



NoiStudiamo

APPUNTI



Ingegneria del Software

Appunti completi: OOP, Design Pattern, UML, Testing e Processi Agili.

A CURA DI

Carola Aiello

Ottobre 2026

Indice degli argomenti

Gli argomenti principali trattati negli appunti.

1. Introduzione e Qualità del Software	3
2. Test, TDD e Principi di Progettazione	7
3. Java, JUnit e Maven	8
4. Refactoring e Code Smell	12
5. Classi, Oggetti e Singleton	17
6. Ereditarietà, Interfacce e Polimorfismo	21
7. Design Pattern Template Method	27
8. Design Pattern Creazionali	30
9. Design Pattern Strutturali	39
10. Evoluzione di Java e Programmazione Funzionale	43
11. Stream e Programmazione Parallela	54
12. Composite e Decorator	61
13. Diagrammi UML	68
14. Processo Software e Metodologie Agili	72
15. Design Pattern Comportamentali	83
16. Progettazione e Metriche Software	93
17. Testing e Difetti del Software	99

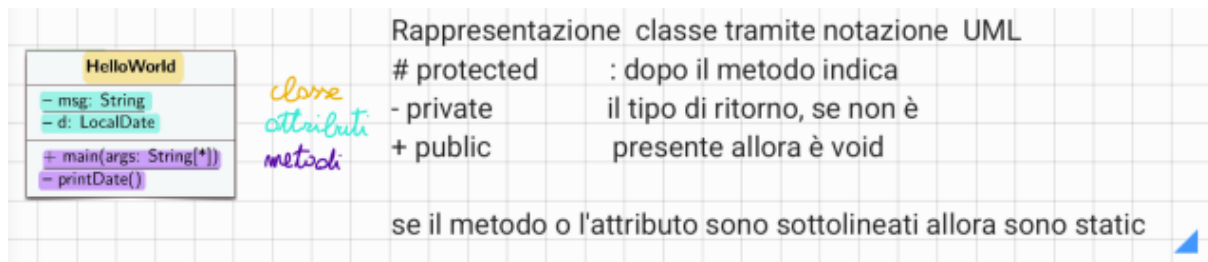
Ingegneria del Software

Un processo software descrive l'insieme delle attività, o task, necessarie allo sviluppo di un prodotto software e definisce come queste attività sono collegate logicamente tra loro.

L'intero ciclo di vita dello sviluppo si articola in cinque fasi fondamentali:

1. Analisi dei requisiti;
2. Progettazione (OOP, Design Pattern, Refactoring);
3. Codifica o implementazione;
4. Testing;
5. Manutenzione.

Lo standard UML permette di visualizzare il software ancor prima di scrivere linee di codice, Java invece permetterà l'implementazione della programmazione ad oggetto, uno dei linguaggi più richiesti e diffusi al mondo.



Ha senso studiare Ingegneria del Software nell'era della AI?

Sì, l'intelligenza artificiale riusa algoritmi e soluzioni esistenti ma non progetta l'intero sistema e può generare codice con difetti o gravi falle di sicurezza. Un informatico con competenze e conoscenze in fatto di Ingegneria del Software sarà in grado di utilizzare l'AI come supporto e non come strumento.

Una caratteristica fondamentale di un software è il suo essere **intrinsecamente modificabile**, non essendo composto da componenti fisiche l'unica barriera da abbattere è quella logica. Un software di successo deve essere in grado di soddisfare tutte le richieste di espansione e adattamento da parte degli utenti, di conseguenza deve essere altamente modificabile.

Criteri di qualità di un Software

L'obiettivo dell'Ingegneria del Software è di produrre sistemi complessi garantendo un equilibrio tra **costi, tempi e qualità**.

Esistono dei criteri operativi per valutare la qualità di un sistema:

- **Correttezza:** il sistema Software deve aderire allo scopo e deve essere conforme alle specifiche richieste dal cliente. Dobbiamo chiederci “Il sistema fa quello che il cliente vuole?”, “Il sistema soddisfa le specifiche che erano state raccolte?”, per rispondere a queste domande bisogna fare dei **Test**;
- **Efficienza:** un Software di qualità deve operare garantendo l'assenza di sprechi di risorse;
- **Manutenibilità:** un sistema è manutenibile se permette di introdurre modifiche, correzioni o estensioni in maniera semplice, rapida e senza introdurre nuovi errori;
- **Dependability:** questo concetto si suddivide a sua volta in due parti:
 - **Sicurezza:** per sicurezza intendiamo **security**, ovvero la protezione dei dati da attacchi esterni, e la **safety**, ossia la prevenzione di danni a persone o all'ambiente;
 - **Affidabilità:** per affidabilità invece intendiamo **reliability**, ovvero la probabilità che il sistema operi senza guasti;
- **Usabilità:** un Software di qualità deve essere intuitivo e fornire condizioni d'uso efficaci, minimizzando lo sforzo cognitivo per interagire con le sue funzionalità.

Un sistema Software sta dando i risultati corretti se soddisfa tutti i requisiti sopra elencati.

In Java useremo il principio: **programmare per le interfacce e non per le implementazioni**. Ad esempio, in Java, se serve una lista di dati si dovrebbe dichiarare la variabile usando l'interfaccia List piuttosto che usare direttamente la classe concreta ArrayList che la implementa. Questo ci permette di ridurre la propagazione di cambiamenti quando si dovrà modificare il codice in futuro e rende più semplice l'implementazione dei test.

Nella notazione UML un'interfaccia può essere rappresentata in 3 modi equivalenti:



Lo spaghetti code è un tipo di programmazione detto anti-pattern, racchiude tutte le cattive abitudini della programmazione.

Il problema principale è **implementare tante cose in unico flusso**, un unico metodo. Uno spaghetti code risulta quindi troppo sequenziale e con poca possibilità di manipolazione, le conseguenze sono un codice **poco leggibile e difficile da riutilizzare e adattare**.

La soluzione risiede nel progettare correttamente la classe assicurandosi che **ogni metodo abbia una sola responsabilità**.

Nella programmazione ad oggetti è molto importante l'utilizzo del **paradigma Command e Query**:

- I metodi **Query** restituiscono un valore senza modificare lo stato del sistema e possono essere chiamati quante volte vogliamo;
- I metodi **Command** cambiano lo stato interno del sistema ma non restituiscono nulla e vanno gestiti con più attenzione.

Ad esempio se andiamo a guardare getter e setter di una classe distinguiamo i getter come metodi query e setter come metodi command.

Il costrutto sintattico **Enhanced for** indica esplicitamente che si vogliono scorrere gli elementi di una lista, ottenendone uno per ogni passata.

Un esempio di enhanced for per scorrere una lista e sommarne i valori, come visto nell'implementazione della classe Pagamenti, è il seguente:

```
public void calcolaSomma() {  
    this.totale = 0;  
    // Enhanced for: per ogni float 'v' nella collezione 'valori'  
    for (float v : valori) {  
        this.totale += v;  
    }  
}
```

Questo approccio permette di scorrere tutti gli elementi di oggetti che implementano l'interfaccia Iterable (come le Liste) senza dover gestire manualmente un indice, rendendo il codice più pulito, leggibile e meno prone a errori di limite (out of bounds).

- Esempi di domande d'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

Miglioramento sul codice

Il primo passo per ottenere un codice di qualità è la corretta **suddivisione delle responsabilità**. Separare le diverse operazioni, incapsulandole ciascuna in un metodo specifico. Questo approccio offre due grandi vantaggi:

1. più semplice verificare la correttezza delle singole operazioni;
2. favorisce il riutilizzo in altre parti del programma.

Nella programmazione orientata agli oggetti l'**astrazione** risulta essere molto utile. Un'astrazione tipicamente è una classe progettata per nascondere i dettagli di implementazione al resto del sistema. Questo processo prevede la creazione di metodi di "basso livello" (che svolgono compiti elementari) e metodi di "alto livello" (che richiamano i precedenti per logiche più complesse). Il nome che si dà a classi e metodi ne descrive l'obiettivo.

È buona norma adottare una netta separazione tra le operazioni che interrogano (**query**) il sistema e quelle che lo modificano (**command**).

Infine per fare in modo che il codice sia collaudabile lo stato dell'oggetto deve diventare "osservabile" dall'esterno attraverso l'uso di metodi opportuni.

Test

Il test è un programma che esegue il codice sotto certe condizioni (valori in input) e valuta se il risultato fornito è corretto. Per essere affidabili i test devono essere indipendenti l'uno dall'altro e devono sapersi autovalutare confrontando l'esito con il risultato atteso.

Qualità

Soddisfare e verificare i requisiti tramite i test permette di valutare e ottenere la qualità del sistema software.

Una pratica molto molto diffusa è il **TDD (Test Driven Development)**, che consiste nell'implementare prima il codice del test e solo dopo il codice dell'applicazione che risolve quel particolare requisito.

Il ciclo TDD si basa sul motto **Red Green Refactor**:

- **Red**: il test non viene superato, quindi si possono implementare e modificare le classi;
- **Green**: il test viene superato con successo, quindi si possono implementare test o si può fare refactoring;
- **Refactor**: si migliora la struttura del codice ora che si è certi del suo corretto funzionamento.

Una variante di questo approccio è il **BDD (Behaviour Driven Development)**, che struttura la sua logica su 3 cardini temporali:

- **Given**: dato uno specifico contesto di partenza;
- **When**: quando succede un determinato evento;
- **Then**: allora il sistema deve rispondere in un certo modo.

I test scritti devono inoltre seguire l'acronimo **FIRST**: devono essere veloci (Fast), isolati e indipendenti l'uno dall'altro (Isolated/Independant), ripetibili (Repeatable), in grado di validarsi da soli (Self-validating) e scritti in maniera tempestiva (Timely).

Principio della Singola Responsabilità

Per mantenere un alto livello qualitativo la coesione del codice deve essere elevata e per ottenere ciò si utilizza il Principio di Singola Responsabilità (SRP), non vi deve essere una classe o un metodo che cerca di implementare troppi compiti simultaneamente: bisogna sempre avere un solo motivo per cambiare il codice di quella parte, perciò ogni componente deve svolgere un singolo e piccolo compito.

Altri principi fondamentali

Oltre l'SRP vi sono altri grandi capisaldi dell'Ingegneria del Software:

- **KISS** (Keep It Simple, Stupid): Produrre componenti semplici, facili da capire e gestire;
- **DRY** (Don't Repeat Yourself): evitare ripetizioni nel codice;
- **YAGNI** (You Ain't Gonna Need It): non progettare funzionalità che non sono attualmente utili;
- **LoD** (Law of Demeter): non parlare con gli estranei, un oggetto dovrebbe avere conoscenza solo di oggetti correlati.

Compilazione Java

Infine, passiamo a qualche concetto pratico sul linguaggio Java. Quando all'interno di una tua classe Java devi usare tipi di dato o librerie particolari (ad esempio le classi `List` o `File`), devi specificarlo inserendo all'inizio del tuo documento `.java` l'istruzione `import` (come ad esempio `import java.io.File;`).

Per **compilare** si utilizza il comando `javac NomeClasse.java`, se non ci sono errori e la compilazione va a buon fine, il terminale non emetterà nessun messaggio e creerà in automatico un eseguibile chiamato `NomeClasse.class`.

Per **eseguire** si utilizza il comando `java NomeClasse.java`

- 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

JUnit

JUnit è un framework specificatamente progettato per permettere di scrivere ed eseguire test automatici su classi e metodi del nostro applicativo. Quest framework fornisce anche delle API dedicate.

Il suo funzionamento si basa principalmente sull'uso delle **annotazioni** Java, ossia metadati che associamo ad elementi del nostro codice, come classi, metodi o campi che servono a fornire informazioni aggiuntive al compilatore o al framework stesso in fase di esecuzione.

Ecco alcune annotazioni chiave:

- **@Test**: identifica un metodo di test;
- **@BeforeEach**: esegue il metodo prima di ogni test;
- **@BeforeAll**: esegue il metodo una volta prima di tutti i test.

Per verificare il comportamento del codice, JUnit mette a disposizione i metodi della classe **Assertions**, le più comuni sono:

- **assertEquals(expected, actual)**: per controllare che un valore ottenuto coincida con quello atteso;
- **assertTrue(condition)**: per validare un'espressione booleana;

- `assertNull(object)`: per accertarsi che un dato oggetto non sia stato istanziato.

L'annotazione

`@ParameterizedTest` permette di eseguire un metodo più volte passando in input valori diversi senza inserire i valori nel codice di test. Un modo per fornire i valori è tramite `@CsvSource`

```
public class CalcTest {
    private final Calc calculator = new Calc();

    @ParameterizedTest
    @CsvSource({
        "1, 1, 2",
        "2, 3, 5",
        "10, 5, 15",
        "-1, 1, 0"
    })
    void testAdd(int a, int b, int expected) {
        assertEquals(expected, result);
        int result = calculator.add(a, b);
    }
}
```

Code coverage

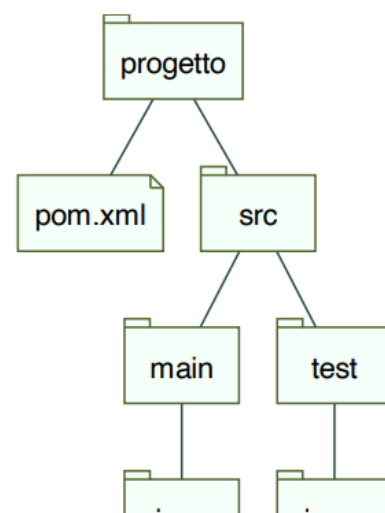
Quando eseguiamo i test, stiamo eseguendo porzioni dell'applicazione principale. La copertura è una metrica espressa in percentuale che indica il rapporto tra linee di codice attraversate dal test e linee totali che compongono l'applicazione. Se otteniamo una copertura del 70%, significa che il 70% delle istruzioni del programma è stato verificato in fase di test. Arrivare a una copertura del 100% significa che i nostri test sono passati attraverso ogni singola riga di codice.

Maven

Maven è uno strumento standard del settore per la gestione dei progetti e l'automazione delle procedure di build. Il suo vantaggio è imporre una serie di convenzioni per standardizzare la compilazione, gestione delle librerie esterne, il testing, la pacchettizzazione e il rilascio del software.

A causa di queste convenzioni la struttura delle cartelle è fissa e predeterminata:

- il codice sorgente dell'applicazione si trova nella cartella a `src/main/java`;
- tutti i test JUnit risiedono in `src/test/java`;
- i vari package del programma vengono inseriti all'interno delle specifiche cartelle `java`;
- il cervello di tutto il progetto Maven è un singolo file chiamato `pom.xml`,



acronimo di Project Object Model. Utilizzando dei tag XML predefiniti questo file descrive tutto ciò che riguarda il progetto: il nome, la versione attuale, i plugin necessari e le librerie esterne da includere.

Maven: File POM

```
<project>
  <groupId>com.example</groupId>
  <artifactId>prj-junit6</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>25</maven.compiler.source>
    <maven.compiler.target>25</maven.compiler.target>
    <maven.compiler.release>25</maven.compiler.release>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>6.0.3</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>6.0.3</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Prof. Tramontana - Marzo 2026

Operativamente, quando prepariamo un progetto (tramite un IDE o con il comando `mvn archetype:generate`), e capiamo di aver bisogno di uno strumento come JUnit, andiamo a cercare la libreria sul "Maven Central" (il repository globale) e copiamo il relativo blocco XML `<dependency>` nel nostro `pom.xml`. A quel punto, Maven si preoccuperà di scaricare in totale autonomia sia la libreria richiesta sia tutte le eventuali dipendenze secondarie ad essa collegate, secondo le direttive fornite dagli sviluppatori della libreria stessa.

Infine, Maven ci offre dei comandi standard per gestire le diverse "fasi" della vita del software. Una regola fondamentale di Maven è che richiamare una determinata fase comporta l'esecuzione automatica di tutte le fasi precedenti a essa. I comandi principali che utilizzeremo sono:

- `mvn compile`, che si limita a compilare il codice sorgente.
- `mvn test`, che avvia l'esecuzione di tutti i test di unità.
- `mvn package`, che si occupa di prendere il codice compilato e testato per produrre un archivio distribuibile (ad esempio un file `.jar`).
- `mvn verify`, utilizzato per avviare controlli di qualità più approfonditi.

All'interno del file `pom.xml` è inoltre possibile configurare ulteriori plugin per espandere queste fasi o gestire operazioni complesse come il deploy dell'applicazione finale.

Refactoring

Il 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**. Un esempio molto comune di questa pratica è la rimozione di frammenti di codice duplicati all'interno dei rami condizionali.

I vantaggi sono diversi:

- dimensione del codice ridotta;
- strutture complicate si trasformano in strutture più semplici e più facili da capire e mantenere;
- si evita di introdurre un **debito tecnico** all'interno della progettazione.

Perché fare refactoring?

Senza un'attività di refactoring, la progettazione del software tende a deteriorarsi inevitabilmente man mano che si apportano nuove modifiche o si implementano nuove funzioni. Il refactoring aiuta concretamente a scoprire bug e difetti in modo più rapido, promuovendo una comprensione profonda del codice, aumentando così la velocità di programmazione.

Dopo ogni intervento, bisogna sempre verificare che tutti i test continuino ad essere superati con successo. Se un test fallisce significa che il refactoring ha compromesso una funzionalità operativa.

Code Smell

Per capire se è necessario il refactoring bisogna prestare attenzione a specifici sintomi progettuali, comunemente noti come **bad smell**:

- **codice duplicato**, criticità risolvibile applicando la tecnica **Estrai Metodo**;
- **metodi lunghi**, affrontabili con la tecnica **Estrai Metodo** insieme alla tecnica **Sostituisci Tempo con Query**;
- **classi di grandi dimensioni**, scomponibile con la tecnica **Estrai classe**;
- **lunga lista di parametri per un metodo**, sistemabile usando la tecnica **Introduci Oggetto Parametro**;
- **cambiamenti divergenti**, ossia quando si è costretti a modificare in continuazione parti diverse della stessa classe per motivi funzionali differenti, risolvibile con la tecnica **Estrai classe**.

Tecnica Estrai Metodo

Questa tecnica è mirata a ridurre drasticamente la lunghezza dei metodi o a chiarire frammenti di logica che risultano oscuri senza l'ausilio di un commento.

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

Prima

Dopo

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

L'operazione consiste nell'isolare un determinato frammento di codice racchiudendolo all'interno di un nuovo metodo, al quale va assegnato un nome estremamente chiaro ed esplicativo del suo scopo. **Metodi più piccoli e coesi** hanno non solo una maggiore probabilità di essere riutilizzati all'interno del programma, ma rendono anche i metodi di livello gerarchico superiore estremamente leggibili, quasi come fossero costituiti da una sequenza discorsiva di commenti. Questa brevità semplifica notevolmente anche eventuali operazioni di ridefinizione (overriding) nelle sottoclassi.

Bisogna però fare attenzione alle variabili locali. Quelle necessarie passano come parametri al nuovo metodo mentre le altre diventano variabili temporanee locali. 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 della chiamata alla variabile locale del metodo sorgente.

Tecnica Sostituisci Temp con Query

Spesso si utilizzano le variabili temporanee (“temp”) per immagazzinare il risultato di un’espressione matematica o logica. Tuttavia esse forzano lo sviluppatore a scrivere metodi lunghi perchè quello è l’unico spazio in cui quel dato è accessibile.

```
public double getPrezzo1() {  
    double prezzoBase = quantita * prezzo;  
    double sconto;  
  
    if (prezzoBase > 1000) sconto = 0.95;  
    else sconto = 0.98;  
  
    return prezzoBase * sconto;  
}
```

Prima

Dopo

```
public double getPrezzo2() {  
    return prezzoBase() * sconto();  
}  
  
private double prezzoBase() {  
    return quantita * prezzo;  
}  
  
private double sconto() {  
    if (prezzoBase() > 1000) return 0.95;  
    return 0.98;  
}
```

La soluzione risiede nell'estrarre quell'espressione ponendola all'interno di un metodo dedicato e indipendente (“query”). Così facendo qualsiasi altro metodo potrà accedere a quel dato e la vecchia variabile “temp” verrà eliminata e sostituita dall'invocazione di questo metodo. Questa pratica spesso viene associata alla tecnica **Estrai metodo** ed è facilmente realizzabile se la variabile di partenza viene assegnata solo una volta.

Tecnica Dividi Variabile Temp

Questo refactoring entra in gioco quando una variabile temporanea viene assegnata più volte all'interno dello stesso metodo (esclusi casi come contatori nei cicli o accumulatori di valori).

```
double temp = 2 * (height + width);  
System.out.println(temp);  
temp = height * width;  
System.out.println(temp);
```

Prima

Dopo

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

Se una variabile riceve più assegnazioni di natura diversa, significa che sta assumendo responsabilità distinte il che confonde il lettore. Ogni variabile deve avere una e una sola responsabilità semantica. La tecnica impone di dichiarare una variabile separata, marcata come **final**.

Tecnica Sposta Metodo

Questo refactoring risolve l'accoppiamento eccessivo tra due classi. Si presenta quando il metodo di una classe finisce per utilizzare costantemente attributi o operazioni appartenenti ad un'altra classe, molto più di quanto non sfrutti quelli della propria.

