemento dello stream
 - `collect()` permette di raggruppare i risultati e prende in ingresso un `Collector`
 - La classe `Collectors` implementa metodi utili per raggruppamenti, il metodo `toList()` restituisce un `Collector` che accumula elementi in una `List`
 - Java 16 dà il metodo `Stream.toList()`

```
// in stile imperativo
List<Persona> p; // come sopra
List<Integer> e = new ArrayList<>();
for (Persona x : p)
    e.add(x.getEta());
```

 - `.toList()` trasforma in `List` e basta
 - `.collect()` permette di trasformare in `List` o qualsiasi altra cosa, quindi risulta più **FLESSIBILE**

Programmazione Parallela

Quando si opera con gli Stream, è più utile andare in parallelo

Le istruzioni viste in precedenza si applicano sequenzialmente nel tempo agli elementi iniziali ma, se si vuole usare il parallelismo, si può fare:

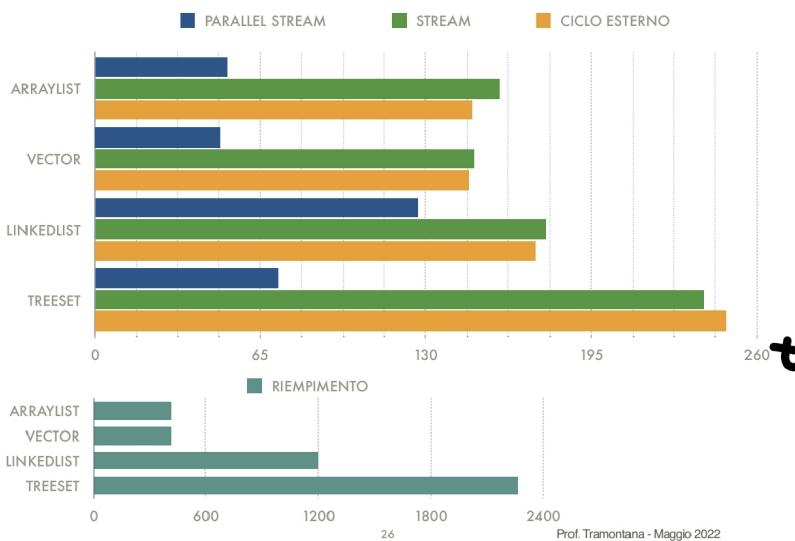
- piuttosto che invocare `Stream` si usa `parallelStream()` e dà uno Stream in parallelo
- Esiste un altro metodo, `parallel()` invocato sullo Stream

Le operazioni fatte in parallelo, dal punto di vista della correttezza, funzionano bene in tutti i casi se il programmatore sta attento a quello che fa.

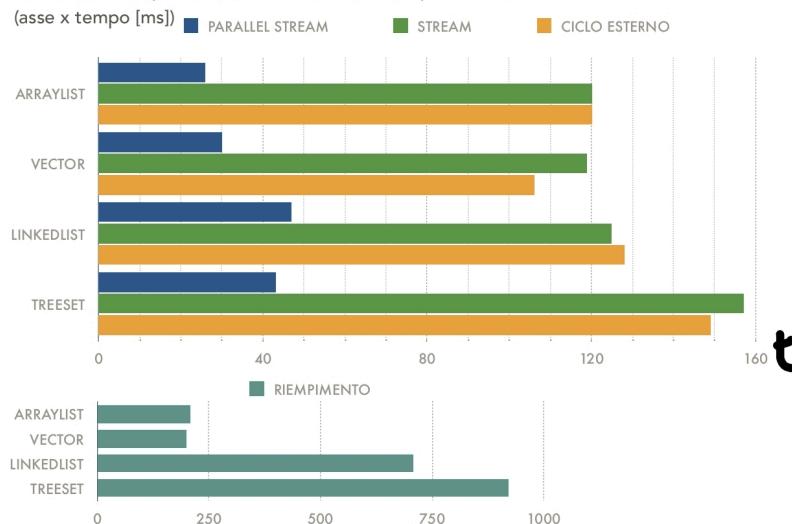
- In generale, si produce sempre uno Stream (output) differente dallo Stream in input
- Tutto quello che avviene nello Stream iniziale può avvenire separatamente rispetto a quello che si produce in output e ciò **garantisce la CORRETTEZZA** visto che l'input non viene modificato in alcun modo (in particolar modo quando uso il parallelismo)
- I programmatore potrebbero sbagliare nello scrivere il codice:
 - l'esecuzione in parallelo non prevede di avere uno **stato globale** e quindi non si aggiorna uno stato globale e in questo caso il parallelismo va benissimo
 - In caso contrario, **quindi si modifica uno stato globale**, ci si deve chiedere se è *necessario farlo*. Se serve farlo, bisogna *valutare e controllare di farlo correttamente*, controllando anche l'*ordine delle operazioni* (*diventa complicato*).
 - Per **STATO GLOBALE** si intende una qualsiasi **variabile/classe condivisa** con le altre parti di codice

La programmazione parallela consente di avere un notevole guadagno nelle prestazioni (*ricerca di elementi*):

Hardware 4 core, Ricerca su 5 Milioni di elementi, Java 11.0.2



Hardware 8 core, Ricerca su 5 Milioni di elementi, Java 18.0.1



- La barra (*blu*) più "corta" indica tempi migliori con programmazione parallela

Considerazioni

- Le prestazioni migliorano in caso di parallelismo (basta inserire `.parallel()`)
- Se uso un `LinkedList` ci perdo rispetto all'uso di un `ArrayList`
- Se si usano gli Stream di default si possono avere dei guadagni in termini di tempo e prestazioni man mano che l'hardware si aggiorna
- Il metodo `parallelStream()` **POSSIBILMENTE** da uno Stream parallelo perchè si deve valutare se il parallelismo è supportato dall'hardware prima di avviare tale procedura. Viene valutata anche la dimensione lo Stream e la sua dimensione: se è molto poca allora viene scindita in maniera sequenziale perchè ci vuole meno tempo rispetto all'avvio del parallelismo stesso

Esempio 1

- Data una lista di istanze di `Persona` trovare i nomi delle persone che sono giovani ed hanno ruolo `Programmer`, e ordinare i risultati

```
List<Persona> team = List.of(new Persona("Kent", 29, "CTO"), new Persona("Luigi", 25, "Programmer"), new Persona("Andrea", 26, "GrLeader"), new Persona("Sofia", 26, "Programmer"));

team.stream()
```

```
.filter(p -> p.giovane())
.filter(p -> p.isRuolo("Programmer"))
.sorted(Comparator.comparing(Persona::getNome))
.forEach(p -> System.out.print(p.getNome() + " "));
```