```
class Punto {
    public double x;
    public double y;
}

class Geometria {
    public double calcolaDistanza(Punto p) {
        return Math.sqrt(p.x * p.x + p.y * p.y);
    }
}
```

Prima

Dopo

```
class Punto {
    public double x;
    public double y;

    public double calcolaDistanza() {
        return Math.sqrt(x * x + y * y);
    }
}

class Geometria {
    public double calcolaDistanza(Punto p) {
        return p.calcolaDistanza();
    }
}
```

L'approccio risolutivo prevede la creazione di un **nuovo metodo analogo** all'interno della classe maggiormente utilizzata e lo spostamento logico del

codice al suo interno, verificando che non ci siano dichiarazioni pendenti in eventuali superclassi.

A questo punto si può decidere se rimuovere il metodo originario o tenerlo per delegare (chiamare al suo interno il nuovo metodo nell'altra classe).

Tecnica Estrai Classe

Questo refactoring diventa obbligatorio quando ci si rende conto che una classe fa il lavoro che dovrebbero fare più classi, quindi ha troppe responsabilità.

```
class Libro {  
    public String titolo;  
    public String autoreNome;  
    public String autoreCognome;  
}
```

Prima

Dopo

```
class Autore {  
    public String nome;  
    public String cognome;  
}  
  
class Libro {  
    public String titolo;  
    public Autore autore;  
}
```

Una classe dovrebbe avere una responsabilità, ma durante lo sviluppo si aggiungono attributi e metodi e la classe cresce e una classe grande è difficile da comprendere. Se un sottoinsieme di dati ed un sotto-insieme di metodi dovrebbero andare insieme, o se un sottoinsieme di dati di solito viene cambiato insieme, o tali dati dipendono tra loro, allora è bene **dividere la classe**. Si procede creando una nuova classe e stabilendo un collegamento tra la vecchia struttura e quella nuova (dipendenza tramite istanza). Si iniziano così a spostare attributi e metodi nella nuova struttura.

Classi e Oggetti

Ogni oggetto è un'istanza di una classe e ogni classe a zero o più istanze. Le classi sono costrutti statici, ciò significa che la loro esistenza, la semantica e le relazioni reciproche sono fissate prima dell'esecuzione del programma. Anche l'appartenenza di un oggetto a una determinata classe è statica.

A livello concettuale le classi formano il vocabolario del dominio del problema che stiamo modellando, mentre gli oggetti interagiscono insieme a runtime per soddisfare i requisiti del problema stesso.

Quante istanze ho bisogno per la classe Pagamenti?

Posso avere una sola istanza per tutta l'applicazione o una istanza per ciascun file letto ma se decidessi di avere una sola istanza come posso vietare la creazione di più istanze?

```
public class ClientPagam { // versione 0.2

    public static void main(String[] args) {

        Pagamenti p = new Pagamenti(); // p è l'unica istanza

        p.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
        System.out.println("file 1 totale: " + p.calcolaSomma());
        System.out.println("file 1 max: " + p.calcolaMassimo());

        p.leggiFile("csvfiles", "PagMarzo.csv"); // lettura secondo file
        // p adesso contiene tutti i valori letti da entrambi i file

        System.out.println("file 1 e 2 totale: " + p.calcolaSomma());
        System.out.println("file 1 e 2 max: " + p.calcolaMassimo());
    }
}
```

1 istanza,
2 letture

```
public class MainPagam { // versione 2.3

    public static void main(String[] args) {

        Pagamenti p1 = new Pagamenti(); // prima istanza
        Pagamenti p2 = new Pagamenti(); // seconda istanza

        p1.leggiFile("csvfiles", "Importi.csv"); // lettura primo file
        p2.leggiFile("csvfiles", "PagMarzo.csv"); // lettura secondo file
```

2 istanze, 2 letture

Design Pattern Singleton

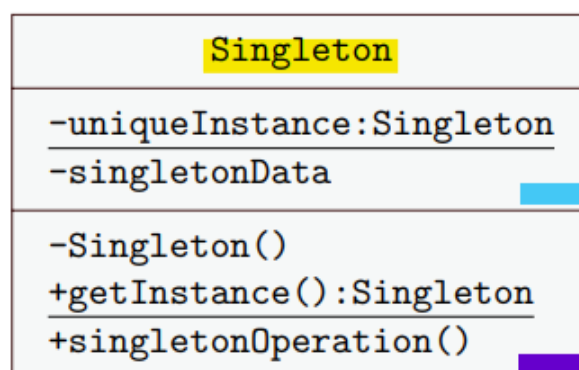
La risposta a questa necessità è il Design Pattern Singleton.

L'intento di questo pattern è quello di assicurare che una classe abbia una e una sola istanza fornendo un punto di accesso globale a tale istanza.

Le motivazioni dietro al suo utilizzo sono molteplici: ci sono classi che per natura dovrebbero esistere in un'unica istanza all'interno di un'applicazione, come ad esempio uno spooler di stampa. Può sorgere spontanea la domanda perchè non usare una semplice variabile globale? Il motivo è che una variabile globale rende un oggetto accessibile ovunque ma non proibisce l'istanziamento di più oggetti per la stessa classe. La classe stessa dovrebbe essere responsabile di tener traccia del suo unico punto di accesso.

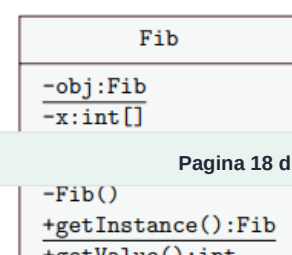
La soluzione tecnica richiede due elementi chiave:

- il costruttore della classe Singleton deve essere reso **privato**, per non permettere la creazione di oggetti tramite la keyword **new** ad altre classi. La classe Singleton diventa così l'unica responsabile per la creazione della propria istanza;
- La classe deve implementare un'operazione **static**, chiamata **getInstance()**, la quale si occupa di restituire l'unica istanza creata.



Esempio classe Singleton: Fib

La classe **Fib** è un Singleton che mantiene un array di numeri della



sequenza di Fibonacci. Contiene un attributo privato statico `obj` che detiene l'istanza (`private static Fib obj = new Fib();`). Il costruttore è privato, quindi se nel main provassimo a scrivere `Fib f = new Fib();`, otterremmo un errore a tempo di compilazione. Il metodo pubblico statico `getInstance()` si limita a ritornare `obj`. Quando nel codice client richiediamo due volte l'istanza assegnandola alle variabili `f` e `f2`, entrambe le variabili manipoleranno esattamente lo stesso oggetto in memoria e condivideranno il suo stato interno.

```
public class Fib { // Classe Singleton che tiene un array di interi

    // L'unica istanza è tenuta da obj e viene creata subito (Eager initialization)
    private static Fib obj = new Fib();

    // Dati interni della classe
    private int[] x = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144};
    private int i = 3;

    // Il costruttore è privato per impedire "new Fib()" da fuori
    private Fib() {}

    // Metodo statico per ottenere l'unica istanza
    public static Fib getInstance() {
        return obj;
    }

    public int getValue() {
        if (i < 11) {
            i++;
        }
        return x[i-1];
    }

    public void revert() {
        i = 3;
    }
}
```

```
public class MainFib {

    public static void main(String[] args) {

        // Richiede l'istanza di Fib. Non si può usare 'new Fib()'
        Fib f = Fib.getInstance();
        System.out.print("f " + f.getValue());
        System.out.println(" " + f.getValue());

        // Richiede una "nuova" istanza, ma in realtà ottiene sempre la stessa
        Fib f2 = Fib.getInstance();
        System.out.print("f2 " + f2.getValue());
        System.out.println(" " + f2.getValue());
    }
}
```

```
// Se de-commenti la riga sotto, avrai un errore a tempo di compilazione:
// Fib f4 = new Fib();
```

(se vuoi vedere un altro esempio+test guarda la slide 2 L4-Singleton)

Trasformando la nostra classe iniziale **Pagamenti** in un Singleton, impediremmo matematicamente la creazione di istanze multiple a runtime, esattamente come avviene per **Fib**. Si dice in questo caso che la classe implementa (o svolge) il ruolo di Singleton.

L'adozione di questo pattern porta diverse conseguenze a livello di design:

- la classe ha il pieno controllo di come e quando i client accedono al valore della sua sola istanza;
- evita l'esistenza di variabili globali per mantenere l'istanza condivisa;
- permette di controllare il numero di istanze create in un programma, facilmente e in un punto solo.

Un aspetto fondamentale nell'Ingegneria del Software è la **flessibilità**. Usare un Singleton è molto più flessibile rispetto a dichiarare tutte le operazioni e le variabili della classe come **static**.

Se i requisiti dovessero cambiare e l'applicazione non dovesse più usare una singola istanza, ma magari un numero finito di istanze (ad esempio un'istanza per ciascuna cartella letta), l'approccio Singleton renderebbe la transizione facilissima. La variante del pattern che permette un numero finito e controllato di istanze si chiama **Multiton**.

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.

Questo concetto è il fondamento del **Principio delle Conseguenze Locali**, secondo cui *un cambiamento in qualche punto del codice non dovrebbe causare problemi o necessitare modifiche a cascata in altri punti del sistema*.

Riuso delle Classi e l'Ereditarietà

Nello sviluppo software spesso ci si trova ad aver bisogno di classi molto simili tra di loro e fare un copia-incolla e modificare qualche attributo o metodo non è pratico e porterebbe solo a una proliferazione incontrollata di classi e a una mole di **lavoro inutile** per il programmatore. Il riuso del codice per essere efficace deve avvenire in modo semplice e senza dover modificare codice preesistente e funzionante.

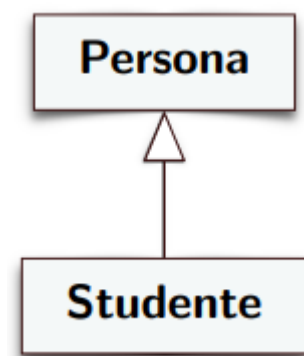
Per fare ciò si fa ricorso all'**ereditarietà**, ossia definire una nuova classe indicando solo **cosa ha in più** rispetto ad una classe esistente.

La sottoclasse eredita tutti i metodi e gli attributi della superclasse come se fossero definiti localmente, aggiunge altri metodi, può ridefinire metodi della superclasse (overriding). L'unico vincolo della sottoclasse è che **non** può ma eliminare metodi o attributi che ha ereditato dalla superclasse.

Ad esempio una classe **Persona** ha **nome** e **cognome** (più vari metodi), una classe **Studente** dovrebbe avere tutto ciò e anche nuovi attributi come **esami**, **voti** etc.

Allora la classe **Studente** eredita da **Persona**, in Java si scrive: `public class Studente extends Persona{}`

Diciamo quindi che **Studente** è una sottoclasse di **Persona** e **Persona** è superclasse di **Studente**.



Record

In Java esiste una classe speciale, record, pensata esplicitamente per contenere dati **immutabili**.

La sua caratteristica principale è una sintassi estremamente concisa ad esempio `public record Esame (String materia, int voto) {}`.

Questa riga fornisce automaticamente un costruttore che assegna tutti i campi e implementa i getter per accedervi. I campi record sono dichiarati **final** e per questo il record non mette a disposizione i setter. Tuttavia però un record può implementare un'interfaccia anche se non può estendere classi: può avere campi static, metodi static e metodi di oggetti.

Visibilità

Quando si strutturano gerarchie di classi è fondamentale gestire la visibilità delle informazioni:

- un elemento dichiarato **private** è visibile e accessibile solo all'interno della classe in cui è stato definito;
- un elemento dichiarato **protected** è visibile all'interno della classe in cui è stato definito e alle sue sottoclassi;
- un elemento dichiarato **public** è visibile a chiunque.

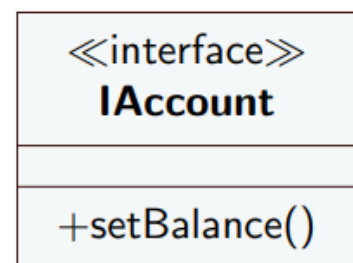
Interfacce e Classi Astratte

Molto spesso è necessario definire un “modello” di comportamento senza specificare subito i dettagli del codice, per questo usiamo le interfacce e classi astratte.

In Java, un’**interfaccia** permette di definire un tipo senza fornire alcuna implementazione per i suoi metodi. Al suo interno, l’interfaccia si limita a elencare le **signature** (firme) dei metodi pubblici senza definire il corpo.

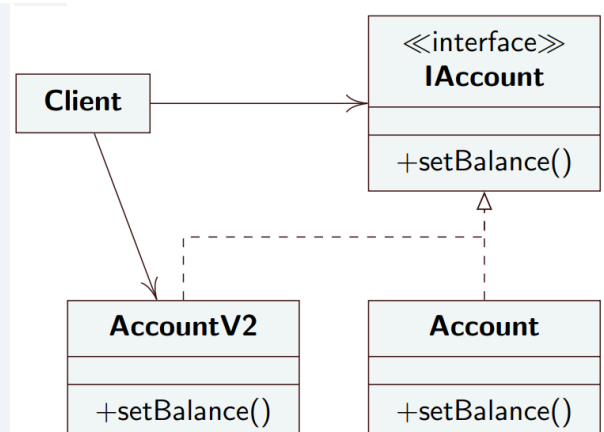
Non può contenere costruttori nè attributi che sono già inizializzati e poiché è priva di implementazione un’interfaccia non può essere direttamente istanziata per creare oggetti. Il suo scopo è essere implementata da una classe che si prende l’impegno di fornire il codice effettivo per metodi che l’interfaccia ha dichiarato.

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



Questo approccio è molto utile per il client che sa esattamente quali operazioni può richiedere attraverso l’interfaccia ignorando i dettagli implementativi. Se l’implementazione dell’interfaccia dovesse cambiare, il codice utilizzato dal client resterebbe invariato.

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

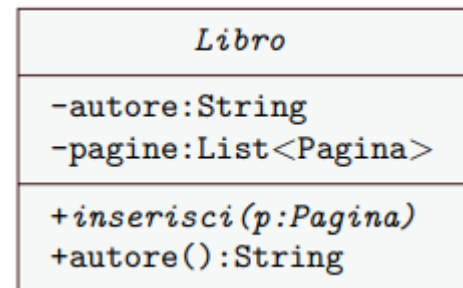


Una **classe astratta** invece si tratta di una classe implementata parzialmente. A differenza dell’interfaccia contiene alcuni metodi regolarmente implementati ma ne possiede altri dichiarati come **abstract** e privi di implementazione.

Anch'essa essendo incompleta non può essere istanziata direttamente in un oggetto.

L'utilità dei metodi astratti risiede nel fatto che gli altri metodi (quelli implementati completamente) possono invocarli, i client stessi si aspettano di poter invocare tali metodi. Inoltre se nella superclasse è definito un metodo **abstract** la sottoclasse è **costretta a implementare** il metodo.

```
public abstract class Libro {  
    private String autore;  
    protected List<Pagina> pagine;  
  
    public abstract void inserisci(Pagina p);  
  
    public String autore() {  
        return autore;  
    }  
}
```



L'ereditarietà stabilisce una forte compatibilità tra tipi: una sottoclasse è considerata a tutti gli effetti un **sottotipo compatibile** con la sua superclasse, tutto ciò che è una superclasse, lo è anche la sottoclasse.

Attenzione:

Questa regola vale solo in un senso, non si può usare la superclasse dove si usava la sottoclasse.

Diagrammi UML

Per capire come gli oggetti interagiscono tra di loro utilizziamo i diagrammi di sequenza UML. In questi diagrammi il tempo scorre lungo un asse verticale orientato verso il basso, nella parte alta posizionati in orizzontale troviamo i vari oggetti coinvolti nel processo. Da ciascun oggetto scende una linea tratteggiata chiamata **lifeline**, sulla quale appare un rettangolo detto **barra di attivazione**. Le interazioni fra oggetti (chiamate a metodo) vengono rappresentate da frecce che partono dalla barra di attivazione del chiamante e arrivano a quella del chiamato.

```
public class MainEsami {  
    public static void main(String[] args) {  
        Studente s = new Studente("Alan", "Rossi", "M12345");  
        s.nuovoEsame("Italiano", 8);  
    }  
}
```

MainEsami

nuovoEsame()

esempio chiamate metodo a cascata

Polimorfismo e Late Binding

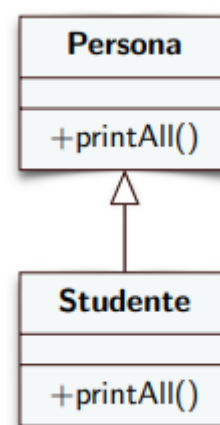
Il polimorfismo è una caratteristica fondamentale nei sistemi orientati ad oggetti. Grazie all'ereditarietà, nei nostri sistemi possono esistere metodi con lo stesso nome e la stessa firma ma definiti in classi diverse.

Questo significa che invocare un metodo può generare effetti diversi a seconda dell'oggetto reale su cui viene chiamato: il comportamento è **polimorfico**.

Senza polimorfismo dovremmo scrivere continui blocchi **switch** nel codice per valutare manualmente a quale classe appartiene ogni istanza e decidere quale metodo chiamare.

Il meccanismo che rende possibile tutto questo in Java si chiama **Late Binding** (o binding dinamico).

```
public void m() {  
    Persona p = new Persona();  
    Studente s = new Studente();  
    Persona px;  
  
    if (i > 10)  
        px = p;  
    else  
        px = s;  
  
    px.printAll();  
}
```



In fase di compilazione, il compilatore sa solo che la variabile **px** è di tipo **Persona**, e si accontenta di verificare che il metodo **printAll()** sia definito in tale classe. Tuttavia, durante l'esecuzione del programma (a runtime), **px** potrebbe puntare all'istanza originaria di **Persona**, oppure all'istanza di **Studente** (se **i <= 10**). È solo in quel preciso momento, a runtime, che la Java Virtual Machine decide quale versione di **printAll()** eseguire in base all'istanza reale. Di conseguenza, **printAll()** invocato su **px** può assumere il comportamento definito nella classe **Persona** o quello ridefinito nella classe **Studente**.

Sottoclassi e Dinamiche di Dispatch

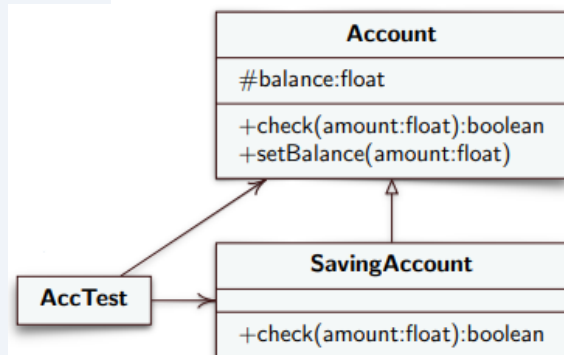
Il dispatch è il meccanismo interno tramite il quale il sistema decide quale implementazione di un metodo deve essere mandata in esecuzione quando quel metodo viene invocato nel codice.