// Output: Luigi Sofia

- `filter()` operazione intermedia che restituisce gli elementi che soddisfano il predicato passato
- `sorted()` operazione intermedia stateful che restituisce uno stream che ha gli elementi ordinati in base al Comparator passato
- `comparing()` permette di estrarre la chiave per il confronto
- `forEach()` operazione terminale che esegue un'azione su ciascun elemento dello stream (su uno stream parallelo l'ordine non è garantito) Prof. Tramontana - Maggio 2019

- `.sorted()` ordina i risultati secondo una caratteristica passata in input tramite `Comparator.comparing(Persona::getNome)` e quindi si ordina secondo il nome della persona
 - è un'**operazione intermedia**, detta **STATEFUL**, cioè hanno visione di tutti gli elementi dello Stream per lavorare
 - Generalizzando, i metodi `min()`, `max()` e `sorted()` vogliono in input un `Comparator` dello stesso formato
- `.forEach()` è **terminale** e permette di **eseguire un'operazione per ogni elemento** dello Stream ed **ELIMINA LO STREAM**. Quindi non ha un valore di ritorno.
 - In questo caso, lo **Stream viene eliminato** e viene stampato solo l'informazione richiesta con `System.out.println()`
 - (**Attenzione!**) Segue che, **DOPO** `.forEach()`, **NON** si possono invocare altri metodi di Stream visto che esso viene **ELIMINATO**. Tutte le operazioni che si vogliono invocare devono essere chiamate prima di invocare tale metodo, appunto
- E' sempre meglio fare 2 filter piuttosto che due condizioni in AND.
- I `filter()` **concatenati** sono **AND LOGICI** e quindi vengono selezionati gli elementi dello Stream che **soddisfano ENTRAMBI I FILTER** espressi.

Esempio 2

- Data una lista di istanze di Persona trovare i diversi ruoli

```
team.stream()
    .map(p -> p.getRuolo())
    .distinct()
    .forEach(s -> System.out.print(s + " "));

// Output: CTO Programmer GrLeader
```

- `distinct()` operazione intermedia stateful che restituisce uno stream di elementi distinti

- Data una lista di istanze di 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());

// Output: Luigi
```

• `findAny()` (simile a `findFirst()`) operazione terminale che restituisce un `Optional`, per valutarla non è necessario esaminare tutto lo stream, si dice short-circuiting (può far sì che alcune parti non eseguano)

Prof. Tramontana - Maggio 2019

- `distinct()` permette di avere dei risultati ***NON DUPLICATI*** all'interno di uno Stream
 - è un'***operazione intermedia*** ed è ***stateful***

Si può usare `parallel()` per l'operazione `sorted()` e ci penseranno le varie operazioni e le librerie a parallelizzare

Stateless vs Stateful

- Le operazioni `map()` e `filter()` sono stateless, ovvero non hanno uno stato interno, prendono un elemento dello stream e danno zero o un risultato
- Le operazioni come `reduce()`, `max()` accumulano un risultato. Quest'ultimo ha una dimensione limitata, indipendente da quanti elementi vi sono nello stream. Il risultato in una passata viene dato in ingresso alla passata successiva
- Le operazioni come `sorted()` e `distinct()` devono conoscere gli altri elementi dello stream per poter eseguire, si dicono stateful

Generare Stream: `iterate()`

- `iterate()` produce uno ***Stream infinito***
- Per questo motivo si chiama `limit()` che dice quante volte deve essere chiamata `iterate()`, e quindi il numero di elementi dello Stream
- `iterate(elemento_iniziale, espressione_lambda)` è la firma della funzione e lo Stream è riempito con gli elementi nel modo seguente:
 - valore iniziale seguito dal risultato dell'espressione lambda applicata all'elemento stesso e, chiaramente, deve essere compatibile con il tipo degli elementi
- Si conosce già il seme e il risultato dell'espressione lambda viene ***applicata al risultato dell'applicazione precedente***

```
Stream.iterate(2, n -> n * 2)
    .limit(10)
    .forEach(System.out::println);
```

Generare Stream: `generate()`

- Non prende parametri in ingresso ma applica la funzione in ingresso infinite volte (va aggiunto anche `limit()`)
- Il valore viene fornito in maniera indipendente rispetto agli altri valori dello Stream
 - Il metodo `generate()` permette di produrre uno stream infinito di valori, tramite una funzione di tipo `Supplier`, ovvero che fornisce un valore
 - `generate()` non applica una funzione ad ogni nuovo valore prodotto, come invece fa `iterate()`

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

- Il codice sopra genera uno stream di 5 numeri casuali, ciascuno fra 0 e 10
- `limit()` si può usare anche con `filter()` per indicare di selezionare solo un determinato numero massimo di elementi che "passano" il controllo
 - E' obbligatoria per `generate()` e `iterate()` e facoltativa per il resto

Tipo IntStream

Rappresenta uno Stream di valore Intero.

- `Stream<Integer>` e `IntStream` **NON** sono compatibili
- `IntStream` può avere **SOLO** valori interi
- *Mette a disposizione:*
 - `rangeClosed(1,6)` che produce un `IntStream` che comprende valori interi compresi fra gli estremi indicati (**inclusi**)
 - `.sum()` somma gli elementi di uno `IntStream` e restituisce, appunto, un valore intero.
 - Questa funzione **NON E' DISPONIBILE** per gli Stream normali

Conversioni Stream -> IntStream

```
int result =
    Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
        .mapToInt(x -> x.length()).sum();
// Output: result = 31
```

• `mapToInt()` esegue la funzione passata e restituisce un `IntStream`

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

• `boxed()` restituisce uno `Stream` di `Integer` a partire da un `IntStream`

• Si abbia la lista che contiene istanze di `Persona`, si generi uno stream contenente i primi 4 elementi

```
List<Persona> lista;
```

```
Stream<Persona> p =
    IntStream.rangeClosed(0, 3)
        .mapToObj(i -> lista.get(i));
```

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

- `mapToInt()` è un metodo di `Stream` e produce un `IntStream`
- Seguendo la serie di invocazioni, `sum()` è invocato su un tipo `IntStream`
- Da un `IntStream` posso generare uno `Stream<Integer>` invocando `.boxed()`
- `mapToObj()` è un metodo `IntStream` e produce Oggetti da mettere all'interno di uno Stream

Progettare per preservare (riepilogo e usi dei Design Pattern)

Slides che non erano caricate

I design pattern servono per ragionare bene per la progettazione di un sistema software.

Per orientarsi su quale pattern vedere e utilizzare, prima di valutarli singolarmente, si possono considerare le seguenti possibili esigenze:

- **Non ci si vuole legare alle singole istanze** ma ad un'interfaccia più generica. Serve più flessibilità quando si usa una classe. Allora si usa **Factory Method** e **Prototype**
- Si vuole usare il software per il futuro e deve essere pronto per prossimi aggiornamenti- Non si sa chi è il destinatario della nostra chiamata. quindi non si legge il richiedente con un particolare ricevente. Quando si modifica il software, si ha l'esigenza di adeguare il software ad ogni tipo di ricevente. Allora si usa **Chain of Responsibility** e **Command**
- Il software, in genere, dura molti anni. In tutto questo tempo cambiano le caratteristiche della macchina sulla quale questo software "gira" (cambiano le librerie, **tutto si evolve**). Non bisogna legarsi fortemente alle librerie e alle piattaforme hardware e quindi per permettere future modifiche. Se nel software si capisce che, nel corso del tempo, sarà utile utilizzare nuove librerie, allora vuol dire che bisogna fare adattamenti. Si possono usare **Factory Method (ma anche Abstract Factory)**, **Bridge** permettono il cambiamento di librerie e adattamenti alle nuove tecnologie
- Si ha una **dipendenza di implementazione**, quindi c'è un algoritmo che sa prendere i dati dalle varie parti e li sa elaborare. Se si pensa che ci saranno varianti di questo algoritmo allora non bisogna legarsi troppo ad un'implementazione. Si può usare il **Factory Method** cambiando all'interno dei *ConcreteProduct* le varie implementazioni. Oppure **Bridge**, **Memento**, **Proxy**.
- Una classe, in un determinato momento, sa quello che deve fare (**in base al suo stato**). L'algoritmo con cui operare è diverso in vari momenti del software in base al suo stato. In questo caso si può usare **State**. Simile a State esiste **Strategy** e, in particolare:
 - E' come uno *State*. I ruoli (nomi) sono molto simili allo *State* e le relazioni fra le classi sono identiche. Questi due design pattern sono identici e i ruoli sono mappati allo stesso modo.
 - **Per State**: si ha un sistema che attraversa vari stati.
 - **Per Strategy**: si ha un sistema che vuole applicare diversi algoritmi, a seconda di un'impostazione che c'è in un determinato momento. Questi diversi algoritmi li implemento sui *ConcreteStrategy* (analogamente ai *ConcreteState* di *State*)
 - Si può anche **usare Template Method** (che usa *dispatch*). Si diminuiscono le dipendenze con l'algoritmo
 - Altri Design Pattern sono: Iterator (scorre in maniera flessibile liste di dati), Visitor (Non serve spesso ma solo quando si vuole eseguire una navigazione del codice già scritto)
- Nel progetto si ha uno **stretto accoppiamento** che incrementa la probabilità che una classe sia riusata da sola e che il sistema sia compreso, modificato ed esteso più facilmente. Si incapsulano vari algoritmi in un macro sistema in sottoclassi. **Si usa il Façade per disaccoppiare le singole classi dai chiamanti**. Alternativamente si possono usare **Factory**, **Bridge**, **Chain**, **Command**, **Mediator**, **Observer** che eseguono questo **disaccoppiamento fondamentale**.
- Sebbene l'ereditarietà permette di riusare codice già scritto (ed estenderlo), non si deve abusare. Se nel sistema si hanno gerarchie molto profonde (+10 livelli classi), allora i livelli più

profondi sono troppo complessi perché contengono tutti i metodi delle classi che stanno al livello superiore. Testare questo sistema vuol dire valutare tutti i possibili classi. Si può riusare il codice tramite **composizione**: all'interno di una classe si crea un'**istanza di un'altra classe e determinate chiamate verranno rimandate a tale istanza**. L'interfaccia sulla quale si chiamano i **metodi rimandati, è molto più specifica, ristretta e limitata**.

- E' necessario **aggiungere funzionalità senza essere abilitati a modificare** un determinato codice (quindi **non si hanno i permessi sul codice**). Si può usare **Adapter (cambia l'interfaccia)** e **Decorator (non cambia l'interfaccia)**

Metriche su evoluzione dei software

Dopo il completamento del prodotto viene fornito al cliente in modo definitivo. In questo caso vuol dire che è stato testato ed è pronto per la consegna. La consegna definisce il **completamento del processo di sviluppo**

- Dopo questo momento intervengono ulteriori notifiche per vari motivi: per esempio -> i programmatore vogliono fare delle correzioni prima che il cliente lo dica

Tutte le modifiche che seguono la consegna del software fanno parte della procedura di **MANUTENZIONE**.

***Manutenzione** ed **evoluzione** sono spesso equivalenti. Se si vuole essere precisi: **manutenzione** (modifica del software) ed **evoluzione**(nuovo software, rilascio di patch, rimozione sistema)*

Mediamente i costi di manutenzione sono abbastanza alti perchè potrebbe potenzialmente durare molto di più rispetto allo sviluppo. Ci sono delle **categorie per cambiamenti di evoluzione** del software:

- **Correttivo** (*statisticamente 17% fra tutti i cambiamenti*) = **Rimozione di difetti**. Vengono rimossi il prima possibile per evitare problemi futuri
- **Adattivi** (*18% fra tutti i cambiamenti*) = **cambiamento dovuto per farlo girare su un'altra piattaforma software**. Può essere eseguito su un nuovo SO, usa una nuova libreria ecc...
- **Perfettivi** (*60% fra tutti i cambiamenti*) = Ogni volta che cambio il software per **aggiungere funzionalità migliorandole** oppure **miglioro le prestazioni**. Se il software è utile al cliente, egli vorrà farci sempre più cose e, quindi, richiede sempre nuove funzionalità. Quindi il software ha *vita lunga*