```
public class Account {
    protected float balance = 0;

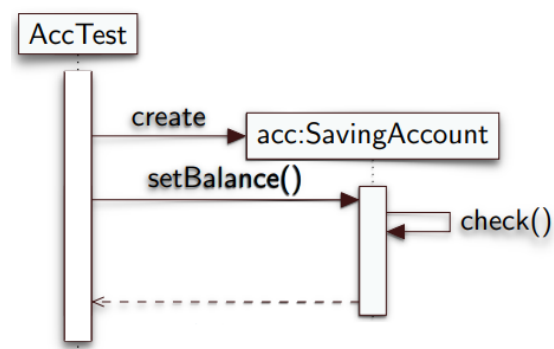
    public void setBalance(float amount) {
        System.out.println("in Account set-balance");
        if (check(amount))
            balance += amount;
    }

    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);
    }
}
```



Cosa succede passo dopo passo quando chiamiamo **setBalance**?

1. **Verifica del tipo:** Quando **setBalance()** viene invocato, il sistema controlla prima di tutto il tipo a runtime dell'oggetto puntato da **acc**, che in questo caso è **SavingAccount**.
2. **Ricerca del metodo:** Il sistema cerca **setBalance()** all'interno di **SavingAccount**. Dato che non lo trova (perché **SavingAccount** non

lo ha ridefinito), risale alla superclasse `Account`, lo trova e inizia a eseguire `Account.setBalance()`.

3. **Il Dispatch interno:** All'interno del codice di `Account.setBalance()`, si incontra la chiamata a un altro metodo: `check()`. Quale versione verrà chiamata?
4. **Ritorno al tipo a runtime:** Anche se ci troviamo "fisicamente" dentro il codice della superclasse `Account`, il sistema ricomincia la ricerca sempre partendo dal tipo reale dell'oggetto a runtime, ovvero `SavingAccount`.
5. **Esecuzione:** Trova il metodo `check()` in `SavingAccount` e lo esegue, ignorando quello di `Account`.

Distinzione tra Tipo di Variabili e Tipi a Runtime

Questo ci porta a distinguere due concetti fondamentali:

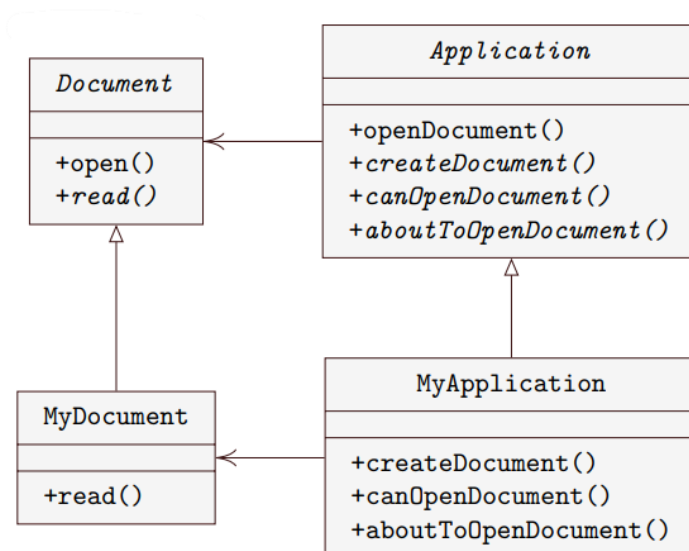
- Tipo a compile-time: è il tipo con cui dichiarare una variabile nel codice (che può essere una classe o un'interfaccia). Questo tipo stabilisce esclusivamente quali metodi e signature il compilatore ti permetterà di chiamare;
- Tipo a run-time: è il tipo concreto dell'oggetto creato in memoria, che rappresenta la sua unica implementazione (e che è sempre una classe, mai un'interfaccia). Il tipo a runtime di un oggetto non cambia mai durante la sua vita.

Possiamo anche usare il **Cast** per forzare l'assegnazione, dichiarando esplicitamente al compilatore di trattare l'oggetto in un certo modo. Ad esempio, scrivere `Studente s = (Studente) p;` va bene per il compilatore e funzionerà anche a runtime, perché l'istanza contenuta in `p` è effettivamente del tipo `Studente`. Se invece provassimo a fare `Studente s = (Studente) new Persona();`, il compilatore lo accetterebbe fidandosi del nostro cast, ma a runtime otterremmo un'eccezione `ClassCastException`, dato che una `Persona` non possiede le caratteristiche necessarie per essere trattata come uno `Studente`.

Design Pattern Template Method

L'intento di questo pattern è quello di definire lo “**scheletro**” di un algoritmo all'interno di un metodo, rinviando però la definizione e l'implementazione di alcuni passaggi specifici alle sottoclassi. Così facendo le sottoclassi possono ridefinire o personalizzare solo alcuni step senza poter modificare la struttura logica e l'ordine generale dell'algoritmo.

È molto utile quando si ha bisogno di classi che definiscono passi necessari per un algoritmo ma non i dettagli di tali passi. Ad esempio una classe astratta **Application** che definisce l'algoritmo per aprire un documento in un metodo chiamato **openDocument()** (che funge da Template Method). Tale algoritmo è definito tramite operazioni astratte che le sottoclassi implementano (override).



Riassumendo possiamo dividerlo nei seguenti passaggi:

- **Lo Scheletro dell'Algoritmo (Il Metodo Template)**: La tecnica di base consiste nel definire l'intera sequenza di un algoritmo all'interno di un singolo metodo della superclasse astratta.
- **Inversione del Controllo (Il Principio Hollywood)**: È la caratteristica architetturale più distintiva. Il flusso di esecuzione non è guidato dal codice "client" della sottoclasse che invoca metodi di libreria, ma è la superclasse astratta che prende il controllo e richiama i metodi specifici della sottoclasse solo nel momento esatto in cui servono.
- **Delega tramite Operazioni Primitive (Metodi Astratti)**: Per i passaggi dell'algoritmo che devono necessariamente variare a seconda del contesto, la superclasse definisce dei metodi astratti.

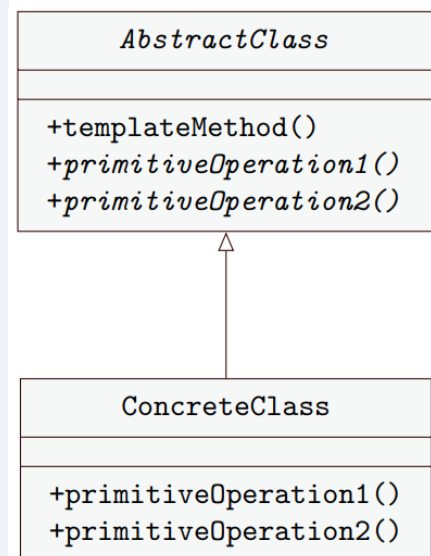
- **Estensibilità tramite Operazioni di Aggancio (Hook Methods):** Sono metodi definiti nella superclasse con un'implementazione di default (spesso vuota o basilare).
- **Riutilizzo del Codice Comune:** Tutti i passaggi dell'algoritmo che sono identici per ogni casistica vengono implementati concretamente una sola volta all'interno della superclasse astratta.

Una `ConcreteClass` estenderà l' `AbstractClass` facendo l'override di `primitiveOperation1()` e `primitiveOperation2()` per svolgere il lavoro specifico.

```
public abstract class AbstractClass {
    public void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
    }
    protected abstract void primitiveOperation1();
    protected abstract void primitiveOperation2();
}

public class ConcreteClass extends AbstractClass {
    @Override
    protected void primitiveOperation1() {
        // codice per l'operazione
    }

    @Override
    protected void primitiveOperation2() {
        // codice per l'operazione
    }
}
```



I template method sono una tecnica eccezionale per il riutilizzo del codice, motivo per cui sono pesantemente utilizzati all'interno di framework e librerie. La loro peculiarità è che generano una struttura di controllo invertita, "**Principio Hollywood**". Le operazioni primitive **devono** essere ridefinite, le operazioni di aggancio **possono** essere ridefinite, nelle sottoclassi.

Design Pattern

Abbiamo parlato del design pattern Singleton e Template Method, ma cos'è un design pattern?

I design pattern sono delle micro-architetture software che coinvolgono un numero ristretto di classi e descrivono soluzioni di successo per problemi di progettazione che si presentano in modo ricorrente. L'obiettivo principale è quello di **riusare** un insieme di classi che costituiscono la soluzione a un problema noto. Questo approccio è utile in fase di progettazione, un momento in cui le conseguenze di determinate scelte architetturali potrebbero non essere del tutto note e in cui si rischia di creare classi difficili da riusare.

Adottare i design pattern porta numerosi vantaggi: aiuta i principianti ad agire con la competenza degli specialisti, evita di "reinventare la ruota" abbattendo i costi di sviluppo e fornisce un **vocabolario comune** che facilita la comprensione del design del software.

Inoltre, i pattern si rivelano uno strumento essenziale per analizzare le proprietà non funzionali di un sistema, come ad esempio l'affidabilità, la sicurezza, la testabilità e la facilità di modifica.

La descrizione formale di un design pattern si articola generalmente in 5 parti fondamentali:

- **nome**: permette di identificare il pattern in una singola parola, favorendo un alto livello di astrazione;
- **intento**: descrive in modo conciso le funzionalità offerte dal pattern;
- **problema (motivazione+applicabilità)**: illustra lo scenario in cui il pattern viene applicato e le condizioni necessarie per il suo utilizzo. All'interno di questa sezione si parla spesso degli obiettivi e vincoli, spesso in contrasto tra loro che caratterizzano il contesto del problema;
- **soluzione**: definisce le classi partecipanti, le loro responsabilità e le reciproche relazioni;
- **conclusione**: analizza i compromessi, i vantaggi, gli svantaggi e i risultati derivanti dall'applicazione del pattern.

I design pattern sono organizzati su un catalogo, conosciuto come **GoF (Gang of Four)** in base al proprio scopo:

- **Creazionali:** riguardano le logiche di creazione delle istanze, *Singleton, Factory Method, Abstract Factory, Builder, Prototype*.
- **Strutturali:** riguardano la composizione di classi e oggetti per formare strutture più grandi, *Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy*.
- **Comportamentali:** si concentrano sull'assegnazione delle responsabilità e sull'incapsulamento, *Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsibility, Command, Interpreter, Memento, Visitor*.

Design Pattern Creazionali

I pattern creazionali servono ad astrarre il processo di creazione degli oggetti, rendendo di fatto un **sistema completamente indipendente** da come i suoi oggetti vengono creati e rappresentati.

Questi pattern diventano cruciali quando i sistemi software si evolvono in modo da basarsi più sulla **composizione di oggetti** anziché sulla classica ereditarietà tra classi. Il loro compito è incapsulare la conoscenza relativa alle classi concrete utilizzate dal sistema nascondendo le logiche di istanziazione.

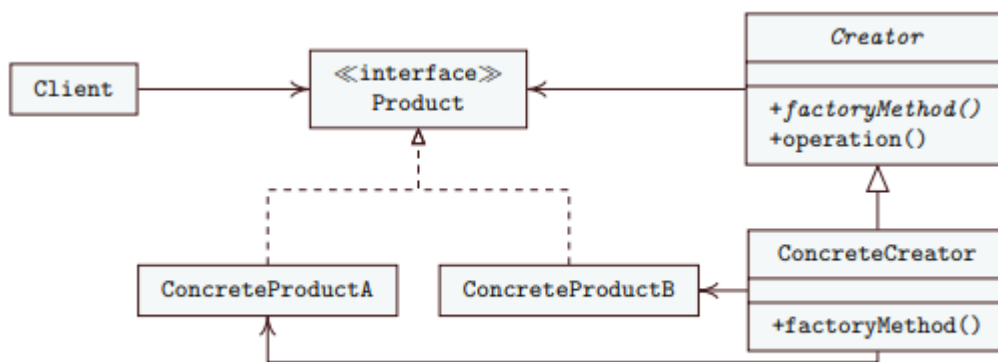
Design Pattern Factory Method

Uno dei pattern creazionali più diffusi è il Factory Method. Il suo intento è quello di definire un'**interfaccia per creare un oggetto** ma lasciare la **decisione** su quale classe concreta istanziare **alle sottoclassi**. In questo modo, il pattern permetta a una classe di rimandare l'effettiva operazione di istanziazione alle sue sottoclassi.

Il problema classi che risolve è quello in cui un framework deve creare e gestire relazioni tra oggetti ma lavorando unicamente con classi astratte o interfacce, non è in grado di istanziarli direttamente. La soluzione consiste in un metodo dedicato alla creazione ("factory") che incapsula la logica per decidere quale classe produrre.

Il Factory Method è diviso in 4 parti:

- **Product:** l'interfaccia comune condivisa da tutti gli oggetti creati;
- **Concrete Product:** l'implementazione specifica e concreta dell'interfaccia Product;
- **Creator:** la classe che dichiara il `factoryMethod()`, il quale restituirà un oggetto di tipo Product;
- **Concrete Creator:** la sottoclasse che implementa (o fa l'override) del `factoryMethod()` scegliendo esattamente quale Concrete Product istanziare e restituire.



L'interfaccia `Studente` svolge il ruolo di Product, mentre le classi `StudenteAttivo` e `StBloccato` sono i Concrete Product. La classe `StCreator` funge da Concrete Creator. Il metodo `getStudente()` decide quale stato dello studente istanziare: se lo studente è attivo, restituisce `new StudenteAttivo()`, altrimenti restituisce `new StBloccato()`.

esempio applicazione
factory method



(troppo lungo per
metterlo qui)

Object Pool

Un object pool è un vero e proprio **deposito** contenente istanze di oggetti già precedentemente create. Quando il client richiede un oggetto, questo viene estratto dal pool e al termine del suo utilizzo, il client ha il compito di restituirlo affinché possa essere riutilizzato. Questo pool può avere dimensioni fisse (se non ci sono oggetti liberi, non se ne possono creare di nuovi) oppure può essere in grado di crescere dinamicamente.

Il **Factory Method** si adatta perfettamente a questa logica:

un Concrete Creator gestisce la richiesta delle istanze. Un esempio pratico lo vediamo nella classe **CreatePool** che funge da **Concrete Creator** e implementa un Object Pool per gestire oggetti di tipo **Shape**. Il suo metodo factory **getShape()** controlla se la lista interna contiene già delle istanze: se sì, ne rimuove e restituisce una preesistente, altrimenti provvede a creare e restituire un nuovo oggetto.

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

public class CreatorPool {
    private List<Shape> pool = new ArrayList<>();

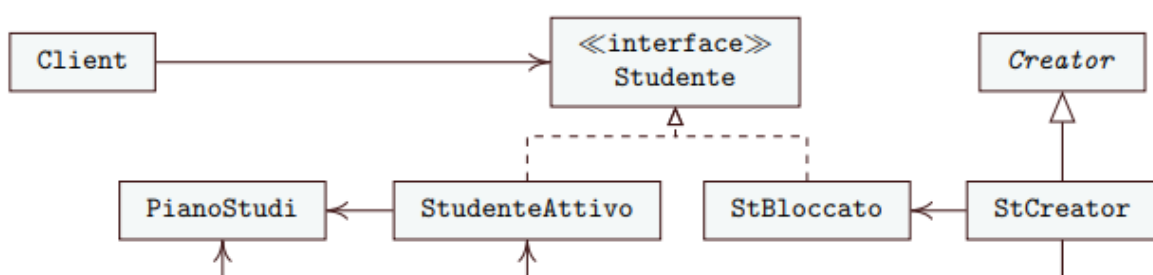
    public Shape getShape() {
        if (pool.size() > 0) {
            return pool.remove(0);
        }
        return new Circle();
    }

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

Dependency Injection

Il Factory Method può anche essere impiegato per **iniettare le dipendenze**, ovvero per passare a un oggetto (il client) gli altri oggetti esterni di cui ha strettamente bisogno per funzionare.

Questa tecnica separa in modo netto la fase in cui un'istanza viene costruita dal momento in cui viene effettivamente utilizzata. Il client **non si occupa** più di creare l'istanza di cui ha bisogno. Al contrario, le dipendenze vengono iniettate direttamente tramite parametri nel suo costruttore. Ciò rende il codice molto **più pulito** e **semplice**, evitando la presenza di metodi setter o di continui controlli per verificare che le dipendenze non siano nulle.



In questo esempio, la classe **Studente** ha bisogno di un oggetto **PianoStudi** per operare. Anzichè istanziarlo internamente, **Studente** lo riceve già pronto nel proprio costruttore e conosce solo il tipo astratto **PianoStudi**, ignorando quali eventuali sottotipi specifici gli stanno venendo passati.

Design Pattern Abstract Factory

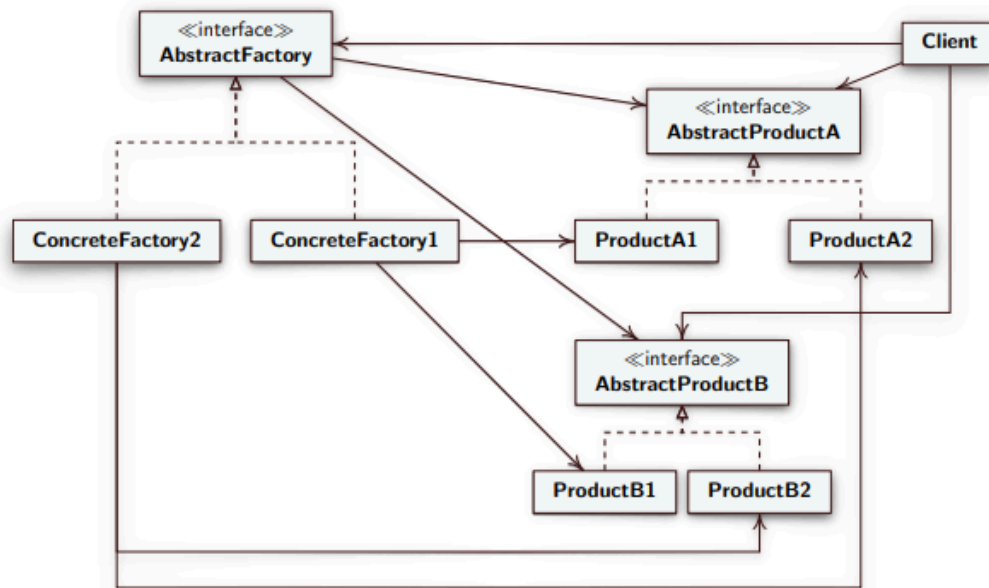
Salendo di complessità troviamo il design pattern Abstract Factory. Il suo intento è quello di fornire un'interfaccia utile a creare **famiglie di oggetti** tra loro correlati o dipendenti, senza doversi preoccupare di specificare le classi concrete.

Questo pattern entra in gioco quando l'intero sistema software deve rimanere indipendente dai dettagli implementativi delle singole classi al fine di poter essere configurato dinamicamente con una fra diverse famiglie di classi disponibili. Lo scopo principale, infatti, è garantire che gli oggetti di una stessa famiglia vengano sempre usati insieme mantenendo **coerenza** e **consistenza**.

Un caso d'uso classico è la gestione dello stato di interfaccia utente (UI): il software potrebbe dover supportare i vari tipi di "look-and-feel" di sistemi operativi differenti, necessitando quindi di generare componenti grafici (widget) semanticamente simili ma visivamente e comportamentalmente diversi per ciascuna piattaforma.

La soluzione astratta dell'Abstract Factory comprende:

- **AbstractFactory**: un'interfaccia astratta che definisce i metodi per creare ciascun prodotto della famiglia;
- **ConcreteFactory**: le implementazioni specifiche dell'interfaccia che creano gli oggetti di una singola famiglia concreta;
- **AbstractProduct**: l'interfaccia comune che definisce una specifica tipologia di oggetto all'interno della famiglia;
- **Product (o ConcreteProduct)**: l'oggetto concreto, creato dal ConcreteFactory che implementa l'AbstractProduct;
- **Client**: usa solo interfacce dichiarate da AbstractFactory e AbstractProduct.



Un grosso vantaggio di questo pattern è la garanzia che gli oggetti vengano impiegati coerentemente e che intere famiglie di classi possano essere scambiate a runtime con grande facilità. Tuttavia, c'è un compromesso significativo: supportare un prodotto concettualmente nuovo è complicato, perché costringe ad aggiungere un nuovo metodo all'interfaccia principale **AbstractFactory** e, di conseguenza, a modificare il codice di tutte le **ConcreteFactory** esistenti.

esempio applicazione
abstract factory



(troppo lungo per
metterlo qui)

Design Pattern Prototype

L'intento del pattern Prototype è quello di specificare gli oggetti da creare utilizzando un'istanza "prototipo" per poi creare i nuovi oggetti semplicemente **copiando questa istanza di base**.

Questo pattern si rivela fondamentale quando la creazione di un nuovo oggetto richiede che questo abbia uno stato iniziale molto simile a quello di un oggetto esistente. Spesso l'oggetto originale ha raggiunto quello specifico stato attraverso una serie di operazioni e ricreare quello stato da zero risulterebbe dispendioso o complesso.

Il pattern risolve un tipico problema architetturale: normalmente quando si crea un oggetto, si deve indicare in modo esplicito la classe da istanziare. Questo genera uno stretto accoppiamento tra la classe che richiede l'istanza (il client) e la classe specifica che viene istanziata. Questo accoppiamento rende l'**architettura rigida** e rende difficile l'aggiunta di nuovi tipi a sistema. Ad esempio con il Factory Method per poter aggiungere nuovi tipi siamo costretti a creare un'intera gerarchia parallela di classi **Creator**.

Per superare questi limiti, il pattern Prototype si basa su 3 partecipanti:

- **Prototype**: un'interfaccia che definisce le operazioni comuni agli oggetti e dichiara il metodo necessario per clonare se stessa;
- **ConcretePrototype**: classe concreta che implementa l'operazione di clonazione definita nell'interfaccia;
- **Client**: classe responsabile di creare nuovi oggetti. Lo fa richiedendo la clonazione dell'istanza prototipo, il che le permette di ottenere il nuovo oggetto senza mai dover indicare la classe esplicitamente. Il client conosce e interagisce unicamente con l'interfaccia Prototype.

Adottare il pattern Prototype porta svariati vantaggi per il design del software:

- **riduzione delle dipendenze**: il client comunica solo attraverso un'interfaccia comune, questo riduce drasticamente l'accoppiamento;
- **flessibilità a runtime**: il client può utilizzare sottoclassi specifiche pur non conoscendone il nome, e il sistema permette di registrare o rimuovere nuovi prototipi dinamicamente durante l'esecuzione (a runtime);
- **meno classi**: si possono definire nuovi oggetti con stati e valori differenti senza creare nuove classi apposite;
- **supporto alla composizione**: un nuovo prototipo può essere utilizzato per mantenere una struttura di oggetti differente; gli

oggetti che compongono la struttura diventano attributi del prototipo. Concettualmente simile al pattern **Composite**;

- **nessuna gerarchia parallela** : non c'è bisogno di implementare una complessa gerarchia di classi **Creator**, come nel Factory Method, perché i nuovi oggetti si ottengono semplicemente per clonazione.

Quando si implementa questo pattern, è bene tenere a mente alcune accortezze tecniche.

L'uso di un **manager di prototipi** (che funge da registro delle istanze) è utile per permettere ai client di immagazzinare e recuperare i prototipi in modo ordinato prima di invocare la clonazione.

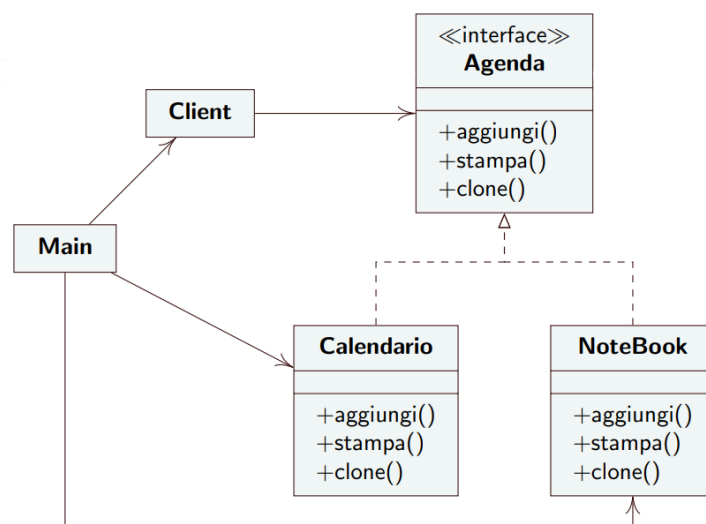
L'implementazione dell'operazione di clonazione può risultare complessa se le strutture dati degli oggetti contengono riferimenti circolari. Molti linguaggi di programmazione forniscono meccanismi automatici di clonazione ma spesso si tratta di una **shallow copy**, ossia una copia superficiale dove i riferimenti agli oggetti contenuti negli attributi vengono condivisi tra il prototipo originale e il suo clone. Non è sempre un comportamento desiderato ed è quindi necessario implementare manualmente una **deep copy**, ossia una copia profonda in modo che gli oggetti interni vengano duplicati e siano del tutto indipendenti.

I cloni potrebbero aver bisogno di essere inizializzati con valori specifici, ragion per cui la classe prototipo potrebbe definire ulteriori operazioni per l'impostazione dei dati.

Immaginiamo di voler gestire un'agenda di impegni e un quaderno di note. Il problema da risolvere è che lo stato dell'agenda è dato dall'insieme degli impegni già presi e, quando viene richiesta una nuova istanza dell'agenda, questa deve poter avere accesso agli elementi già inseriti.

In questo desing:

L'interfaccia **Agenda** svolge il ruolo del **Prototype**. Definisce sia i metodi di business (come **aggiungi** e **stampa**) sia l'operazione **clone()** che restituisce un tipo **Agenda**. Le classi **Calendario** e **NoteBook** agiscono come **Concrete Prototype**. Queste classi



implementano effettivamente i metodi definiti nell'interfaccia, inclusa la logica di clonazione.

La classe **Client** invocherà semplicemente il metodo `clone()` sull'interfaccia per ottenere un nuovo oggetto.