- **Preventivi** (*5% fra tutti i cambiamenti*) = modifica internamente la struttura (refactoring) senza incidere sulle funzionalità e correttezza di esecuzione. Prima della consegna alcune cose potevano essere progettate meglio (*magari serviva qualche design pattern non utilizzato*). Queste modifiche **ci si prepara per poter intervenire in maniera più semplice** in futuro sul software.

E' buona pratica anticipare i cambiamenti a *design time* (tramite parametrizzazione, encapsulamento, etc.)

Dinamiche di evoluzione

Lehman e Belady studiavano i comportanti dei sistemi software:

"I sistemi materiali sono sottoposti a leggi fisiche ma i sistemi software no. Ci sono delle leggi che possono regolare i sistemi software e che fanno vedere le leggi soddisfatte da molti sistemi software? "

Vi sono delle **leggi** (*che sono state dimostrate*) che si sono rilevate veritieri per i sistemi software. Queste leggi riguardano sistemi software di **GRANDI DIMENSIONI** sviluppati da **grandi aziende**.

Leggi di Lehman

- **Cambiamento continuo:** I sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti
 - Se un software è ritenuto di successo, allora deve continuamente cambiare perché altrimenti sarà sempre meno soddisfacente.
- **Aumento della complessità:** Quando un sistema evolve, la sua struttura aumenta di complessità, a meno che del lavoro viene fatto per preservare o semplificare la sua struttura
 - Se **NON si riflette profondamente sulla struttura del sistema software**, si elimina la struttura iniziale del sistema, quindi **si degrada**
- **Auto-regolazione:** Attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti
 - Se si misura la dimensione del software e il tempo necessario fra una release importante e un'altra anch'essa importante, si vede che gli errori fra le release sono circa uguali. Nonostante si facciano delle modifiche, certi parametri rimangono costanti nel tempo. La dimensione rimane circa uguale perché alcune parti vengono rimosse e, spesso, le aggiunte non sono di grandissime dimensione.
 - Gli errori sono sempre circa costanti fra le release. Le scelte sul design e progettazione sono quelle fatte all'inizio. La capacità di un programmatore di scrivere codice e progettare rimane sempre basata sui requisiti iniziali e per questo gli errori possibili saranno pressoché uguali
- **Stabilità organizzativa:** Durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo
 - si legga alla terza legge ma si focalizza sul tasso di sviluppo: "Quante modifiche si possono fare per fornire una nuova release?". Se si ha una nuova versione e si devono aggiungere delle funzionalità, esse saranno limitate nel range di funzionalità da poter sviluppare per release. Occorre un certo tempo per studiare un certo software indipendentemente dal numero di persone assegnata al team di sviluppo

Più persone si aggiungono al team, più si rallenta lo sviluppo (si devono coordinare, si devono fare domande per le scelte fatte, devono studiare per un bel po' di tempo il software GIA' SVILUPPATO).

Applicabilità delle leggi

- Sono in generale applicabili a grandi sistemi sviluppati da grandi organizzazioni
- Non è chiaro come si adattano a
 - Piccoli prodotti
 - Prodotti che incorporano un certo numero di COTS
 - Piccole organizzazioni

Costo di manutenzione

Occorre un **grande tempo per lo studio del software** prima di eseguire modifiche.

- Spesso gli sviluppatori della prima versione non sono mai gli stessi di una successiva versione. Man mano gli sviluppatori si allontanano dalla scrittura del codice perché, man mano che si diventa più bravi, si sale di livello e si diventa sempre più importanti fino a diventare coordinatori e progettisti.
- Il software **si degrada nel tempo** e quindi **apportare modifiche diventa sempre più dispendioso** di tempo e quindi richiede **più soldi**

- Gli sviluppatori che hanno scritto il software inizialmente erano spinti a produrre in poco tempo la prima release. Ne segue che chi scrive il codice, non è incentivato a scrivere codice che DURI NEL TEMPO proprio perché si vuole consegnare il prima possibile. Questa mancanza, si ripercuote su una cattiva versione del codice e quindi su un maggior costo futuro di manutenzione

Modelli di manutenzione

Ci sono 3 modi:

- **Quick-fix:** Si ha poco tempo per aggiustare. La visione è relativa al codice scritto senza avere visione della documentazione o dei test quindi non esiste un'analisi del sistema
- **Miglioramento iterativo:** Cambiamenti fatti in base ad un'analisi del sistema esistente. Si fa un controllo sulla complessità e sul mantenimento del design.
- **Riuso:** diventa impossibile modificare il software e quindi si riusano delle componenti. Si prendono parti dalla versione precedente. Stabilire i requisiti del nuovo sistema riusando il più possibile

Tipi di modifiche

- **Re-factoring o re-structuring:** Processo di cambiamento del software che non altera il comportamento del codice ma migliora la struttura interna. Ovvero: prendere un sistema fatto male e modificarlo per ottenere una struttura ben fatta
- **Reverse engineering:** Analizzare un sistema per estrarre informazioni sul suo comportamento o sulla sua struttura
 - Capire dal codice disponibile cosa è stato fatto nel sorgente
- **Re-engineering:** Alterare un sistema per ricostituirlo in un'altra forma. Cambiare totalmente il design e le varie interazioni fra le parti del sistema

Metriche

Si possono fare delle misurazioni sul software in sviluppo o già sviluppato per analizzare se si sta procedendo nella maniera corretta o no.

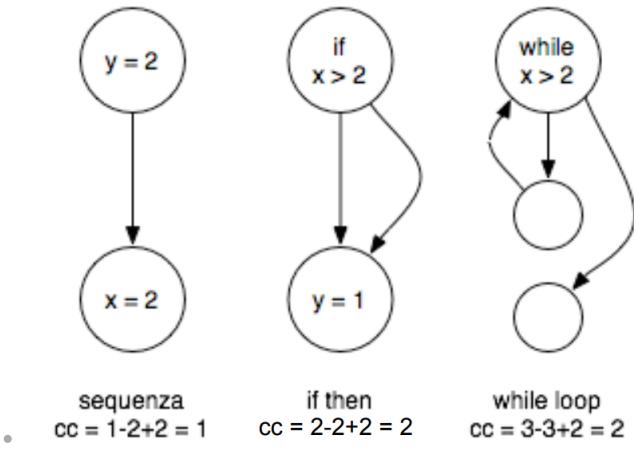
La metrica è solo un'indicazione con determinati limiti e a volte è utile avere uno strumento tale a disposizione.

Si distinguono:

- **METRICHE TRADIZIONALI**-> nate prima del sistema ad oggetti (quindi specificate per tale campo) ma si applicano anche per sistemi non a oggetti

Metriche tradizionali

- **LOC (lines of code)** -> indica il numero di linee scritte nel codice sorgente. Si può misurare aprendo il codice e contando le linee. Si contano le linee vuote, le parentesi isolate su una singola riga ecc....Si conta tutto.
- **NCNB (Non comment - non blank)** -> si conta tutto tranne commenti o linee vuote
- Si potrebbero anche contare i punti e virgola (;) che separano le istruzioni
- **Complessità ciclomatica** (cc): si applica ad un algoritmo (non ad un oggetto). Si può rappresentare l'algoritmo tramite un grafo dove:
 - linea di codice/istruzione = nodo del grafo
 - flusso fra le linee = arco del grafo



- Per calcolare cc si contano nodi/archi del grafo
- $cc = \text{archi} - \text{nodi} + 2$
- **Per istruzioni sequenziali** -> 2 nodi e 1 arco -> cc = **numero di test** = 1
- **Per istruzioni condizionali** -> 2 archi e 2 nodi -> cc = **numero di test** = 2
- **Per istruzioni iterative** -> 3 archi e 3 nodi -> cc = **numero di test** = 2
- In generale, **più è alto cc, più condizioni nidificate ci sono e quindi più è complesso il codice (anche nella sua comprensione)**
- Se si vuole percorrere almeno una volta ciascun ramo del grafo, bisogna leggere proprio cc e questa "variabile" viene detta "*numero di test*" e **serve per capire quanti test bisogna fare per testare un determinato codice**

Metriche CK (Chidamber e Kemerer)

Sono metriche per sistemi ad oggetti. I numeri che ci danno queste metriche sono testate a fondo e quindi sono molto affidabili. **Ogni classe avrà un singolo valore fra le misure che seguono.**

| **VALORI ALTI**=maggiora complessità -> **TRANNE NOC**=minor complessità

Ci sono 6 modi per fare le misure:

- **WMC** (*Weighted Methods per Class*): per ogni classe si può ricavare la somma pesata dei metodi delle classi. Si sceglie un peso adatto per la misura e si vede quanto quella classe "pesa" e quanto è importante all'interno del sistema. **Non vengono considerate le superclassi**. Un peso si attribuisce:
 - **peso = costante** al metodo: WMC sarà, quindi, il numero di metodi della classe
 - **peso = LOC** del metodo -> i metodi più lunghi sono quelli più complessi
 - **peso = complessità ciclomatica** per ogni metodo
 - **NOTE WMC**: se è alto, è **dificile riusarla** e quindi molto più **complessa da comprendere**.
- **DIT** (*Depth of Inheritance Tree*): profondità (**numeri di livelli**) albero della gerarchia. Alla radice della gerarchia si dà un valore di default (anche 0). Più è profonda la gerarchia, più è complessa quella classe.
 - **NOTE DIT** : se il valore è alto vuol dire che la complessità è alta e i test si fanno troppo in profondità **Conta il numero di classi che ereditano da una classe specifica**.
- **NOC** (*Number of Children of a Class*): conta il **numero di figli che si trovano immediatamente sotto** per una singola classe.
 - **NOTE NOC**: Possibile riuso se NOC è alto
- **CBO** (*Coupling Between Object Classes*): dà l'accoppiamento fra le classi. Si devono contare le associazioni fra le classi (quindi le interazioni) e non l'ereditarietà.

- Es: A → B A → C B → D: $CBO(A) = 2$ e $CBO(B) = 1$
- **NOTE CBO:** Più è alto e più il sistema interagisce con altre classi. Quindi è molto più complesso
- **RFC (Response for a Class)**: quante **interazioni vengono fatte quando una classe riceve un'invocazione**. Quanti metodi diversi sono chiamati all'interno di un unico metodo. A tale numero viene sommato il numero totale di metodi. In altre parole: quanti messaggi distinti può inoltrare una classe (RFC). Più chiamate fa una classe, più è accoppiata con altre classi.
- **NOTE RFC:** Più è alto, allora ci sono molte interazioni. Implica più test e più complessità
- **LCOM (Lack of Cohesion of Methods):** mancanza di coesione fra i metodi di una classe. Coesione = singola responsabilità per i metodi -> ogni metodo svolge un unico piccolo compito. **Idealmente si deve avere: alta coesione e basso accoppiamento fra le classi.**

$$LCOM = 1 - \frac{\sum_i^a m_{A_i}}{m * a}$$

- a=numero di campi, m=numero di metodi, m_{A_i} =numero di metodi che usano il campo A_i
- **Per CAMPO si intende una qualsiasi variabile/attributo.** Il metodo 1 usa la variabile? E il metodo 2? ecc...
- Esempio: *con 3 campi e 3 metodi:*
 - *ciascun metodo usa 3 campi*, allora $(3 + 3 + 3)/3 * 3 = 1$
 - *ciascun metodo usa 2 campi*, allora $(2 + 2 + 2)/3 * 3 = 2/3$
 - *ciascun metodo usa un campo*, allora $(1 + 1 + 1)/3 * 3 = 1/3$
 - *un solo metodo usa 3 campi*, allora $(1 + 1 + 1)/3 * 3 = 1/3$
- **NOTE LCOM:** LCOM è basso -> coesione alta -> buona progettazione -> minor complessità.
Una classe si deve dividere se devo riusarla per altri scopi.

Test

Un software è corretto perché, leggo i requisiti, progetto i test sulla base dei requisiti, valuto il risultato calcolato ed eseguo il software: se i risultati coincidono, allora il software è corretto

- Verificare tramite test serve per **CONVALIDARE il sistema software**.

Si distingue:

- **Verifica: Software inerente alle specifiche** (test approvati)
- **Validazione: Convalidare software:** il software è corretto. *Ma è anche utile?* Il cliente darà un feedback sul software prodotto. *"Il software fa quello che deve fare ma non mi soddisfa"*. Quindi c'è stata una mancanza di specifiche all'inizio.

Processo V & V (Verifica e Validazione)

Bisogna verificare che non ci siano contraddizioni e che il cliente sia soddisfatto (convalida).

La progettazione può avere un processo V&V: l'architettura , gli algoritmi scelti sono quelli voluti.

- V&V sul codice si fa tramite test. Si deve verificare se ci sono problemi a livello di codice: magari uso costrutti che generano risultati non voluti. Magari ci sono errori di scrittura nel codice. Il tipo di verifica che si fa (test) fa emergere situazioni di questo tipo che potrebbero essere sparsi su più punti del codice. Ne segue che potrebbero **nascere certi problemi che prima ci erano sfuggiti**

Fase di Test: scrittura, esecuzione, analisi del risultato del test nei modi seguenti:

- **debug** se i risultati **NON** sono quelli aspettati
- **andare avanti** se i risultati sono quelli aspettati

Se il test esercita piccole parti di codice, allora risulta più semplice individuare l'errore

Test: definizioni

I **dati dei test sono dati in input** (servono anche per portare ad uno certo stato determinate istanze/variabili) e vi sono degli **output stimati** (che serve per controllo del risultato dei test)

- **Caso di test** (*Test case*) è l'insieme dell'eseguibile, output stimato e dati in input
- **Test suite** = Insieme dei test case

Debug con Stream

Gli Stream possono essere usati anche per effettuare debugging. Ogni operazione produce uno stream diverso rispetto a quello che ha ricevuto in input.