```
public interface Agenda {  
    public void aggiungi(String evento, LocalDateTime data);  
    public void stampa();  
    public Agenda clone();  
}
```

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

osserviamo l'implementazione della **deep copy**: all'interno del metodo `clone()`, non viene restituita la classe originaria o condivisa la lista delle note ma viene creata una nuova `ArrayList` di stringhe (`List<String> n = new ArrayList();`), al cui interno vengono copiati tutti gli elementi della lista originale (`n.addAll(note);`). Successivamente, viene istanziato un

nuovo oggetto **NoteBook** a cui viene assegnata questa nuova lista indipendente, per poi restituirlo al Client (`NoteBook notenew = new Notebook(); notenew.note = n; return notenew;`). In questo modo, l'istanza clonata è totalmente slegata in memoria da quella originaria.

Design Pattern Adapter

L'intento principale del pattern Adapter è quello di convertire l'interfaccia di una classe in un'altra interfaccia che il client si aspetta di utilizzare. In sostanza questo pattern permette a classi diverse di interagire fra di loro risolvendo il **problema delle interfacce incompatibili**.

Molto spesso durante lo sviluppo di un software ci troviamo a dover integrare una classe proveniente da una libreria esterna ma quest'ultima non può essere usata direttamente perché incompatibile con l'interfaccia che la nostra applicazione si aspetta. Questa incompatibilità può manifestarsi in vari modi: il nome dei metodi, il numero o il tipo dei parametri non corrispondono a quelli previsti dalle chiamate all'interno della nostra applicazione.

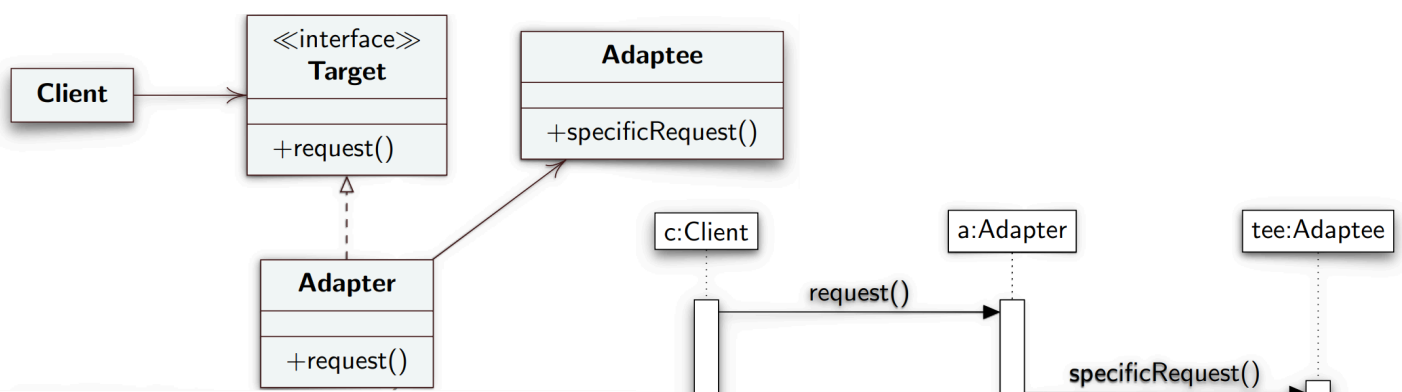
Ci ritroviamo a non poter cambiare l'interfaccia della libreria, poichè non si dispone del codice sorgente, e non è possibile cambiare l'applicazione per adattarla alla libreria poichè vorremmo cambiare il metodo da chiamare in modo trasparente.

La soluzione a questo problema la offre il pattern Adapter introducendo una nuova classe che fa da "**ponte**". Esistono due modi principali per implementare questa soluzione.

Soluzione Object Adapter

In questa versione abbiamo 4 elementi fondamentali:

- **Target**: l'interfaccia che il Client si aspetta di utilizzare;
- **Client**: la classe che usa oggetti conformi all'interfaccia Target;
- **Adaptee**: l'oggetto della libreria esterna che ha un'interfaccia diversa e necessita di essere adattato;
- **Adapter**: la classe "ponte" che adatta le chiamate del Client traducendole nell'interfaccia della classe libreria.

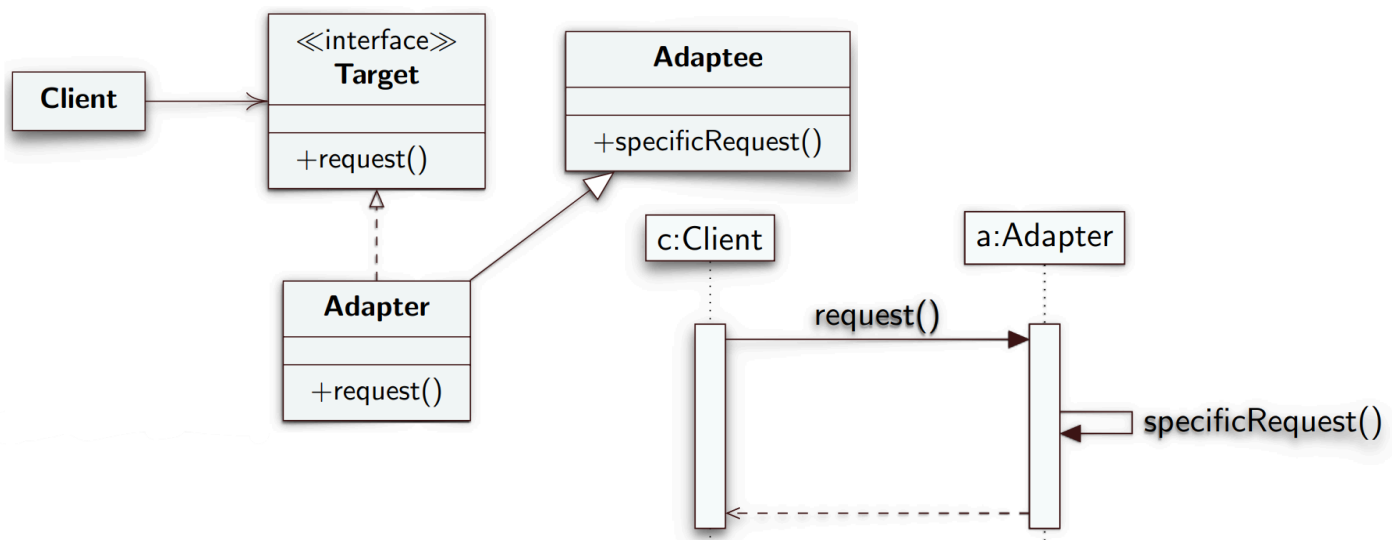


A livello pratico il **Client** usa l'**Adapter** come se fosse l'oggetto nativo che si aspetta. Per far sì che questo funzioni, la classe **Adapter** implementa l'interfaccia **Target**, mantiene un riferimento all'oggetto di libreria (**Adaptee**) e, quando il Client invoca un metodo, l'**Adapter** sa come tradurlo chiamando i metodi corretti dell'**Adaptee**.

Soluzione Class Adapter

Questa è una variante architetturale. Qui i ruoli di **Client**, **Target** e **Adaptee** rimangono invariati rispetto alla soluzione precedente. La differenza risiede in come l'**Adapter** si collega all'**Adaptee**.

La classe Adapter diventa una vera e propria sottoclasse di Adaptee, allo stesso tempo implementa l'interfaccia **Target** e per tradurre le chiamate invoca direttamente i metodi ereditati dalla superclasse.



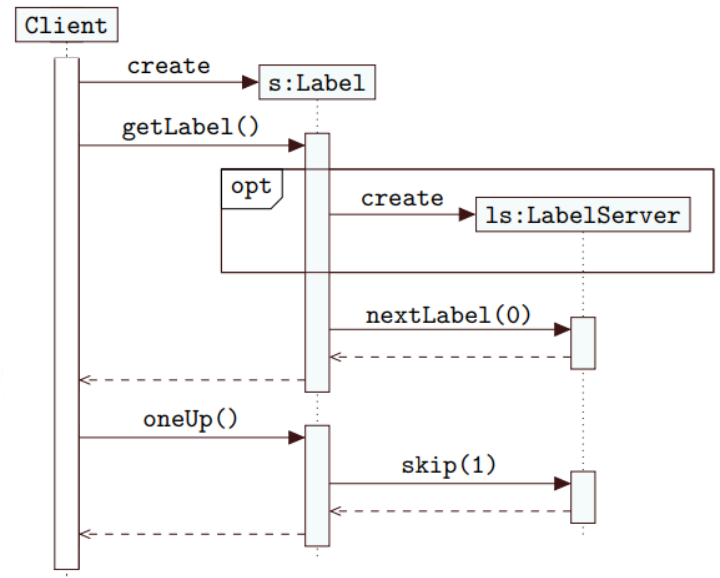
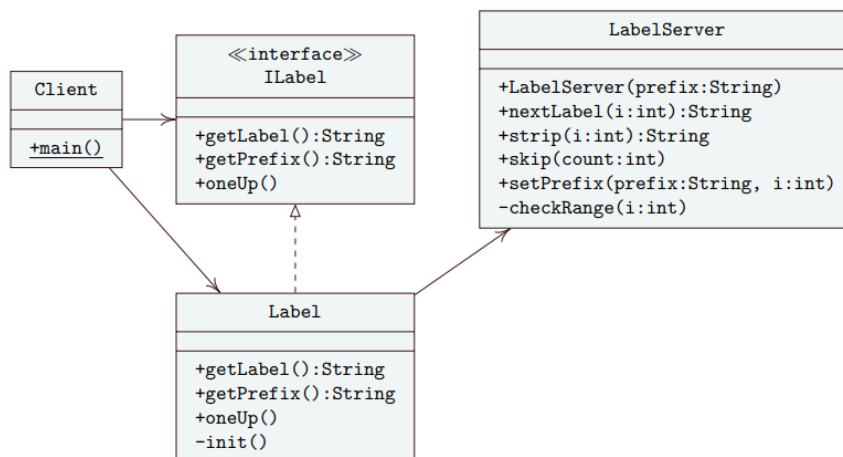
Questa versione rientra nella variante del pattern nota come **Adapter a due vie**. In un Adapter a due vie la classe fornisce contemporaneamente l'interfaccia del **Target** e l'interfaccia dell'**Adaptee**. Essendo una sottoclasse, il pattern Class Adapter espone inevitabilmente i metodi di entrambi.

Utilizzare il pattern Adapter porta a diverse conseguenze progettuali:

- **indipendenza**: il Client e la classe di libreria (Adaptee) rimangono completamente indipendenti;
- **aggiunta di controlli**: il ruolo Adapter può alterare il comportamento o aggiungere test di precondizioni (cosa si deve soddisfare prima di eseguire) e postcondizioni (cosa è verificato se tutto è andato bene);
- **lazy initialization**: l'**Object Adapter** si presta bene a implementare questa tecnica: l'istanza dell'Adaptee può essere creata solo nel

momento in cui avviene effettivamente la prima invocazione di un suo metodo risparmiando memoria se l'oggetto non viene mai usato;

- **overhead delle prestazioni:** di contro, il pattern introduce un livello di indirettezza. Ogni singola invocazione fatta dal Client ne scatena una seconda fatta dall'Adapter. Questo può portare a un rallentamento (seppur trascurabile).



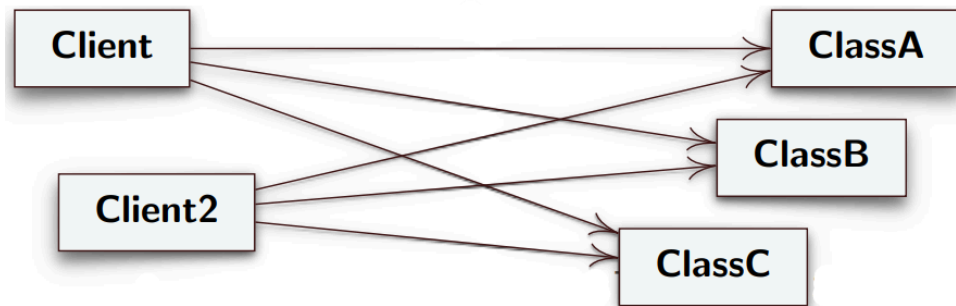
esempio applicazione
adapter (object e class)

Design Pattern Facade

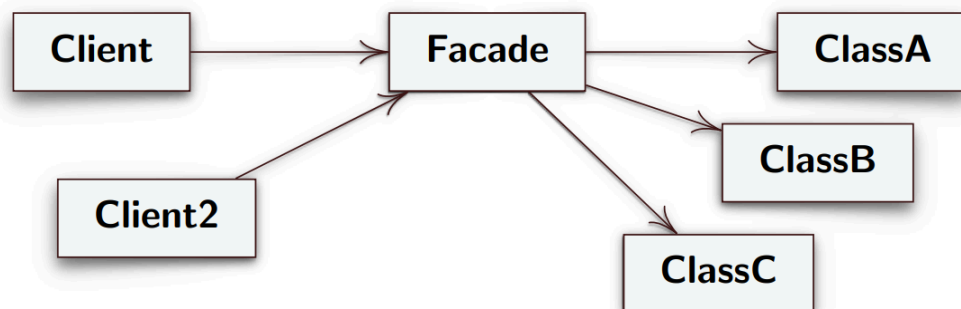
L'intento di questo pattern è quello di fornire un'**interfaccia unificata** che vada a sostituire un insieme di interfacce più piccole e frammentate all'interno di un sottosistema composto da varie classi. L'obiettivo è di definire un'interfaccia di alto livello che risulti molto più semplificata, rendendo l'intero sottosistema più facile da utilizzare.

Spesso nello sviluppo del software ci si ritrova a lavorare con un gran numero di classi che svolgono funzioni strettamente correlate tra loro. In questi scenari l'insieme delle interfacce esposte può diventare complesso da gestire e consultare, soprattutto risulta difficile capire qual è l'interfaccia

essenziale ai client. Si cerca così di **ridurre le comunicazioni e dipendenze dirette** tra i client e il sottosistema.



La soluzione proposta da questo pattern è elegante e diretta: il Facade fornisce un'unica interfaccia semplificata al client **nascondendo gli oggetti** che compongono il sottosistema. Questo approccio riduce la complessità dell'interfaccia esposta e semplifica il codice necessario per le chiamate ai metodi. Nella pratica il Client interagisce solo con l'oggetto Facade. Sarà poi il Facade stesso a occuparsi di smistare la richiesta, invocando internamente i metodi degli oggetti che tiene nascosti.



Adottare il pattern Facade porta con sé diverse conseguenze importanti per l'architettura del sistema. Innanzitutto **nasconde** ai client tutta l'implementazione del sottosistema. **Promuove l'accoppiamento debole** (loose coupling) tra il sottosistema e i client che lo utilizzano.

Nei software di grandi dimensioni questo pattern **riduce le dipendenze di compilazione**. Ciò significa che se viene modificata una classe appartenente al sottosistema, sarà sufficiente ricompilare solo la parte di codice che arriva al Facade, senza dover ricompilare tutti i vari client.

Tuttavia il Facade non è un blocco invalicabile: **non previene** l'uso di client più complessi che, se strettamente necessario, possono comunque accedere direttamente agli oggetti del sottosistema aggirando il Facade.

Dal punto di vista implementativo, se si vuole garantire che gli oggetti del sottosistema risultino **totalmente inaccessibili** al client bisogna annidare

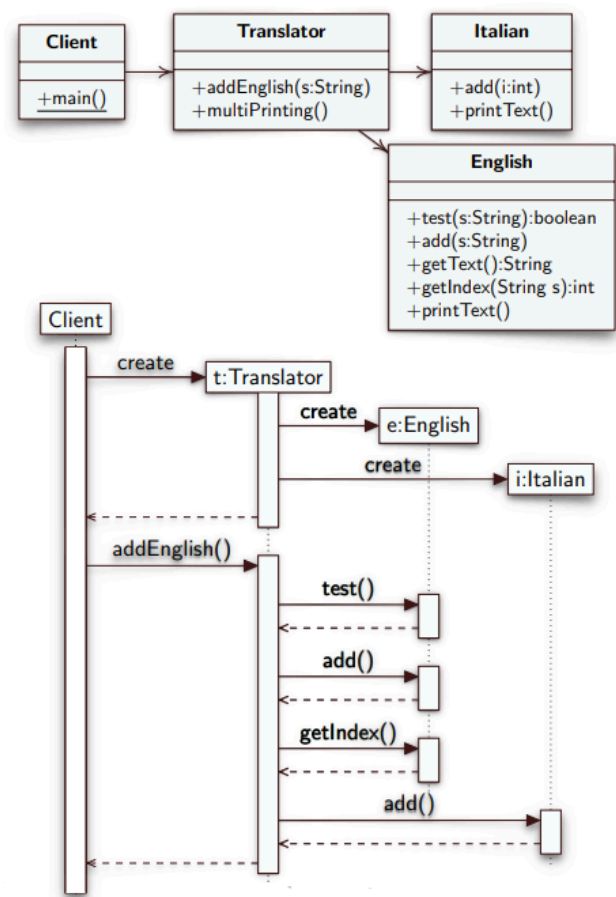
(creare **nested classes**) le classi corrispondenti direttamente all'interno della classe facade.

Immaginiamo di avere una classe **Client** che necessita di inserire e stampare stringhe tradotte. Piuttosto che interagire con le singole logiche delle lingue, il client si affida a una classe **Translator**, che in questo scenario svolge proprio il ruolo di Facade.

Il **Translator** possiede al suo interno, in modo privato e nascosto, le istanze delle classi del sottosistema, ovvero **English** e **Italian**. Quando il client chiama il metodo unificato **t.addEnglish("Hello")**, non deve preoccuparsi di cosa succede dietro le quinte. È il **Translator** (Facade) a farsi carico della logica: invoca il metodo **e.test(s)** sulla classe **English** per verificare la parola, poi usa **e.add(s)** per registrarla. Subito dopo, recupera l'indice della parola inserita con **e.getIndex(s)** e lo usa per aggiungere il corrispettivo tradotto nella classe **Italian** tramite il metodo **i.add()**. Allo stesso modo, se il client vuole stampare tutto chiamando **t.multiPrinting()**, è il Facade che invoca i metodi **printText()** di entrambe le classi interne (**i.printText()** ed **e.printText()**) incapsulando anche la formattazione grafica dell'output ("Italiano: ", "English: ").



esempio applicazione facade



Evoluzione di Java negli anni

I nuovi linguaggi e paradigmi di programmazione nascono con un obiettivo preciso: semplificare il lavoro ai programmatori, nascondendo gran parte della complessità nelle librerie o nei costrutti stessi del linguaggio.

Se osserviamo l'evoluzione di Java nel tempo, notiamo chiaramente questo percorso.

Java 5: uscito nel settembre del 2004, ha introdotto il ciclo **for** avanzato permettendo così di scorrere una lista o un array senza dover fornire il valore di inizio e fine dell'indice .

```
List<String> nomi;  
for (String nome: nomi)
```

Java 7: nel luglio del 2011 Java ha introdotto un'ulteriore semplificazione: l'**inferenza dei tipi**, ovvero evitare di indicare il tipo di elementi nella collection al momento dell'istanziamento.

```
List<String> nomi = new ArrayList();
```

Java 8: a marzo del 2014 avviene una vera rivoluzione per il linguaggio Java, nato come **linguaggio imperativo** ma dopo questa versione furono introdotte caratteristiche tipiche della **programmazione funzionale** che invece è più concisa, espressiva e facile da parallelizzare rispetto alla programmazione ad oggetti.

Classi Anonime e Espressioni Lambda

Il cuore dell'introduzione funzionale in Java sono le espressioni lambda. A livello concettuale, un'espressione lambda non è altro che una **funzione anonima**, ovvero una funzione priva di un identificatore.

È molto simile ad un metodo tradizionale: possiede una lista di parametri in input, un corpo che esegue delle istruzioni e un tipo di ritorno. La sintassi è estremamente **concisa**, il che la rende molto comoda visto che può essere passata direttamente come argomento di un altro metodo.

A livello di sintassi, la lambda è divisa in **due parti** dal simbolo della freccia **->**: a sinistra troviamo i parametri di ingresso e a destra il corpo della funzione. Ad esempio se volessimo implementare una somma di due valori scriveremo **(x, y) -> x + y**, i parametri x e y si trovano a sinistra della freccia, ed è importante notare che il loro **tipo di dato viene determinato automaticamente dal compilatore**. Il corpo a destra esegue l'operazione e

il risultato costituisce automaticamente il valore di ritorno. Se la funzione non accetta alcun parametro, oppure ne accetta più di uno, è obbligatorio racchiudere i parametri tra parentesi tonde. Ad esempio, per stampare semplicemente un saluto **senza argomenti**, useremo la sintassi `() -> System.out.println("Buongiorno")`. Se invece il corpo della funzione è complesso e richiede di eseguire più di un'istruzione, non basta scrivere il codice di seguito alla freccia, ma diventa necessario racchiudere l'intero blocco all'interno delle parentesi graffe. Un esempio è: `(x, y) -> { System.out.println("x: " + x); return x+y; }`.

Nei linguaggi funzionali “puri”, un'espressione lambda rappresenta una funzione pura, il che significa che non mantiene e non altera alcuno stato del sistema e il suo risultato dipende esclusivamente dagli input che riceve, in Java invece le espressioni lambda possono accedere anche ad uno stato esterno ad esse, per questo non possono essere funzioni del tutto pure.

Per capire l'importanza delle espressioni lambda bisogna confrontarle con le **classi anonime** che sono presenti già dalla versione **Java 11**, contro le espressioni lambda introdotte con **Java 8**.

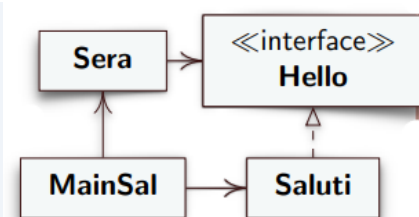
Con una classe anonima si possono implementare interfacce. Prendiamo

```
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();
    }
}
```



un'interfaccia molto semplice chiamata **Hello**, contenente solo il metodo `public void greetings(String s)`. Se avevamo una classe **Sera** il cui costruttore richiedeva in ingresso un oggetto di tipo **Hello**, l'unico modo per istanziarla fornendo il comportamento sul momento era scrivere un blocco di codice molto verboso.

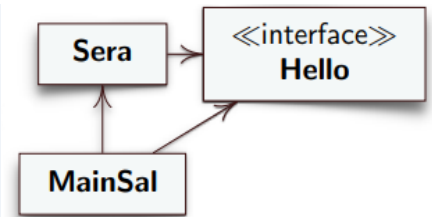
```

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();
    }
}

```



Dopo aver introdotto le classi anonime, non serve più creare il file per la classe **Saluti**: definisci il corpo del metodo direttamente "al volo" dentro l'istruzione `new Hello() { ... }` nel momento esatto in cui istanzi l'oggetto.

Le lambda expression ci permettono di evitare tutto questo "codice standard" o "**boilerplate**", permettendoci di focalizzarci esclusivamente sull'implementazione del comportamento. Questo è possibile quando l'interfaccia possiede un **singolo metodo**.

L'espressione lambda definisce il metodo della classe **Hello** e lo passa direttamente al costruttore di **Sera**. Il compilatore si occupa del resto: intuisce da solo che il parametro **s2** è di tipo **String**, semplicemente guardando la dichiarazione originale del metodo nell'interfaccia.

```

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

```

Differenze tra stile Imperativo, Dichiarativo e Funzionale

```
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");  
    }  
}  
  
// in stile funzionale  
public void trovaFunzionale() {  
    if (listaNomi.stream().anyMatch(nome -> nome.equals("Nobi")))  
        System.out.println("Nobi trovato");  
    else  
        System.out.println("Nobi non trovato");  
}
```

Osserviamo questo esempio e vediamo i vari stili come si comportano:

- **Stile Imperativo:** in questo approccio usiamo un ciclo `for` per ispezionare manualmente la lista, elemento per elemento. Attraverso una variabile booleana teniamo traccia del ritrovamento, una volta che questa variabile diventa `true` usciamo dal ciclo. Sebbene questo stile assicuri al programmatore il **controllo totale** sull'esecuzione, presenta evidenti limitazioni: richiede di scrivere **molte righe di codice per logiche ripetitive** (ricerche, somme, ecc.) e obbliga il lettore del codice a decifrare il corpo del ciclo per comprenderne lo scopo finale. Inoltre, il ciclo è strutturalmente isolato dal codice implementativo della lista, ragion per cui si definisce un'**iterazione esterna**.

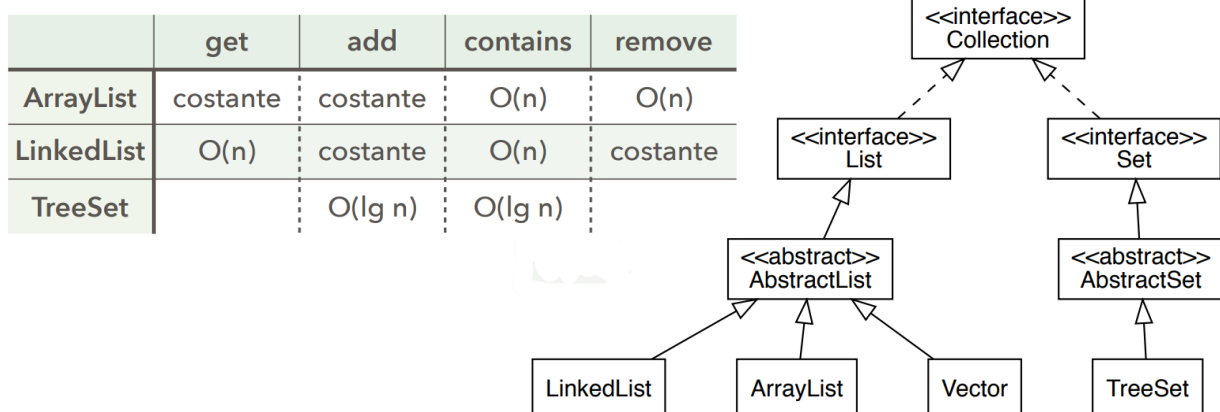
- **Stile Dichiarativo:** questo approccio si concentra più su “come” fare questa ricerca. L'intera operazione si comprime in una semplice istruzione: `if (listaNomi.contains("Nobi"))`. Non abbiamo cicli espliciti o flag: i **dettagli tecnici** su come iterare e verificare la condizione sono opportunamente **incapsulati** e delegati al metodo `contains()` della libreria di ArrayList sottostante. Dal momento che la lista gestisce da sé l'attraversamento degli elementi, parliamo di **iterazione interna**.
- **Stile Funzionale:** questo stile è uno stile dichiarativo ma con alcune aggiunte, come ad esempio le **funzioni di ordine più alto**. Si tratta di particolari funzioni in grado di accettare altre funzioni come parametri di input e grazie alla versione **Java 8** è possibile farlo anche in Java. Quindi, ogni metodo o funzione che abbia tra i parametri o ritorni un'altra funzione viene considerata come funzione di ordine più alto. Ricorda sempre la distinzione formale: in Java, un **metodo** esiste sempre come **componente definita** all'interno di una classe specifica, mentre il concetto di “**funzione**” in sé descrive un'**entità slegata** dall'appartenenza rigida ad una classe.

Java Collections

Le librerie di Java hanno tante interfacce e classi collection. **Collection** è l'interfaccia primaria che definisce i **metodi** `add()`, `remove()`, `size()`, `contains()`, `containsAll()`, etc.

Al di sotto troviamo l'interfaccia **List**, dedicata a collezioni basate su indici, tra i suoi metodi troviamo specificatamente `get()`, `indexOf()` e `set()`.

Tra le implementazioni di queste interfacce troviamo: ArrayList, LinkedList e TreeSet.



Metodi di Default

L'introduzione di Java 8 ha portato una novità architetturale importante: **metodi di default per le interfacce**. Questa funzionalità è nata per permettere l'inclusione di nuovi metodi su interfacce già esistenti senza rompere la compatibilità con le vecchie applicazioni scritte per versioni precedenti di Java. A differenza delle classi astratte, che possono avere metodi implementati e costruttori in grado di agire sullo stato interno, i metodi di default non agiscono sullo stato dell'oggetto.

Un esempio di metodo di default è `stream()` aggiunto all'interfaccia `Collection`. Quando viene invocato questo metodo su una collezione esso restituisce un oggetto di tipo `Stream<T>`, ovvero una sequenza di elementi il cui tipo `T` è definito dalla collection su cui è stato invocato. Attraverso gli `Stream` è possibile eseguire una serie di operazioni sugli elementi della collection passandogli in ingresso delle funzioni specifiche.

Metodo Filter

Una delle operazioni più frequenti che si compie su uno `Stream` è la selezione dei dati tramite il metodo `filter()`.

Prendiamo come esempio una lista di nomi non modificabile, creata usando il metodo statico `of()` introdotto in Java 9. Se volessimo sapere quante volte il valore "Nobi" è presente possiamo scrivere così:

```
List nomi = List.of("Nobita", "Nobi", "Suneo");
long c = nomi.stream().filter(s->s.equals("Nobi")).count();
```

Il metodo `filter(Predicate<T> p)` di `Stream` prende come argomento una funzione chiamata **predicato**, ovvero una funzione che ritorna esclusivamente un booleano, l'espressione lambda `s -> s.equals("Nobi")` è esattamente quel predicato: riceve in ingresso `s` (un elemento dello stream) e restituisce un booleano valutando il metodo `equals()` della classe `String`. Il `filter()` genererà quindi un nuovo stream composto unicamente dagli elementi che soddisfano questa condizione. Queste due righe di codice permettono di evitare di scrivere tutto questo:

```
Stream s1 = nomi.stream();
Stream s2 = s1.filter(s->s.equals("Nobi"));
long c = s2.count();
```

Possiamo usare il filtro per controllare molteplici condizioni, ad esempio per contare quanti elementi hanno 5 caratteri non possiamo usare il metodo `contains()` delle liste, ma possiamo scrivere `.filter(s ->`

`s.length() == 5`), dove su ogni elemento `s` si valuta il metodo `length()` per estrarne la lunghezza. Allo stesso modo potremmo verificare le stringhe vuote passando al filtro `s -> s.isEmpty()`, dove `isEmpty()` è un metodo della classe `String` invocato su ciascun elemento.

```
long c = nomi.stream().filter(s -> s.length() == 5).count();
```

```
long c = nomi.stream().filter(s -> s.isEmpty()).count();
```

È importante fare una distinzione però: `filter()` è un'operazione **intermedia**, detta anche **lazy**, e può essere eseguita in un momento successivo rispetto a dove è inserita. È eseguita quando è necessario, tutte le operazioni che restituiscono un nuovo Stream sono sempre operazioni intermedie. Al contrario, `count()` è un'operazione **terminale**, detta anche **eager**, essa genera un valore finale ed è proprio lei a forzare l'effettiva esecuzione di tutte le operazioni intermedie precedenti.

In Java quando parliamo di **Predicate**, ci stiamo riferendo a una specifica interfaccia funzionale. Un'interfaccia funzionale è per definizione deve contenere **un solo e unico metodo astratto**. Questa regola è così ferrea perché è ciò che permette a Java di supportare le espressioni lambda.

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

In questo caso Java deve sapere esattamente quale metodo associare a questa logica. Se l'interfaccia avesse più metodi, il compilatore non saprebbe quale compilare.

L'annotazione `@FunctionalInterface` messa sopra la dichiarazione dell'interfaccia, indica al compilatore che questa interfaccia ha un solo metodo astratto. Non è obbligatoria per far funzionare le cose ma è una direttiva fondamentale.

Il singolo metodo definito all'interno di `Predicate<T>` si chiama `test(T t)`. Accetta in ingresso un parametro di tipo `Object`, o del tipo generico `T`, e restituisce un booleano.

Quando si invoca il metodo `filter()` di uno Stream, questo metodo si aspetta di ricevere un oggetto di tipo `Predicate`. Il `filter` prenderà poi questo predicato e chiamerà il suo metodo `test()` su ogni singolo elemento dello stream.

Metodo Reduce

Quando lo scopo finale è passare da un insieme di valori disparati a un singolo risultato (come ad esempio una somma) si utilizza il metodo `reduce(T identity, BinaryOperator<T> accumulator)`. È un metodo di Stream che prende un valore di partenza detto identità dello stesso tipo degli elementi nello stream e una lambda che riceve due valori per restituirne uno. se scrivessimo `reduce(0, (accum,v) -> accum +v)`, il sistema applica questa somma accumulata a tutti gli elementi: il parametro `v` rappresenta di volta in volta l'elemento dello stream estratto, mentre `accum` tiene traccia della somma, partendo inizialmente da 0. Essendo anch'essa un'espressione terminale restituisce infine il valore accumulato.

La sintassi con i due punti, detta **riferimento ai metodi**, permette di passare direttamente il riferimento ad un metodo esistente, in sostituzione di una lambda esplicita producendo lo stesso calcolo. La sintassi si ridurrebbe a `reduce(0, Integer::sum)`. `Integer::sum` è il riferimento al metodo statico `sum()` di `Integer` che prende due parametri integer e ne ritorna la somma. Il metodo `reduce()` prende in ingresso una sola espressione lambda e in caso restituisce un tipo `Optional<T>`, il risultato però non esiste se lo stream di partenza è vuoto.

`Optional` rappresenta un valore che può esistere o meno, per verificare il suo contenuto si controlla il metodo `isPresent()` e poi si estrae l'effettivo valore tramite `get()`.

```
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;
    }

    public float calcolaSomma() {
        return importi.stream().reduce(0f, Float::sum);
    }
}
```

Metodo Map

Un'altra operazione importantissima è `map(Function<T, R> mapper)`, che invece di escludere gli elementi come il filter, prende una funzione e la applica per restituire uno stream completamente nuovo contenente i risultati di tale funzione. Ad esempio applicando `map(x -> x*2)` su uno stream di interi, otterremo uno stream finale in cui ogni valore iniziale sarà raddoppiato. `map()` è un'operazione intermedia.

```
public class Persona {
    private String nome;
    private int eta;
    public Persona(String n, int e) {
        nome = n;
        eta = e;
    }
    public String getNome() {
        return nome;
    }
    public int getEta() {
        return eta;
    }
}

// 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);

List<Persona> amici = Arrays.asList(
    new Persona("Saro", 24), new Persona("Taro", 21),
    new Persona("Ian", 19), new Persona("Al", 16));

int somma = amici.stream()
    .map(Persona::getEta)
    .reduce(0, Integer::sum);
```

Consideriamo questa classe `Persona` che ha due proprietà private: un nome (string) e un'età (int). Consideriamo una lista di amici, vogliamo calcolare la somma delle età totali, l'approccio classico imperativo prevede di creare una variabile `somma` aggiornata all'interno di un ciclo estraendo manualmente l'età `x.getEta()` ad ogni iterazione.

L'approccio funzionale invece permette di invocare `amici.stream().map(Persona::getEta).reduce(0, Integer::sum);`. In questo frangente, `map(Persona::getEta)` valuta su ogni istanza il metodo `getEta()` mettendone output l'età per alimentare la somma della `reduce`.

Scrivere in maniera **dichiarativo-funzionale** permette una flessibilità di estrazione di dati che il vecchio stile non consente. Ad esempio se su una lista di stringhe è sufficiente scrivere `nomi.contains("Saro")` per trovare una corrispondenza con collezioni di oggetti complessi (come le istanze `Persona`) lo stream permette di analizzare e concatenare condizioni in sequenza precise. Per cercare ad esempio quanti soggetti si chiamano "Taro" e hanno 21 anni potremmo accodare due operazioni di filtro `.filter(s -> s.getNome().equals("Taro")).filter(s -> s.getEta() == 21)` seguito dall'operazione `count()`.

Tutto questo semplifica problemi complessi, come ad esempio estrarre la persona più vecchia fra quelle che hanno meno di 20 anni. In modo imperativo dovremmo scorrere la lista verificando l'età con un ciclo if, inizializzare una variabile temporanea per il massimo e poi stamparla.

In modo funzionale possiamo vedere due approcci distinti:

- **metodo reduce**: dove `filter()` è usata per separare gli elementi che soddisfano la condizione sull'età, prende in ingresso la funzione (predicato) da eseguire su ciascun elemento mentre `reduce()` è usata per selezionare un elemento, invocando `getMax()` che confronta a due a due.

```
//restituisce l'istanza con l valore massimo d'età
public static Persona getMax(Persona p1, Persona p2){
    if(p1.getEta()>p2.getEta())
        return p1;
    return p2;
}

Optional<Persona> pmax=amici.stream()
                            .filter(x->x.getEta()<20)
                            .reduce(Persona::getMax);

if(pmax.isPresent())
    System.out.println("persona: "+pmax.get().getNome());
```

- **metodo max**: qui `max()` trova il valore massimo, è un'**operazione terminale**, prende un `Comparator`, restituisce un `Optional`, opera in modo simile a `reduce()`. `Comparator` implementa una funzione di confronto (`compare()`).

```
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());
```

Comparator

`Comparator<T>` è un'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)`:

1. Se `o1 == o2` restituisce 0;
2. Se `o1 > o2` restituisce un valore positivo;
3. Se `o1 < o2` restituisce un valore negativo.

```
Comparator<Persona> myComp=(p1,p2)->p1.getEta()-p2.getEta();
Optional<Persona> pmax=amici.stream()
    .filter(x->x.getEta()<20)
    .max(myComp);
```

Metodo Collect

L'ultimo metodo introdotto da Java 8 è `collect()`.

```
Lista<Persona> amici= List.of(new Persona("Saro", 24),
    new Persona("Taro", 21),new Persona("Ian",19),new Persona("Al", 16 ));

//ricaviamo l'età
List<Integer> e = amici.stream().map(x->x.getEta())
    .collect(Collectors.toList());
```

In questo caso `map()` restituisce uno stream con i valori delle età e la funzione passata dice come trasformare ciascun elemento dello stream, tramite `collect()` possiamo raggruppare i risultati prendendo in input un `Collector`.

La classe `Collectors` implementa metodi utili per raggruppamenti come ad esempio il metodo `toList()`, introdotto con Java 16, che accumula elementi in una `List` e restituisce un `Collector`.

Se invece usassimo il classico stile imperativo avremmo dovuto scrivere una cosa del genere:

```
List<Persona> p;
List<Integer> e = new ArrayList<>();
for(Persona x : p)
    e.add(x.getEta());
```

Programmazione Parallela

Oggi i processori sfruttano un'architettura multi-core e la programmazione sequenziale non sfrutta adeguatamente l'hardware in quanto parallelizzare l'esecuzione in stile classico risulta difficile.

In Java la classe `Thread` permette di lanciare un nuovo thread di esecuzione, tuttavia però bisogna risolvere a mano problemi delicati come i **problemi di corsa critica**, usando opportunamente `synchronized`, `wait`, `notify`. Con Java 5 sono stati messi a disposizione Lock ed Esecutori che forniscono un aiuto al programmatore.

Operazioni di elaborazione stream come `map()` e `filter()`, sono definite **stateless**, ovvero durante l'esecuzione non mantengono uno stato, quindi il risultato non dipende dall'ordine su cui i singoli elementi vengono prelevati e lo stream originale non subisce mai modifiche, questo facilita la **parallelizzazione**.

Bisogna però fare attenzione se durante l'esecuzione di un'operazione, ad esempio `map()`, si aggiorna uno stato globale, a quel punto l'operazione non è più stateless.

Per facilitare l'uso del multithreading sulle strutture dati, l'interfaccia `Collection` mette a disposizione due metodi default fondamentali: `stream()` e `parallelStream()`.

Il `parallelStream()` permette di restituire, se possibile, uno stream parallelo. In questo modo si può ottenere un'esecuzione parallela dell'elaborazione senza avere bisogno di dover gestire esplicitamente thread e blocchi di sincronizzazione.

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

Eseguendo su milioni di elementi il seguente codice, le prestazioni migliorano (su hardware con più core) per `ArrayList`, `Vector`, `TreeSet`; migliorano poco (o non

migliorano) quando si usa `LinkedList` a causa dell'accesso sequenziale.

Filtraggio dei Dati con gli Stream

Quando lavoriamo con le collezioni in Java, gli Stream ci permettono di dichiarare cosa vogliamo ottenere dai dati, piuttosto che scrivere cicli complessi su come ottenerlo.

Data una lista di Persone **Persone** vogliamo trovare i nomi di tutti i programmatori con meno di 26 anni e stamparli in ordine alfabetico.