```
//Esempio
List numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
    .map(x -> x + 17)
    .filter(x -> x % 2 == 0)
    .limit(3)
    .forEach(System.out::println);
// Output: 20 22
```

Se si vuole capire cosa sta succedendo durante le operazioni sopra scritte, prima di forEach (che risulta l'unica operazione utile che riesce a mostrarmi l'output), allora si può usare `peek()`.

`peek()` **non modifica lo Stream** su cui opera ed è un'operazione che solitamente si usa solo per il debug. Quindi è possibile mostrare cosa sta succedendo nelle varie operazioni dello Stream:

```
List numbers = Arrays.asList(2, 3, 4, 5);
numbers.stream()
    .peek(x -> System.out.println("from stream: " + x))
    .map(x -> x + 17)
    .peek(x -> System.out.println("after map: " + x))
    .filter(x -> x % 2 == 0)
    .peek(x -> System.out.println("after filter: " + x))
    .limit(3)
    .peek(x -> System.out.println("after limit: " + x))
    .collect(Collectors.toList());

// Output:
// from stream: 2
// after map: 19
// from stream: 3
// after map: 20
// after filter: 20
// after limit: 20
// from stream: 4
// after map: 21
// from stream: 5
// after map: 22
// after filter: 22
// after limit: 22
```

Usare la `peek()` comporta delle modifiche al comportamento che ogni singolo elemento fa durante l'esecuzione. Può accadere che solo 2 (o più) fanno prima la `map` e poi la `filter` (per poi tornare alla `map`). Insomma, l'ordine delle operazioni è un po' più "per i fatti suoi" quando si usa la `peek()` ma comunque sia il risultato in output è sempre quello desiderato.

Esercizi con gli Stream

1

Data una lista di stringhe {"author", "auto", "autocorrect", "begin", "big", "bigger", "biggish"} produrre una lista che contiene solo le stringhe che cominciano con un certo prefisso noto
Esempio: se il prefisso è "au", la lista prodotta è {"author", "auto", "autocorrect"}

- Il metodo `startsWith(String prefix)` restituisce true se il parametro passato costituisce la parte iniziale della stringa
- Usare il metodo `substring(int beginIndex, int endIndex)` della classe String che restituisce la sottostringa che inizia a beginIndex e termina a endIndex
 - Esempio: "ciao".`substring(0, 2)` restituisce "ci"

```
List<String> l1 = List.of("author", "auto", "autocorrect", "begin", "big", "bigger",  
"biggish");
```

```
final String pref = "aut";  
  
List<String> res = l1.stream()  
                    .filter(s -> s.startsWith(pref))  
                    .collect(Collectors.toList());  
risultato.forEach(System.out::println);  
  
//Applicazione equivalente che produce lo stesso risultato  
//List<String> risultato = l1.stream()  
//                                .filter(s -> s.startsWith(pref))  
//                                .toList();  
//risultato.forEach(s -> System.out.println(s));
```

2

Data una lista di stringhe : {"to", "speak", "the", "truth", "and", "pay", "your", "debts"} produrre una stringa contenente le iniziali di ciascuna stringa della lista

- Esempio: per la lista sopra si produrrà la stringa "tsttapyd"

```
List<String> l1 = List.of("to", "speak", "the", "truth", "and", "pay", "your", "debts");  
  
String res = l1.stream()  
                .map(s->substring(0,0))  
                .reduce("", (r, v) -> r.concat(v));
```

3

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

- In un triangolo, ciascun lato è minore della somma degli altri due
- Si può rappresentare la terna come un array di tre elementi interi
- `int[] t = new int[] { 2, 3, 4 };`
- Si può rappresentare la lista di terne come lista di array di interi `List<int[]> lista;`

```

List<int[]> lista = Arrays.asList(new int[] { 2, 2, 3 }, new int[] { 3, 2, 3 }, new int[] { 3, 3, 3 }, new int[] { 3, 4, 5 }, new int[] { 5, 2, 3 });

List<int[]> risultato = lista.stream
    .filter( v -> v[0] < v[1] + v[2])
    .filter( v -> v[1] < v[2] + v[0])
    .filter( v -> v[2] < v[0] + v[1])
    .map(v -> v[0] +v[1] +v[2]) //perimetro
    .toList();

```

4

- 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}
- Restituire la lista delle terne che rappresentano triangoli
- Esempio: terne {3, 5, 7}, {5, 7, 8}

```

private List lista = List.of(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);

private List verifica() {
    return 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[0] + t[1])
        .collect(Collectors.toList());
}

```

5

- Data una lista di numeri interi positivi • Verificare se la lista è ordinata
- Suggerimenti
- Si generano gli indici da 0 a n-1
- Per ciascun valore dell'indice i, si confrontano l'elemento con indice i ed il successivo, se il secondo è minore del primo la lista non è ordinata e si può fermare la verifica

- Soluzione 1
 - Ogni iterazione accede a due elementi della lista
 - L'operazione filter emette l'indice i dell'elemento che è più grande del successivo
 - Appena filter trova un elemento, con l'operazione findAny ferma la ricerca
 - Nessuna operazione conserva uno stato globale

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 12, 3);
private boolean isOrdinata() {
    return IntStream.rangeClosed(0, lista.size() - 2)
        .filter(i -> lista.get(i) > lista.get(i+1))
        .peek(v -> System.out.print(lista.get(v) + " > " + lista.get(v+1)))
        .findAny()
        .isEmpty();
}
```

- Soluzione 2
 - Si conserva uno stato che è condiviso fra varie iterazioni

```
private List<Integer> lista = Arrays.asList(2, 2, 4, 6, 3, 6, 3, 3, 4, 5);
private int prec; // conserva l'elemento precedente, e' lo stato condiviso
private boolean isOrdinata() {
    prec = lista.get(0);
    return lista.stream()
        .filter(v -> seMinoreDiPrec(v)) // modifica prec, scarta valori false
        .findAny() // si ferma quando vi e' un false
        .isEmpty();
}
private boolean seMinoreDiPrec(int x) { // non puo' eseguire in parallelo
    int p = prec;
    prec = x; // modifica lo stato
    return x < p; // ritorna true se elemento corrente > elemento prec
}
```

Considerazioni esercizio 5

- Per la soluzione 1, si ha

```
.filter(i -> lista.get(i) > lista.get(i+1))
```

- l'espressione lambda passata a filter legge due numeri dalla lista e dà in output un boolean, senza altri effetti collaterali (side-effect-free), ovvero **non modifica uno stato condiviso**. Tale espressione lambda è una **funzione pura**

- Per la soluzione 2, si ha

```
.filter(v -> seMinoreDiPrec(v))
```

ovvero

```
.filter(v -> { int p = prec; prec = v; return v < p; })
```

- l'espressione lambda passata a map modifica un attributo. Tale modifica è un effetto collaterale (voluto), quindi non è una funzione pura
- Si noti che: (i) prec è un attributo, definito al di fuori dell'espressione lambda che può essere acceduto da essa (accessi al contesto sono consentiti); (ii) la modifica di uno stato condiviso non permette l'esecuzione parallela