```
List<Persona> team = List.of(
    new Persona("Kent", 29, "CTO"),
    new Persona("Luis", 25, "Programmer"),
    new Persona("Al", 25, "Programmer"),
    new Persona("Andrea", 26, "GroupLeader"),
    new Persona("Joe", 26, "Programmer")
);

team.stream()
    .filter(p -> p.eta() < 26)
    .filter(p -> p.ruolo().equals("Programmer"))
    .sorted(Comparator.comparing(Persona::nome))
    .forEach(p -> System.out.print(p.nome() + " "));
// Output: Al Luis
```

- **filter()**: operazione intermedia che restituisce un nuovo stream con gli elementi che soddisfano il predicato passato;
- **sorted()**: operazione intermedia **stateful** che restituisce uno stream contenente gli elementi ordinati in base al Comparator passato. Grazie a **Comparator.comparing()** possiamo estrarre la chiave (il nome) su cui effettuare il confronto.
- **forEach()**: operazione terminale che esegue un'azione (la stampa) su ciascun elemento dello stream. È importante notare che se usassimo uno stream parallelo, l'ordine di esecuzione del **forEach** non sarebbe garantito.

Data una lista di Persone **Persone** vogliamo trovare i diversi ruoli.

```
team.stream()
    .map(p -> p.ruolo())
    .distinct()
    .forEach(r -> System.out.print(r + " "));
// Output: CTO Programmer GroupLeader
```

In questo esempio vediamo come **distinct()**, che è un'operazione intermedia stateful che restituisce uno stream di elementi distinti, senza quindi elementi ripetuti.

Data una lista di Persone **Persone** vogliamo trovare il nome di una che ha il ruolo Programmer.

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

if (r.isPresent()) System.out.println(r.get().nome());
// Output: Luis
```

Qui vediamo che viene utilizzato **findAny()**, un'operazione terminale (simile a **findFirst()**) che restituisce un oggetto **Optional**. La sua caratteristica principale è quella di essere un'operazione di **short-circuiting**: per essere valutata non necessita di esaminare tutto lo stream, il che significa che blocca l'esecuzione delle operazioni precedenti non appena trova un risultato utile.

Durante la costruzione di una pipeline, è fondamentale distinguere operazioni **stateless** e **stateful**.

Le operazioni come **map()** e **filter()** sono stateless. Questo significa che non mantengono uno stato interno, elaborano un elemento dello stream alla volta, in modo totalmente indipendente dagli altri e forniscono subito un risultato.

Al contrario, operazioni come **sorted()** e **distinct()** sono stateful. Per poter ordinare una sequenza o capire se un elemento è duplicato, l'operazione deve necessariamente conoscere e confrontare altri elementi dello stream prima di poter procedere.

Infine le operazioni terminali come **reduce()** e **max()** accumulano un risultato passo dopo passo. Il risultato in una passata viene reimpresso come input nella passata successiva.

Invece di partire da una lista esistente, è possibile creare stream che generano elemento al volo, potenzialmente **all'infinito**. Java mette a disposizione due metodi principali: `iterate()` e `generate()`. Essendo infiniti, usiamo quasi sempre l'operazione intermedia `limit()` per troncare lo stream ad una determinata lunghezza.

Metodo Iterate

Questo metodo produce uno stream infinito e sequenziale applicando iterativamente una funzione. Prende un elemento iniziale "**seme**" e produce la sequenza, `f(seme)`, `f(f(seme))` e così via. Ogni nuovo valore è calcolato in base al valore precedente.

Questo genererà le potenze di due: 2, 4, 8... fino a 1024, `limit()` tronca la lunghezza a 10.

```
Stream.iterate(2, n -> n * 2)
    .limit(10)
    .forEach(System.out::println);
```

Metodo Generate

A differenza di `iterate()`, `generate()` non applica una funzione al valore precedente. Si appoggia a una funzione di tipo **Supplier** che si limita a fornire un nuovo valore ogni volta che viene richiesto.

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

Debug

Capire cosa succede *esattamente* dentro uno stream può essere complicato. Non possiamo usare `forEach()` nel mezzo della pipeline perché consumerebbe gli elementi e chiuderebbe lo stream. Per ispezionare gli elementi passo dopo passo, usiamo `peek()`.

<pre>List<Integer> numbers=Arrays.asList(2, 3, 4, 5); numbers.stream() .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());</pre>	<pre>Output after map: 19 after map: 20 after filter: 20 after limit: 20 after map: 21 after map: 21 after filter: 22 after limit: 22</pre>
--	---

L'output mostra la natura "lazy" e verticale degli stream. Il primo numero (2) diventa 19, viene stampato dal primo peek ma fallisce il filtro dei numeri pari (si ferma). il numero 3 diventa 20, passa il filtro, passa il limite e arriva alla fine. E così via, elaborando un elemento alla volta attraverso l'intera catena, finché il limite di 3 elementi viene raggiunto.

Primitivi IntStream

Per evitare i costi prestazionali dell'**autoboxing** (conversione automatica tra primitivi e oggetti come `int` in `Integer`), Java offre stream specializzati come `IntStream` che rappresenta direttamente uno stream di valori primitivi `int`.

`rangeClosed(1, 6)` restituisce la sequenza di numeri da 1 a 6 inclusi (con passo 1). Nell'esempio mostrato calcoliamo i quadrati di questi numeri e poi `sum()` li somma tutti insieme (in un altro esempio la somma di 1 a 5 dà come risultato 15).

```
int v = IntStream.rangeClosed(1, 6)
                .map(x -> x * x)
                .sum();
```

Possiamo anche passare agilmente tra stream di oggetti e stream primitivi:

- `mapToInt()`: converte uno Stream di oggetti in un `IntStream`. Ad esempio, per sommare le lunghezze di una lista di parole, possiamo fare `.mapToInt(x -> x.length()).sum()`;
- `boxed()`: fa l'inverso, restituisce uno Stream di `Integer` a partire da un `IntStream`;
- `mapToObj()`: permette di trasformare indici primitivi in oggetti complessi. Se abbiamo un `IntStream.rangeClosed(0, 3)`, possiamo estrarre i primi 4 elementi di una lista usando `.mapToObj(i -> lista.get(i))`.

Interfacce Funzionali

Tutte le espressioni lambda che passiamo ai metodi degli stream implementano delle specifiche Interfacce Funzionali.

L'interfaccia `Function<T, R>` definisce un metodo chiamato `apply()` che prende in input un tipo T e restituisce un risultato di tipo R.

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

In questo caso, il metodo `apply()` riceve una stringa in input e l'implementazione esegue `.length()` restituendo un intero. Questa funzione può essere passata direttamente al metodo `map(stringLength)`

per ricavare la lunghezza delle parole di uno stream da far sommare a `reduce(0, Integer::sum)`.

```
int result=
    Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
        .map(stringLength)
        .reduce(0,Integer::sum);
System.out.println(result);
//Output: 31
```

L'interfaccia `Supplier<T>` è un'interfaccia funzionale con un singolo metodo chiamato `get()` e rappresenta una funzione che **non prende in input alcun parametro** ma restituisce un valore di tipo `T`.

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

Il supplier di per sé non immagazzina il dato (in questo caso una stringa), ma contiene il codice logico per generarlo quando viene invocato il metodo `get()`. Questa interfaccia è quella che il metodo `generate()` di `Stream` prende in input.

Per mettere tutto insieme guarda questo esempio qui:

Data una lista di lavoratori, ognuno con il proprio ruolo, calcolare l'ammontare da pagare a ciascuno per un mese e il totale complessivo

Abbiamo una classe `ListaPersone` che contiene:

- La lista dei lavoratori dipendenti del team.
- Una mappa (`roleCost`) che associa a ogni ruolo aziendale (Programmer, Tester, ecc.) un costo giornaliero.
- Una lista di output (`buste`) che conterrà oggetti `BustaPaga` generati per ogni dipendente.

Il codice si sviluppa nei seguenti passaggi:

1. **Generare i pagamenti** (`genListaPagamenti()`): Si crea uno stream dal team. Si mappa ogni persona in una nuova `BustaPaga`, calcolando lo stipendio moltiplicando la tariffa giornaliera presa dalla mappa per 25 (i giorni lavorativi). I risultati vengono ordinati per nome del dipendente e salvati nella lista con `toList()`.
2. **Calcolare il totale aziendale** (`calcolaSommaBuste()`): Si crea uno stream delle buste paga generate, si mappa lo stream per estrarre solo il totale in denaro di ognuna (`BustaPaga::totale`), e infine si

sommano tutti questi interi con una riduzione: `.reduce(0, Integer::sum)`.

3. **Stampare dipendenti divisi per ruolo** (`stampaRuoliPersone()`): Questo è un uso molto avanzato. Crea uno stream a partire dalle chiavi della mappa `roleCost` (i nomi dei ruoli) e li ordina. Per ogni ruolo, stampa il nome del ruolo, e poi crea uno *stream annidato* all'interno del `forEach` iterando di nuovo su tutto il team per trovare (tramite `filter`) e stampare solo le persone appartenenti a quello specifico ruolo.



Design Pattern Composite

L'intento di questo pattern è molto preciso: comporre oggetti in strutture ad **albero** per rappresentare delle **gerarchie** di parti o del tutto. Il suo scopo principale è permettere al codice del Client di trattare in maniera del tutto uniforme sia i singoli oggetti isolati sia le composizioni complesse di oggetti.

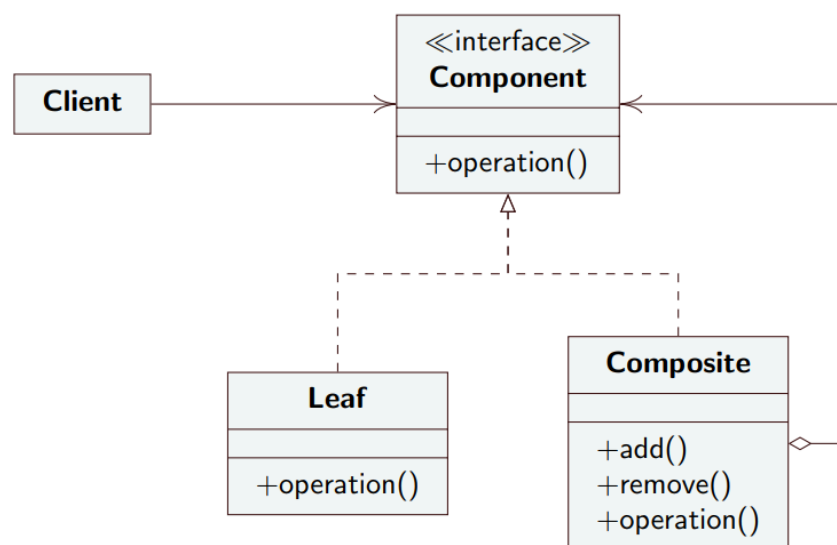
Nello sviluppo di un software, spesso è necessario **raggruppare elementi** semplici fra di loro per formare elementi sempre più grandi e complessi. A livello di codice se facessimo una distinzione tra le classi che rappresentano gli elementi semplici e le classi che fanno da contenitori per questi elementi il codice del client risulterebbe molto complicato. Il pattern Composite risolve il problema descrivendo una **composizione ricorsiva** in modo che il client non debba fare alcuna distinzione tra vari tipi di elementi ma, al contrario, tratterà tutti gli oggetti della struttura in maniera **uniforme**.

Ad esempio tutte le operazioni che eseguiamo sul disco rigido ci permettono di gestire sia i singoli file che le cartelle. Un client del file system esegue operazioni standard come la creazione, la lettura, la scrittura o l'esecuzione. Grazie al pattern, il client invoca queste operazioni

senza dover distinguere se sta lavorando su un file o su una cartella. Inoltre, rispetta la gerarchia "parte-tutto": una cartella può contenere sia file singoli che altre cartelle, le quali a loro volta contengono altri elementi, ricorsivamente.

Per ottenere questo risultato, la struttura del pattern si basa su 3 elementi:

- **Component:** l'interfaccia che rappresenta sia gli elementi semplici che quelli complessi. Implementa le operazioni comuni alle sottoclassi e spesso dichiara le operazioni per accedere e gestire gli elementi "figli" (come aggiungerli o rimuoverli). Può anche definire un'operazione per permettere a un elemento di accedere al proprio elemento "padre" nella gerarchia;
- **Leaf:** la classe che rappresenta gli elementi semplici veri e propri (detti child/children). È una sottoclasse di Component e implementa il comportamento reale per le operazioni base;
- **Composite:** la classe che rappresenta gli elementi contenitori. È anch'essa sottoclasse di Component e la sua peculiarità è che tiene il riferimento di ciascuno dei suoi elementi child e implementa concretamente le operazioni per gestirli. Definisce inoltre il comportamento generale dell'intero aggregato.



Ci sono 3 fasi da distinguere:

- **Associazione:** rappresenta una relazione in cui una classe usa un'altra classe.
- **Aggregazione:** rappresenta una relazione in cui una classe (whole) consiste di altre classi (parti). Ovvero, il Composite aggrega (e consiste di) Leaf (parti). Chi aggrega e le parti esistono separatamente l'uno dall'altro.

- **Composizione:** una forma di aggregazione in cui la parte non può esistere senza l'intero (whole). La parte appartiene all'intero ed è distrutta quando l'intero è distrutto.

Il meccanismo su cui si basa il pattern è molto elegante: i client usano solo l'interfaccia **Component** per interagire con la struttura. Se il ricevente della chiamata è **Leaf**, la richiesta è gestita direttamente. Se il ricevente è un **Composite**, questo invia la richiesta ai suoi child e possibilmente avvia operazioni aggiuntive prima e dopo.

Le conseguenze di questo meccanismo sono ottime a livello di **pulizia del codice**. Si formano composizioni ricorsive all'infinito, il client che si aspetta un elemento semplice può riceverne uno composto senza che il programma si rompa e i client rimangono semplici da scrivere perché non devono sapere con cosa stanno interagendo.

Uno **svantaggio** è che a design time non è possibile vincolare un **Composite** a contenere solo determinati tipi di **Leaf** ma andranno fatti dei controlli a runtime.

Per facilitare la navigazione degli oggetti, gli elementi child mantengono il riferimento all'oggetto che li contiene. Il riferimento è inserito in **Component**, mentre **Leaf** e **Composite** possono implementare le operazioni per gestirlo. Per avere client che non distinguono se trattano con **Leaf** o **Composite** (è uno degli obiettivi), la classe **Component** dovrebbe definire quante più operazioni in comune possibile. Le classi **Leaf** e **Composite** faranno **override** delle operazioni.

Sorge però una domanda: dove dichiarare i metodi per gestire i figli, ovvero **add()** e **remove()**?

- **Trasparenza:** se dichiariamo **add()** e **remove()** nell'interfaccia **Component**, tutti i client vedranno la stessa interfaccia, peccando però in sicurezza. Tali operazioni su una **Leaf** (che non può avere figli) non hanno significato. La sicurezza viene compromessa perché un client potrebbe sbagliarsi e avviare **add()** su una foglia. Se il codice si limita a ignorare la richiesta, potrebbe mascherare un grave difetto logico del programma.
- **Sicurezza:** se dichiariamo queste operazioni solo nella classe **Composite**, a compiletime il compilatore ci bloccherà se proviamo ad aggiungere un figlio a una **Leaf**. È più sicuro, ma perdiamo la trasparenza, perché il client ora deve distinguere i tipi. Ad esempio, non potremmo usare un metodo **getComposite()** definito nel

Component (che ritorna **null** di base e **this** nel Composite) per aggirare il problema, perché costringerebbe comunque il client a distinguere il tipo di risultato, azzerando la trasparenza.

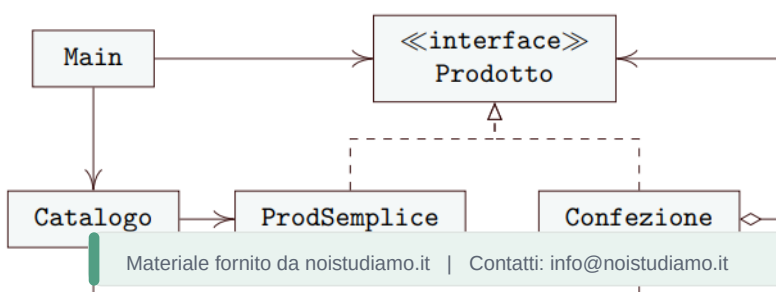
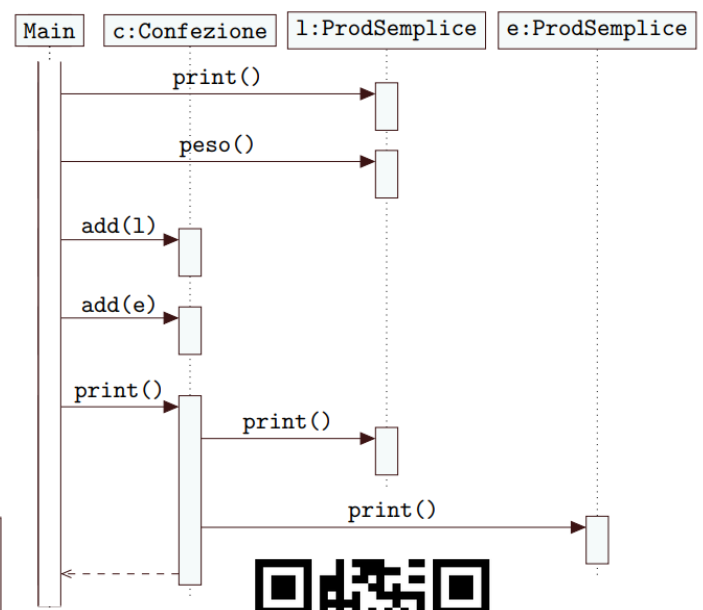
La lista contenente gli elementi figli **deve** essere definita nella classe **Composite**. Se fosse in **Component**, ogni singola foglia inizierebbe una lista vuota che non userebbe mai, sprecando memoria.

Per alcune implementazioni l'ordine dei child nel Composite è importante e va tenuto in considerazione su certe implementazioni.

Se un'operazione nel Composite richiede una ricerca lunga tra figli, si può utilizzare una **cache** per ottimizzare le prestazioni. In questo caso però i componenti che devono avere un metodo per invalidare la cache del padre se il loro stato interno cambia, spesso per fare ciò i child mantengono un riferimento all'oggetto padre che li contiene, posizionato nel **Component**.

Immaginiamo di dover gestire un Prodotto singolo e di una Confezione (insieme di prodotti):

1. **Component (Prodotto)**: l'interfaccia base espone le operazioni **print()**, **descrizione()**, **prezzo()** e **peso()**.
2. **Leaf (ProdSemplice)**: un semplice record (o classe) che implementa **Prodotto**. Ha i suoi campi basilari e, quando chiami il suo metodo **print()**, fa semplicemente una stampa a terminale della sua descrizione e del suo prezzo.
3. **Composite (Confezione)**: È una classe che implementa **Prodotto**, ma internamente possiede una **List<Prodotto>**. Invece di avere un proprio peso o una propria funzione di stampa isolata, delega tutto ai suoi figli. Nel suo metodo **print()**, esegue un iterazione chiamando il metodo **print()** di ogni singolo prodotto contenuto nella lista. Nel metodo **peso()**, somma i pesi di tutti gli elementi usando lo stream. In più, questa classe definisce i metodi **add** e **remove** per manipolare la sua lista interna, aggiornando contestualmente anche il suo **valoreTotale**.



Design Pattern Decorator

Parliamo di un pattern strutturale il cui intento è quello di **aggiungere ulteriori responsabilità** a un oggetto in modo dinamico. Questo pattern fornisce un'alternativa flessibile all'implementazione di sottoclassi per estendere le funzionalità di un sistema.

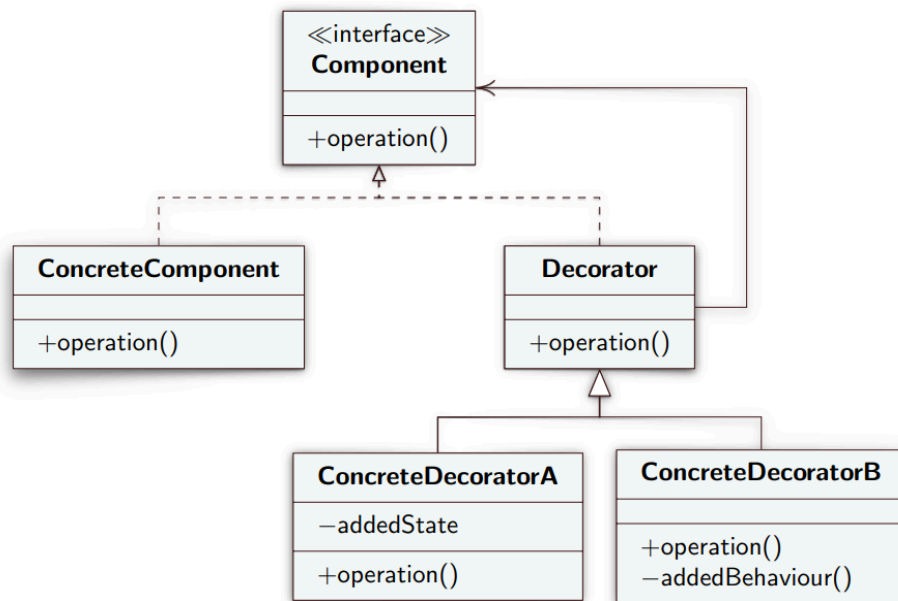
Capita spesso di voler aggiungere responsabilità o funzionalità a singoli oggetti e non alla classe intera. L'idea iniziale potrebbe essere sfruttare l'ereditarietà, tuttavia questo approccio porta a una forte mancanza di flessibilità: un client non potrebbe controllare in modo dinamico come e quando *decorare* un componente.

In scenari più complessi la creazione di sottoclassi diventa non praticabile, un numero grande di estensioni produrrebbe un numero enorme di sottoclassi per gestire tutte le combinazioni. L'approccio più flessibile consiste nel racchiudere (**wrap**) il componente all'interno di un altro oggetto che si fa carico di aggiungere la nuova responsabilità. Questo oggetto wrapper è il **Decorator**.

Poiché il Decorator implementa e risulta **conforme alla stessa interfaccia** del componente che sta decorando, la sua presenza rimane **completamente trasparente** agli oggetti che utilizzano quel componente. Grazie a questo incapsulamento, le responsabilità possono essere non solo aggiunte, ma anche sottratte dinamicamente. Inoltre i decoratori possono essere annidati l'uno dentro l'altro ricorsivamente, permettendo di combinare molteplici responsabilità a runtime.

Questo pattern è composto da alcuni elementi chiave:

- **Component**: l'interfaccia che definisce le operazioni per gli oggetti a cui possono essere aggiunte responsabilità dinamicamente;
- **ConcreteComponent**: definisce un oggetto reale, implementazione di **Component**, su cui verranno aggiunte le responsabilità;
- **Decorator**: mantiene un riferimento a un oggetto di tipo **Component** e definisce un'interfaccia conforme a quella di **Component** stesso. Il suo compito principale è inoltrare le richieste al suo oggetto **Component** interno, mantenendo la possibilità di eseguire altre operazioni;
- **ConcreteDecorator**: l'implementazione pratica del decoratore che aggiunge dinamicamente il nuovo stato o il nuovo comportamento al componente.



A livello di codice il decoratore accetta nel suo costruttore un oggetto di tipo **Component** e lo salva in una variabile interna. Quando viene chiamato un metodo come `operation()`, il decoratore richiama `operation()` sull'oggetto interno, aggiungendovi il proprio comportamento personalizzato. L'istanziazione tipica di questo pattern assume una forma annidata, come ad esempio l'ultima riga del codice qui di fianco.

```

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();
        // ...
    }
}
  
```

L'utilizzo del pattern Decorator porta con sé diversi vantaggi e alcuni compromessi da tenere a mente:

- **massima flessibilità:** si ottiene molta più flessibilità rispetto all'ereditarietà statica, in quanto le responsabilità si aggiungono dinamicamente a tempo di esecuzione;
- **ripetitività:** è possibile aggiungere la medesima responsabilità più volte allo stesso componente, semplicemente wrappandolo con due istanze dello stesso **ConcreteDecorator**;
- **gerarchie snelle:** evita di dover sovraccaricare le classi al vertice della gerarchia con innumerevoli metodi per prevedere ogni possibile responsabilità futura. I **ConcreteDecorator** nascono per essere moduli indipendenti, permettendo di integrare nuove responsabilità in momenti successivi senza toccare il codice base;
- **attenzione all'identità degli oggetti:** l'identità dell'oggetto **ConcreteDecorator** non coincide con quella del **ConcreteCreator** originale. Di conseguenza a livello di programmazione, non si dovrebbero mai confrontare direttamente i loro riferimenti aspettandosi che coincidano;
- **complessità a runtime:** il sistema finale sarà popolato da una miriade di piccoli oggetti simili tra loro, che non differiscono tanto per la classe di appartenenza, ma per il modo in cui sono interconnessi a runtime. Questo può rendere il debugging leggermente più insidioso, pur mantenendo un codice molto pulito e disaccoppiato.

Progettazione Classi e Modelli Dinamici UML

Prima di buttare giù anche solo una riga di codice, bisogna capire come dare una struttura logica al programma e come rappresentarne i comportamenti in maniera chiara. Vediamo ora come passare dall'idea al design vero e proprio **schematizzando** le classi e i flussi di sistema.

Individuare le Classi

Quando iniziamo la progettazione di un software, il primo passo fondamentale è partire dal testo che descrive i requisiti ed effettuare un'attenta analisi grammaticale. La regola da seguire è semplice:

- i **sostantivi** suggeriscono potenziali **classi** o i loro **attributi**;
- i **verbi** indicano le **operazioni**, ovvero i metodi che la classe dovrà avere.

Inoltre leggendo i requisiti, è molto utile individuare gli oggetti fisici menzionati, poiché spesso si traducono direttamente in classi del sistema.

Durante questa fase è cruciale fare attenzione a come raggruppiamo attributi e operazioni all'interno delle classi. L'obiettivo è garantire coesione e rispettare il principio di singola responsabilità (KISS, SRP).

Un Esempio Pratico sui Requisiti

Immaginiamo di avere questi frammenti di requisiti:

- "dovrà essere possibile **cercare** un **cliente** e avere **mostrati** i suoi **dati anagrafici**".
- "la **scheda del cliente** dovrà **mostrare** tutti i **dati anagrafici** e una **lista** dei suoi **fornitori**".
- "dovrà essere possibile **selezionare** un intervallo di tempo e **calcolare** l'importo complessivo degli **ordini effettuati** da un **cliente** in tale intervallo".
- "per ciascun **ordine** dovranno essere **mostrati**: **nome fornitore**, **nome cliente**, **linea di appartenenza** dei **prodotti**, **importo complessivo**".
- "il **report mensile** dovrà **mostrare** per ciascun **cliente**: la sua **provincia** e il **totale** per ciascun **fornitore**".

Da questo piccolo testo possiamo estrapolare:

- **classi**: **Cliente**, **Fornitore**, **Ordine**, **Prodotto**, **Report Mensile**, e **SchedaCliente**;
- **attributi**: dati anagrafici del cliente, nome del cliente, provincia del cliente, linea di appartenenza dei prodotti, importo dell'ordine e nome del fornitore;
- **metodi**: cercare un cliente , mostrare dati anagrafici e fornitori per un cliente , calcolare il totale degli ordini per quel cliente , selezionare gli ordini all'interno di un certo intervallo temporale e calcolare il totale degli ordini per un cliente per ciascun fornitore mensilmente.

È utile suddividerli per colore in modo da risaltare il ruolo di ogni elemento.

Attenzione però: l'analisi grammaticale è utile, ma da sola non basta. Ad esempio, non ha evidenziato in automatico classi fondamentali di "supporto" come **Persona**, **Rubrica**, **ElencoOrdini** e **Stato**; è il progettista che deve capire, analizzando la logica, che queste classi servono al sistema e come si sviluppano i vari metodi.

Diagrammi UML

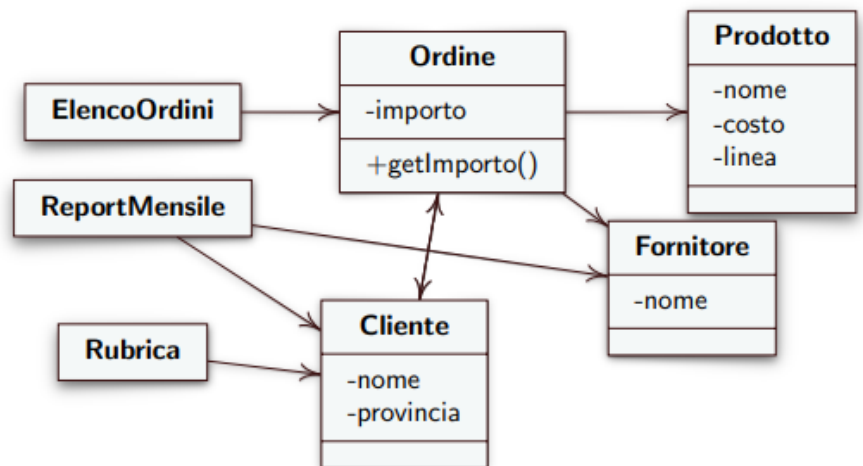
Un diagramma UML (Unified Modeling Language) è una rappresentazione grafica standardizzata per visualizzare, specificare, progettare e documentare la struttura e il comportamento di un sistema.

Esistono diversi tipi di diagrammi, ognuno descrive un aspetto specifico del sistema.

Diagramma delle Classi

Questo diagramma è considerato il cuore vero e proprio della progettazione orientata ad oggetti. Rappresenta i componenti del software: attributi, metodi e classi presenti, tutti disposti come una mappa strutturale dello scheletro del software.

- Il **nome della classe** è collocato in alto;
- Gli **attributi** sono messi subito sotto, accompagnati dalla loro visibilità: -privato +pubblico #protected. A volte sono anche seguiti dal tipo;
- I **metodi** sono distinguibili perché sono presenti le parentesi tonde.



Le interazioni principali che andiamo a definire sono:

- **L'Associazione:** indica un collegamento semantico generico tra due classi. Ad esempio, indica che una **Rubrica** interagisce con la classe **Persona**.
- **L'Aggregazione e la Composizione:** esprimono il concetto di "tutto/parte" tra gli oggetti. L'aggregazione è un legame più debole (un **Ordine** aggrega dei **Prodotti**, ma i prodotti esistono indipendentemente dall'ordine). La composizione, invece, è un legame vitale fortissimo: le "parti" non possono esistere senza il "tutto" che le contiene.
- **L'Ereditarietà (o Generalizzazione):** ci permette di astrarre i concetti creando una gerarchia. Nel nostro sistema, ad esempio, le classi **Fornitore** e **Cliente** sono specializzazioni che ereditano attributi e metodi da una classe padre più generale chiamata **Persona**.

Infine, un dettaglio fondamentale che aggiungiamo sulle linee di associazione è la **molteplicità** (espressa con notazioni come **1..*** oppure **0..1**). Questa ci dice numericamente quanti oggetti di una classe sono legati agli oggetti dell'altra. Sapere se "un cliente può avere zero o molti ordini" o se "un ordine appartiene sempre a uno e un solo cliente" è cruciale. In sintesi, quando questo diagramma è progettato in modo

rigoroso, l'ossatura del codice è praticamente già pronta per essere scritta

```
import java.util.Date;
public class Ordine {
    private int id;
    private Date data;
    private int importo;
    private boolean archiviato;
    private Cliente cli;
    private Fornitore forn;
    private Stato[] s;
    private Prodotto[] prod;

    public boolean verifica() {
        // implementazione
    }
    public int getImporto() {
        // implementazione
    }
}
```

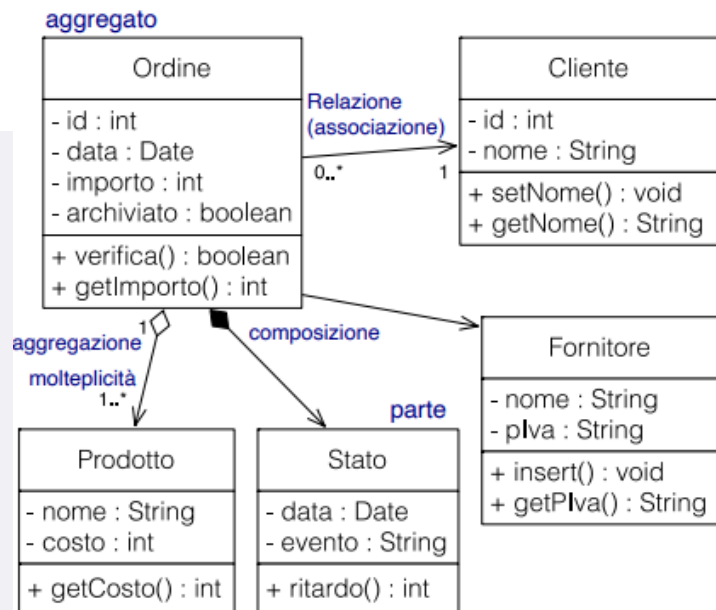


Diagramma di Collaborazione

Mentre il diagramma delle classi mostra la struttura statica del codice, il diagramma di collaborazione mostra le **interazioni dinamiche** fra le istanze a runtime.

In questo diagramma il flusso di messaggi è indicato visivamente da frecce posizionate accanto alle linee di associazione fra le istanze che stanno interagendo. Queste frecce possiedono delle etichette che indicano:

- un **numero sequenziale** che stabilisce l'**ordine** temporale con cui avvengono le interazioni.
- il metodo che viene **chiamato**.
- se occorre un **valore di ritorno**.

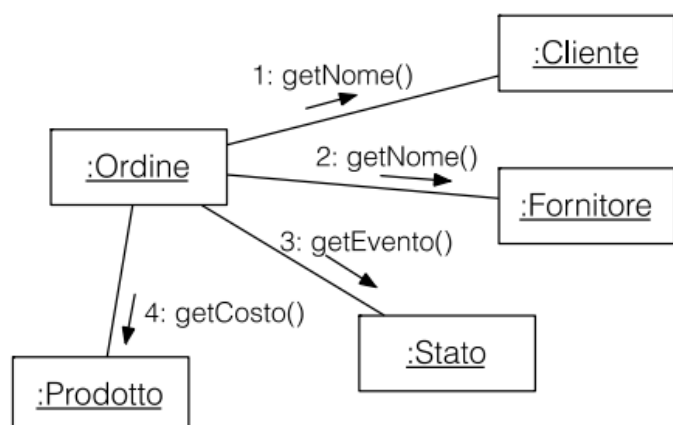
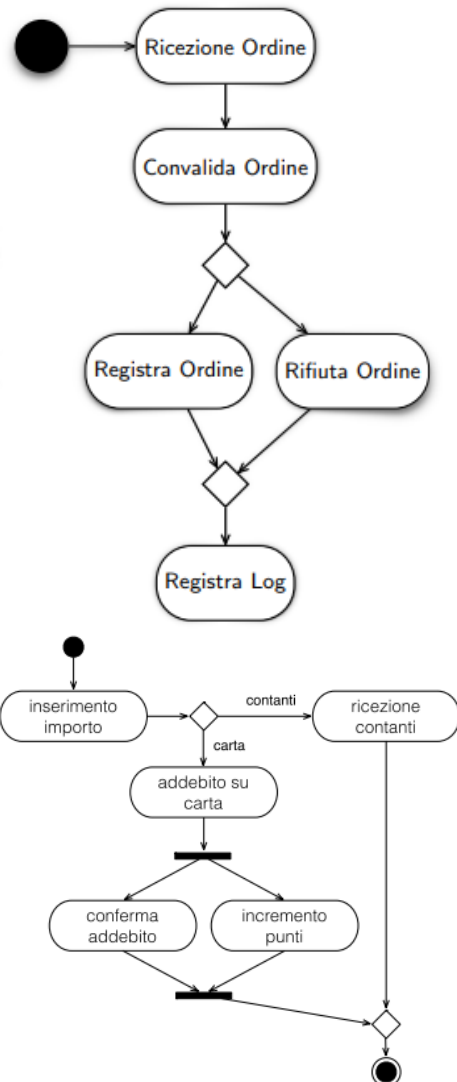


Diagramma delle Attività

I diagrammi UML delle attività rappresentano il punto iniziale della progettazione vera e propria. Sono usati per riflettere primariamente sui processi e le attività che il software dovrà realizzare ma non mostra quali oggetti svolgono le attività.

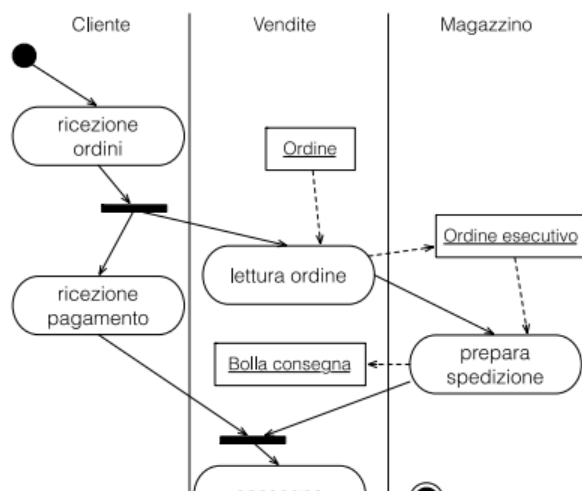
La sintassi grafica prevede:

- Un cerchio pieno per indicare lo **stato iniziale** del processo.
- Rettangoli arrotondati per indicare i vari passi di elaborazione.
- Frecce continue per indicare i **flussi tra un'attività e quella successiva**.
- Rombi per rappresentare diramazioni condizionali o punti di unione.
- Quando si verificano **flussi paralleli** si utilizzano delle barre: una barra con una freccia entrante e due uscenti indica un punto di **parallelizzazione** del processo, mentre il contrario un punto di **sincronizzazione** in cui i flussi si riuniscono. Infine un cerchio pieno circondato da una circonferenza esterna indica lo **stato finale**.



Diagrammi con Corsie (Swimlanes)

Per superare la limitazione dei diagrammi delle attività viste prima, che



non specificano chi fa cosa, si introducono le corsie, conosciute anche come swimlane. Queste partizioni indicano chiaramente quale **responsabile deve farsi carico delle attività** contenute in una determinata corsia. In questi diagrammi i rettangoli non sono più le azioni ma rappresentano **oggetti** o **dati** veri e propri, mentre le frecce

tratteggiate indicano un dato che viene fornito in **input** o restituito in **output**.

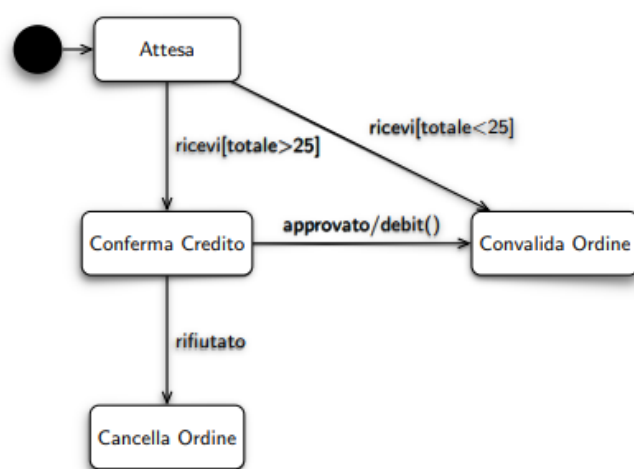
Diagramma degli Stati

Per finire abbiamo il diagramma degli stati che descrive un **intervallo di tempo limitato** durante la vita di uno specifico oggetto. Durante questo stato l'oggetto è caratterizzato da particolari valori e si aspetta che accadono determinati eventi o di compiere determinate azioni.

Il passaggio da uno stato all'altro prende il nome di **transizione**, la quale permette all'oggetto di lasciare uno stato in risposta ad un evento.

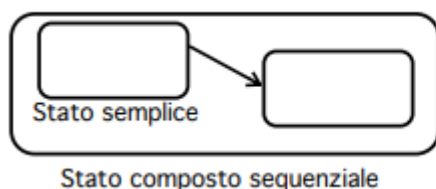
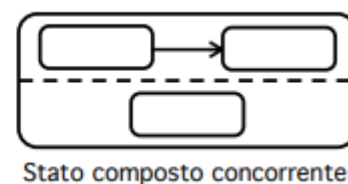
Una transizione è sempre caratterizzata da quattro elementi chiave:

- Evento scatenante;
- Possibile condizione che ne verifica la fattibilità (es. `[totale > 25]`);
- Azione da eseguire contestualmente alla transizione (`approvato/debit()`);
- Stato di arrivo.



Spesso i sistemi necessitano di **Stati Composti**. A differenza di

uno stato semplice che non ha sottostati al suo interno, uno stato composto consiste in vari sottostati che possono agire in modo sequenziale o parallelo. Se i vari sottostati sono sequenziali, solo uno di essi può risultare attivo in un determinato momento. In genere lo **stato esterno** rappresenta la macro-condizione di trovarsi in uno qualsiasi dei sottostati interni. Una transizione che entra verso, o che esce da, uno stato composto può invocare simultaneamente azioni di **entry** o di **exit**.