In generale:

- La seconda soluzione non può essere usata per una soluzione parallela (parallel) perchè modifica uno stato globale e quindi uno stato condiviso.
- La prima soluzione funziona anche per una possibile esecuzione parallela visto che non vi è alcuna modifica ad uno stato globale(condiviso)

Function<T,R>

Una map() prende in ingresso una `Function<T, R>` prende in input un T e restituisce R

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

int result = Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature") // ->
Stream<String>
    .map(stringLength) // -> Stream<Integer>
    .reduce(0, Integer::sum); // 31
```

Function ha al suo interno un metodo chiamato `apply()` e quando si passa l'espressione lambda viene usata proprio questa.

BiFunction

Invece, una BI-FUNCTION è una Function del tipo `BiFunction<T1, T2, R>` e prende in ingresso T1 e T2 e restituisce R.

Supplier<T>

E' un'interfaccia funzionale che ha un singolo metodo chiamato `get()`. **Non prende in input nessun parametro e restituisce un unico valore** di tipo T.

- Applica un'espressione lambda del tipo `() -> //code`

```
Supplier sup = () -> "ciao ciao";
String s = sup.get(); // s: ciao ciao
```

- `sup` non sta tenendo una stringa ma il codice che lo genera.

Factory Method (Ripasso)

Il Factory fa un lavoro del genere:

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

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();
        }
    }
}
```

Quindi il Creator sceglie quale classe istanziare.

Factory con Supplier

- Usando un `Supplier`

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

- La linea di codice sopra è equivalente a

```
Supplier<Prodotto> suppl = () -> new ProdottoA();
```

- Per avere istanze di sottotipi di `Prodotto`, si chiama `get()` sul `Supplier`

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

- Quindi si crea una mappa che fa 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);
```

- Si usa la mappa per istanziare sottotipi di `Prodotto`

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

- Il frammento di codice sopra è equivalente a

```
public static Prodotto getProdotto(String name) {
    return map.getOrDefault(name, ProdottoA::new).get();
}
```

Si usa un `Supplier` per ogni `ConcreteProduct` che servono.

- L'istanza viene creata alla chiamata di `get()`.

- Si **evita di scrivere molte condizioni** e ciò equivale a scrivere codice migliore
- La mappa che si crea non contiene istanze ma avrà solo la coppia (chiave, valore) dove la chiave rappresenterà il parametro che un possibile client passa al metodo in modo da sapere quale istanza creare

Se si vuole evitare anche la parte condizionale seguente:

```
if (s != null)
    return s.get();
return new ProdottoA();
```

Si può usare un metodo della Map stessa nel seguente modo:

```
public static Prodotto getProdotto(String name) {
    return map.getOrDefault(name, ProdottoA::new).get();
}
```

- Il metodo `getOrDefault()` **ritorna il valore riferito alla chiave** passata `name` altrimenti, se non esiste, **ritorna un default** definito dal programmatore

Esempio Classe Persona

```

public class Persona {
    private String nome, ruolo;
    private int eta, costo;

    public Persona(String n, int e, String r, int c) {
        nome = n;
        eta = e;
        ruolo = r;
        costo = c;
    }

    public int getCosto() {
        return costo;
    }

    public int getEta() {
        return eta;
    }

    public String getNome() {
        return nome;
    }

    public String getRuolo() {
        return ruolo;
    }
}

```

5

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

```

private List<Persona> team = List.of(
    new Persona("Al", 28, "Architect", 44),
    new Persona("Claire", 29, "Programmer", 38),
    new Persona("Ed", 26, "Programmer", 36),
    new Persona("Pam", 25, "Programmer", 35),
    new Persona("Ted", 32, "Tester", 40));

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("\nCi sono " + c + " " + ruolo);
}

// Output
// Hanno ruolo Programmer: Claire, Ed, Pam,
// Ci sono 3 Programmer

// chiamante
conta("Programmer");

```

- Data una lista di istanze 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() + " ")));
}

// Output
// Ruolo Architect: Al
// Ruolo Programmer: Claire Ed Pam
// Ruolo Tester: Ted
```

Esempio Classe Pagamento

```

public class Pagamento {
    private Persona pers;
    private int importo;

    public Pagamento(Persona p, int v) {
        pers = p;
        importo = v;
    }

    public Persona getPers() {
        return pers;
    }

    public int getImporto() {
        return importo;
    }
}

```

- 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

```

private List<String> daPagare = List.of("Pam", "Ed", "Ted");

private List<Pagamento> pagati = new ArrayList<>();

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());
}

// Output
// Ed 1080  Pam 1050  Ted 1200

```

```
public class BustaPaga {
    private Persona pers;
    private int totale;

    public BustaPaga(Persona p) {
        pers = p;
    }

    public void calcolaCostoBase() {
        totale = pers.getCosto() * 30;
    }

    public void aggiungiBonus() {
        totale = (int) Math.round(totale * 1.1);
    }

    public Persona getPersona() {
        return pers;
    }

    public void stampa() {
        System.out.println(pers.getNome() + "\t " + totale + " euro");
    }

    public int getImporto() {
        return totale;
    }
}
```

- 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

```
private List<BustaPaga> buste;

public void generaBustePaga() {
    buste = team.stream()
        .map(p -> new BustaPaga(p))
        .peek(b -> b.calcolaCostoBase())
        .peek(b -> b.aggiungiBonus())
        .sorted(Comparator.comparing(b -> b.getPersona().getNome()))
        .collect(Collectors.toList());
}
```

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

```
public int calcolaSomma() {
    return buste.stream()
        .peek(b -> b.stampa())
        .mapToInt(b -> b.getImporto())
        .sum();
}

// Output
// Al      1452 euro
// Claire  1254 euro
// Ed      1188 euro
// Pam     1155 euro
// Ted     1320 euro
// Totale: 6369 euro

// chiamante
System.out.println("Totale:\t " + calcolaSomma() + " euro");
```

Test

Se il test rileva degli errori dipende tutto dal tipo di esecuzione che si è avviata all'interno del test.

Se il test **non segnala errori non si può dire che il programma è corretto** ma si dice che il test non rileva errori. Questo perchè non vengono analizzate tutte le possibili esecuzioni di un programma.

Per dire che un software è corretto **si devono avere dei test esaustivi**: essi scandiscono il funzionamento per ogni tipo di esecuzione e ciò è **estremamente difficile**. Quindi si conclude che **i test esaustivi sono impraticabili**

In genere non si ha il tempo necessario per eseguire tutti i possibili test.

- Se si avesse un metodo che prende in ingresso 2 interi, allora servirebbero circa 2^{32} valori per ogni intero.

- dovrebbe essere eseguita $2^{32} * 2^{32}$ volte, ovvero circa $1.8 * 10^{19}$ volte
- Se la funzione esegue in $1ns = 10^{-9}s$ occorrono $1.8 * 10^{10}s$ ovvero, essendo $1Y = 3 * 10^7 \rightarrow 600$ anni!

Strategie di Test

Si possono scrivere test Efficaci (non esaustivi) e cercare in tutti i modi possibili dove si potrebbe annidare il difetto.

Per scrivere test efficaci, allora, ci sono delle tecniche:

- Si devono controllare gli input che si aspetta il sistema
- Si crea un **set di input VALIDI** che è lecito dare ad un'esecuzione di un pezzo di programma
- Si deve pensare anche a un **set di input NON VALIDI**
- I difetti, spesso, si manifestano in gruppi e ciò comporta che bisogna indagare ulteriormente perché un codice che presenta errori, ne porta altri di seguito. Questo perché se ci sono errori, vuol dire che la soluzione messa in atto non era molto leggibile e non si è applicata la strategia migliore. Quindi **UN CODICE COMPLICATO PRODUCE ERRORI**
- i test devono poter essere riusati ogni volta che si aggiunge nuovo codice al sorgente
- Ogni test **deve invocare un solo metodo** così che si può riusare molto facilmente. Ha solo un determinato insieme di input e un solo insieme di output stimati.
- i test **non devono dipendere** gli uni dagli altri perché altrimenti viene meno il concetto di riusabilità del singolo test

Strategia di test white-box (*glass-box o strutturale*)

In questo caso **si analizza il codice e le relative specifiche** per quel codice.

Quindi dal codice si capisce il tipo di test da eseguire

Strategia di test black-box

Non si guarda il codice ma solo le specifiche. Per il codice deve solo sapere i punti di aggancio, quindi solamente le chiamate che si devono fare per invocare il test. Inoltre, ovviamente, chi scrive i test, conosce anche i possibili input che si devono dare in pasto al test e i relativi output stimati.

Visto che non si può scandagliare 2^{64} valori per i possibili output di un codice, fra questi valori, quali si scelgono?

Partizionamento in classi equivalenti

Il progettista, ragionando su quello che deve eseguire un metodo, sa che **un metodo si comporta allo stesso modo per una gran parte di tipo di input.**

*Esempio: un metodo si comporta allo stesso modo se gli si passa, **come input, una stringa.**

Oppure, un altro esempio di partizione è il seguente: i valori in **input da usare devono essere compresi fra (0-1000)***

Quindi ci si trova nella stessa categoria (**PARTIZIONE**) di input. Chi scrive il codice sceglie un tipo di input da tutte le partizioni che si "vengono a definire".

- Aver partizionato concede **molti vantaggi nella scelta degli input** per un determinato test.
- Se si nota che l'algoritmo si comporta in maniera diversa con un altro tipo di input, allora si va a **scegliere un input di una diversa partizione**

Si parla di **BOUNDED-CONDITION** ovvero condizioni racchiuse in un certo partizionamento

Chi fa il test deve considerare sia classi di equivalenza **VALIDE** che classi di equivalenza **NON VALIDI**.
Una classe di equivalenza non valida rappresenta input inaspettati o errati.

I test si devono eseguire con gli input provenienti dalle ZONE LIMITE di un loro partizionamento:

Esempio: Se il partizionamento degli input è un range fra {0-100} allora si faranno test con i seguenti possibili input: (-1, 0, 1) e (99, 100, 101)

Test del percorso

In fase di test si deve considerare anche la possibilità di prendere più percorsi. Per tale motivo si deve fare un test per ogni possibile percorso.

La **complessità ciclomatica** dà il numero di test di percorso che si devono fare.

In input, comunque, devono essere usati input che creano esclusivamente un percorso evitando così che si copra un percorso 2 volte.

La complessità ciclomatica $cc = \text{numero_archi} - \text{numero_nodi} + 2$

Test sotto stress

Per gli input **PIU' PROBABILI** e le esecuzioni del programma più frequenti si devono sicuramente scrivere dei test.

Si devono eseguire dei test anche sotto stress ovvero in situazioni limite del software:

*Se la memoria della macchina si sta esaurendo, il sistema riesce comunque a funzionare o si blocca?
E se il disco è pieno?
Se nel software ci sono 100 record funziona. Ma se ce ne sono 1000000000 funziona allo stesso modo?*

Copertura del codice

Se si combinano molti pezzi di codice, si ha una combinazione di flussi esponenziale.

La copertura (**COVERAGE**) indica la percentuale di codice eseguito rispetto al totale del programma.

Più copertura si ha, più il codice è stato testato. Si deve capire se c'è una maggiore copertura del codice se si implementa un nuovo test, altrimenti vorrà dire che il test è sbagliato e ripercorre un test già percorso.

Una copertura 100% non significa i test sono esaustivi ma che si ha una copertura con determinati parametri in input (*quindi non tutti i possibili*)

Si deve arrivare al **60% di code coverage** di solito per considerare il codice "testato bene"

Criteri di copertura del codice

Alcuni criteri sono:

- Si devono coprire con test tutte le funzioni
- si devono testare tutte le linee di codice (copertura del costrutto)

- Se tutto il sistema è come un grafo, si devono testare tutti i possibili archi del grafo, quindi tutte le possibili chiamate all'interno del sistema

*Esempio: **un costrutto condizionale** deve essere attraversato usando **2 input diversi** che logicamente percorrono il costrutto con una **condizione vera** e una **condizione falsa***

Copertura delle decisioni

Si devono **coprire anche tutte le possibili condizioni** (se per caso ce ne sono più di una) quindi devono usare tutti i possibili input usati come condizione

Trend dei difetti scoperti

Quando si eseguono dei test si scopre se il codice è corretto.

Un criterio per continuare a scrivere test è la copertura: più copertura si ha, più si è soddisfatti e vuol dire che il sistema è stato testato più sufficientemente

Un altro criterio è IL TREND DEI DIFETTI: si calcola il numero di difetti che man mano si scoprono con le scritture dei test.

Scrivo 100 test, quanti difetti scopro?

Man mano che scopro i difetti (e li correggo man mano), il residuo dei difetti presenti è difficile da rivelare.

Pertanto, il criterio è: **man mano che si aggiungono test, il numero di difetti deve decrescere**

stabilendo una soglia e quando raggiungo il minimo definito dalla soglia, ci si ferma con la stesura di test.