Processo Software e Ciclo di Sviluppo

Come detto all'inizio il ciclo di vita dello sviluppo di un software si divide in 5 fasi: analisi dei requisiti, progettazione, codifica, testing e infine manutenzione.

Partiamo dall'inizio vedendo passo passo come si sviluppano le varie fasi.

Analisi dei Requisiti

Tutto il lavoro ingegneristico inizia con l'analisi dei requisiti, una fase cruciale che porta a definire le **specifiche del sistema**, stabilendo i **servizi** che il software dovrà fornire e i **vincoli** a cui dovrà sottostare.

Bisogna distinguere la terminologia tecnica:

- Il **requisito** è la singola caratteristica che il software deve possedere, tendono ad essere molti e granulari, ovvero di piccole dimensioni.
- La **specificata** è la descrizione formale e rigorosa di queste caratteristiche.
- La **feature** è un insieme di requisiti correlati tra di loro che permette all'utente di raggiungere e soddisfare un proprio obiettivo.

I requisiti non sono tutti uguali e si suddividono in due grandi famiglie:

- **Requisiti Funzionali**: descrivono cosa il sistema deve fare (le funzionalità vere e proprie).
- **Requisiti Non-Funzionali**: descrivono come il sistema lo fa, includendo metriche di qualità come **affidabilità**, **efficienza**, **potenzialità** e la loro **convalida**.

L'ingegneria dei requisiti procede attraverso quattro passi metodologici: (1) Studio di fattibilità, (2) Analisi dei requisiti, (3) Specifica dei requisiti e (4) Convalida requisiti.

Esempi di Feature e Requisiti

- Feature di Firefox 3.6
 - Browsing privatamente: navigazione del web senza lasciare tracce
 - Password manager: ricordare le password dei siti, senza usare pop-up
 - Awesome Bar: trovare i siti preferiti in pochi secondi
 - One-click bookmark: bookmark, cerca e organizza siti web velocemente e facilmente
- Requisiti (sintetici) di Firefox 3.7
 - Eseguire i plug-in in un processo separato per migliorare la stabilità dell'applicazione e diminuire i tempi di risposta
 - Migliorare i tempi di startup
 - Ottimizzare caricamento delle pagine

Progettazione e Implementazione

Una volta terminata l'analisi dei requisiti, si passa alla fase di progettazione che ha lo scopo di **stabilire l'intera struttura software** in grado di realizzare le specifiche pattuite.

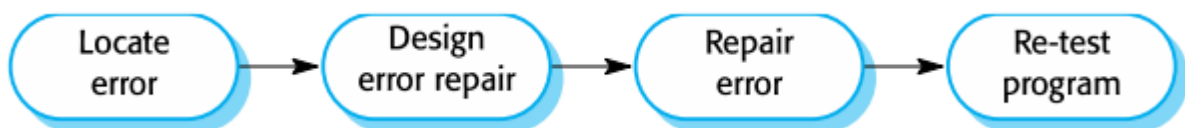
Attività della progettazione:

- suddivisione dei requisiti;
- identificazione sottosistemi, ovvero progettazione architettura software;
- specifica delle responsabilità dei sottosistemi;
- progettazione di interfacce, componenti, strutture dati, algoritmi.

Ciascuna di queste attività produce o va a integrare un documento che descrive un modello formale, come il modello degli oggetti, di sequenza, di transazione stati, strutturale o data-flow.

Subito dopo si avvia la fase di implementazione (o codifica) che si occupa di tradurre questi modelli progettuali in un programma eseguibile.

In realtà queste due fasi sono strettamente correlate e spesso si alternano. Durante l'implementazione, i programmatori iniziano già a effettuare i primi test sul codice prodotto per scovare eventuali errori (bug) e rimuoverli. La **rimozione di un bug** segue un iter preciso: bisogna localizzare l'errore nel codice, correggere concettualmente l'errore prima nel modello di design e poi materialmente nel codice, e infine ritestare il programma per accertarsi della risoluzione.



Convalida, Verifica e Validazione

La fase di Convalida (chiamata anche V&V), ha l'obiettivo di dimostrare in modo definitivo che il sistema software è conforme alle specifiche tecniche e soddisfa le **aspettative del cliente**.

Questo si ottiene tramite processi di revisione e tramite test che mirano ad eseguire il software in condizioni derivate dalle specifiche simulando l'elaborazione di dati reali.

I test sono organizzati in modo gerarchico e mirato:

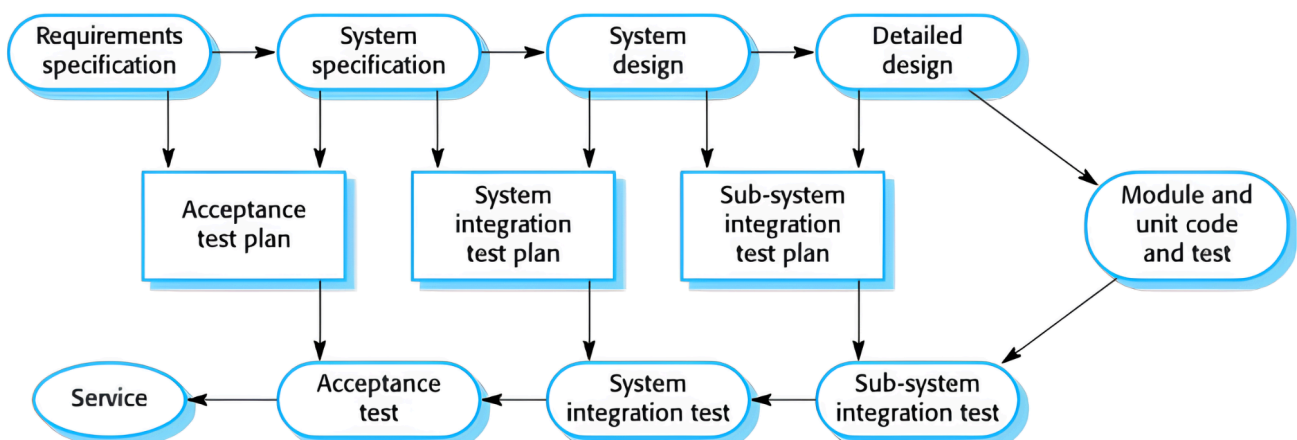
- **Unit test:** i singoli componenti (che possono essere funzioni, oggetti o raggruppamenti di essi) sono testati in maniera totalmente indipendente;

- **Test di sistema:** viene testato l'intero sistema software assemblato, dando speciale importanza all'analisi delle proprietà emergenti;
- **Alpha test** (o anche **test di accettazione**): sono condotti dagli sviluppatori utilizzando dati reali del cliente per verificare che il sistema soddisfi le esigenze;
- **Beta test:** sono condotti da alcuni clienti sul prodotto quasi completo.

A queste fasi di testing corrispondono diverse **release** del software:

- **versione alfa**, già integrata ma contenente qualche difetto o parti mancanti;
- **versione beta**, include tutte le feature richieste ma presenta ancora qualche difetto;
- **versione gold**, completa e pronta per il mercato.

In un quadro logico riassuntivo del ciclo, dai Requisiti (R) si ottiene il documento di specifica (SRS); dall'SRS si ricava il Design di Sistema (DS) e da quest'ultimo il Design Dettagliato (DD), che porta infine alla stesura del codice e ai test. I test di integrazione e di sistema si ricavano basandosi sui design (DS e DD), mentre i test di accettazione derivano direttamente dai Requisiti (R) e dall'SRS.



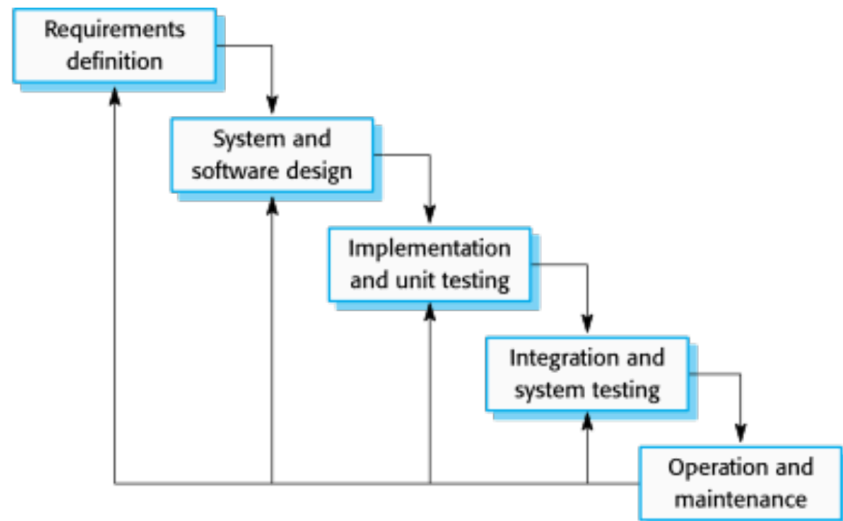
Evoluzione e Processi di Sviluppo

A differenza di un hardware, il software è intrinsecamente soggetto a modifiche e per rimanere utile nel tempo deve per forza **adattarsi ai cambiamenti** dei requisiti dovuti all'ambiente in cui si opera (business, hardware, ecc.). Per gestire questa complessità sono nati diversi modelli.

Modello a Cascata

Primo modello formalizzato nel 1970. Questo approccio si focalizza sul prodotto completo e ha una natura **sequenziale** e **statica**: si può cominciare la fase successiva esclusivamente se quella precedente è stata completata in ogni suo aspetto.

Prima si specifica tutto, poi si produce tutto ed infine si testa tutto.



Vediamo quali sono i vantaggi e gli svantaggi di questo metodo:

- **Vantaggi:** produce un'ampia documentazione e garantisce una grande consistenza tra i vari artefatti, garantisce inoltre un'alta qualità del codice. Risulta particolarmente utile se i requisiti sono stabili sin dall'inizio e per gestire sistemi grandi, complessi e critici ma anche team numerosi.
- **Svantaggi:** richiede tempi lunghi per ottenere un prodotto funzionante e prevede pochissime interazioni col cliente (quasi esclusivamente nella fase iniziale) rendendo difficile introdurre cambiamenti in corso d'opera.

Processi di Evoluzione

Nei processi evolutivi, le attività di specifica, sviluppo e convalida avvengono in maniera concorrente. Si generano versioni iniziali che passano attraverso step intermedi fino alla versione finale.

Distinguiamo due varianti principali:

- **Sviluppo per esplorazione:** sviluppatori e clienti lavorano a stretto contatto. Si parte da specifiche iniziali ben chiare e, tramite trasformazioni ed evoluzioni successive, si aggiungono nuove caratteristiche definite dal cliente fino ad arrivare al sistema finale.
- **Sviluppo Build and Fix:** è un approccio rapido e disorganizzato. La documentazione è quasi inesistente e la fase di design è del tutto trascurata. Si parte con una comprensione limitata del sistema da produrre, si costruisce una prima bozza e la si modifica a tentativi finché il cliente non si dichiara soddisfatto. Produce codice di bassa qualità.

I modelli evolutivi hanno grossi problemi: comportano spesso tempi lunghi di convergenza, mancano di una visione d'insieme del progetto e producono codice di bassa qualità o sistemi difficili da capire, modificare o persino non corretti. Per questo motivo sono applicabili solo a **sistemi di piccole dimensioni**, a **prototipi con vita breve** o **sviluppo di singole parti**, come interfacce utenti, di sistemi più ampi.

Modello Incrementale e CBSE

Oltre ai precedenti, esistono approcci come lo **sviluppo incrementale**, in cui si implementano primariamente le funzionalità di base o ad alta priorità a cui si aggiunge ciclicamente un altro codice per raggruppamenti di funzionalità secondarie fino al completamento totale.

Un altro paradigma è il processo **CBSE** basato su **COTS** (Components Off The Shelf, ovvero componenti preesistenti). Questo prevede un'analisi dei componenti già disponibili sul mercato, un'eventuale modifica dei requisiti per adattarli ai componenti, una progettazione focalizzata sul riuso del codice e infine lo sviluppo per integrare il tutto.

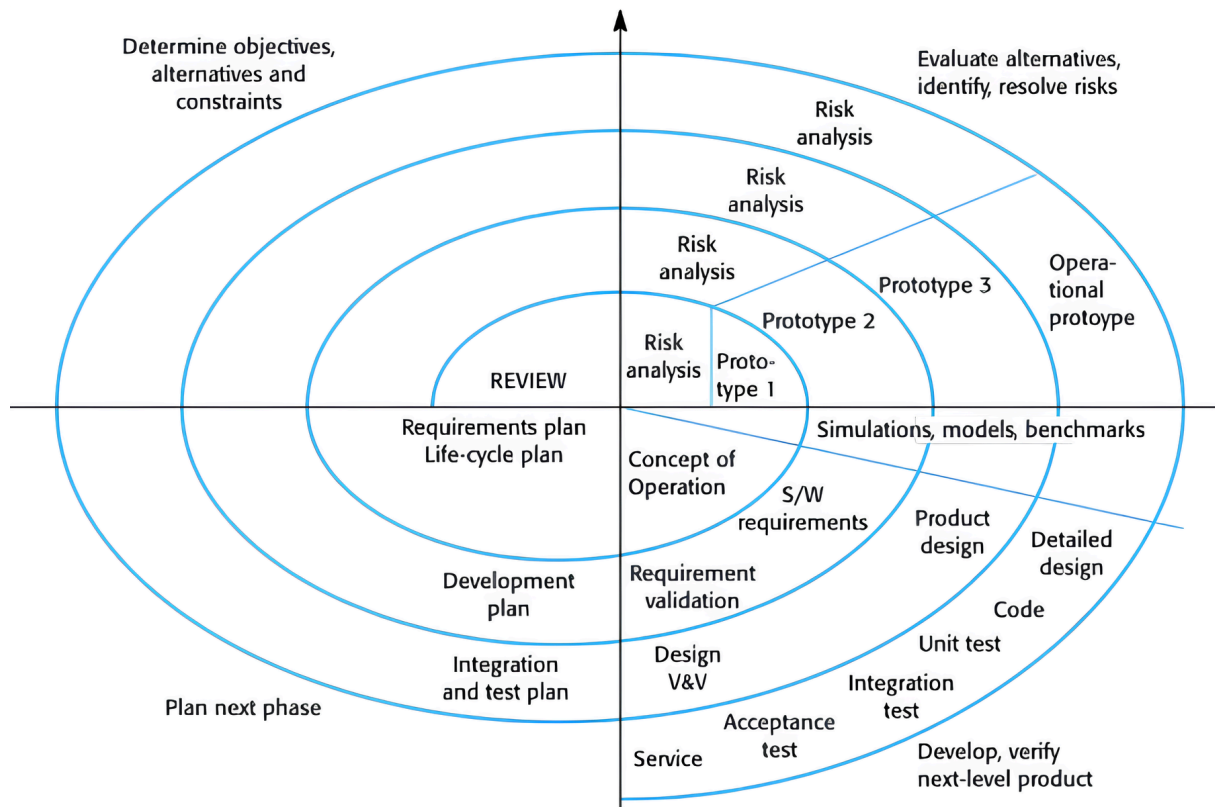
Modello a Spirale

Ideato nel 1988, il processo a spirale non punta immediatamente al software completo ma focalizza lo sviluppo su tanti prodotti parziali o sottosistemi funzionali. È un processo tipicamente **agile** dove ogni ciclo (o spira) della spira rappresenta una precisa fase di progetto.

Ciascun ciclo si articola in quattro settori di lavoro:

- **Determinazione degli obiettivi:** si identificano gli obiettivi specifici per la spirale corrente;
- **Valutazione e analisi dei rischi:** vengono valutati i rischi e si intraprendono azioni per mitigare quelli più critici. Nel software, un rischio è un'incognita che può impedire di soddisfare tutti i requisiti (ovvero il successo del progetto) e si misura per probabilità di occorrenza e impatto o danno sul progetto.
- **Sviluppo e convalida:** si produce e si valida una singola porzione di codice;
- **Pianificazione e revisione:** il progetto viene revisionato e si pianifica il ciclo successivo.

Questo modello agile offre grandi vantaggi operativi: richiede poco tempo per rilasciare la primissima versione del prodotto, massimizza le opportunità di interagire col cliente durante i cicli, e assicura che, al termine di ogni fase, venga prodotto codice che viene subito testato e integrato nel sistema complessivo.



Il punto di partenza per comprendere le moderne metodologie di ingegneria del software è il **Manifesto for Agile Software Development**, stilato nel 2001 da Beck, Cockburn, Cunningham, Fowler e altri.

Questo approccio rivoluziona la gestione dei progetti affermando che l'auto-organizzazione, la collaborazione, la comunicazione tra i membri del team e l'adattabilità del prodotto sono molto più importanti rispetto al mero mantenimento dell'ordine e della coerenza delle attività.

I valori fondamentali del manifesto privilegiano:

- **Individui** rispetto a processi e strumenti;
- **Disponibilità di software funzionante** rispetto alla documentazione;
- **Collaborazione con il cliente** rispetto alla negoziazione dei contratti;
- **Pronta risposta ai cambiamenti** rispetto all'esecuzione di un piano.

L'**agilità**, in questo senso, significa accogliere positivamente le richieste di modifica anche in fasi avanzate dello sviluppo, consegnando frequentemente release del sistema funzionanti e puntando sempre all'eccellenza tecnica con un gruppo di sviluppatori fortemente motivati.

Una delle incarnazioni più celebri e rigorose di questo manifesto è l'**Extreme Programming (XP)**, introdotto da Kent Beck nel 1999. Si tratta di un approccio basato su cicli di sviluppo brevissimi: il software viene consegnato in piccoli incrementi (release) che richiedono solitamente solo

due settimane di lavoro. XP è pensato per piccoli team, dai 2 ai 12 sviluppatori, e sposta il focus sul **miglioramento costante del codice** riducendo al minimo la documentazione tradizionale, che viene sostituita da strumenti agili come le Story Card e le CRC Card. Questo metodo è particolarmente adatto per progetti in cui i requisiti sono instabili o i rischi sono elevati, come nel caso di **tempi di consegna stretti** o **tecnologie innovative** per il team di sviluppo. I principi cardine prevedono di ottenere feedback rapidamente, assumere sempre la via della semplicità, procedere attraverso cambiamenti incrementali e produrre un lavoro di altissima qualità.

Per mettere in pratica questi principi, l'Extreme Programming si avvale di **12 pratiche** specifiche:

- **Gioco di Pianificazione:** è il processo di definizione dei requisiti di un'attività. Gli **utenti** (clienti) scrivono i requisiti sottoforma di **storie** su apposite Story Card e ne definiscono la priorità. Gli **sviluppatori valutano le storie** e ne **stimano i tempi** di realizzazione, se una storia risulta troppo complessa viene chiesto al cliente di dividerla in più parti. Successivamente, si pianificano le release e le iterazioni (spesso riempiendo slot di 3 settimane) in cui gli sviluppatori scompongono ulteriormente ciascuna storia in task pratici, affrontando prima task più rischiose.

Dimensioni: 5" x 3", circa 12x7 cm. Descrizione storie: 2-3 frasi su una card che sono importanti per il cliente e sono scritte dal cliente, possono essere testate, permettono di ricavare una stima del loro tempo di sviluppo, possono essere associate a priorità. Campi di una story card: Data, Numero, Priorità, Tempo stimato, Riferimenti Descrizione requisito, Lista di attività per ciascun requisito, Note.

Story card

Titolo progetto	Autore	Rigo piano progetto	Priorità
Gestione Immagini	Jim	1	1
Storia Mostrare le immagini (jpg, png) presenti su una cartella del file system su una griglia di 10x6 immagini, indipendentemente dalla risoluzione dello schermo.			

Titolo progetto	Autore	Rigo piano progetto	Priorità
Gestione Immagini	Jim	2	1
Storia Mostrare le immagini scalate (ridimensionate) rispettando le proporzioni iniziali delle immagini.			

Titolo progetto	Autore	Rigo piano progetto	Priorità
Gestione Immagini	Jim	3	1
Storia L'utente potrà selezionare immagini digitando lettere all'interno di una casella di testo. Le immagini selezionate saranno quelle i cui nomi di file contengono il testo inserito.			

Stimatore	Tempo stimato	Data	Sviluppatori	Tempo impiegato
Jack	2 giorni	15-03-2011		

- **Programmazione a Coppie** (Pair Programming): la stesura del codice avviene affiancando due programmatori, che assumono **due**

ruoli distinti. Il **driver** pensa al miglior modo per implementare il metodo e usa fisicamente mouse e tastiera. Il **navigator** supervisiona il lavoro ragionando sui test da applicare e cercando costantemente un modo per semplificare il lavoro. Questa pratica migliora la disciplina collettiva e distribuisce la conoscenza dell'architettura di sistema a tutti i membri del team.

- **Piccole release:** lo sviluppo procede rilasciando incrementi (release) di funzionalità più piccoli possibile, con cicli temporali di 2 settimane per release. vantaggi offerti da questo approccio sono diversi: garantisce un feedback estremamente rapido, costruisce un senso di realizzazione, abbassa i rischi di fallimento e costruisce la fiducia del cliente, garantendo una forte reattività ai cambiamenti in corso d'opera.
- **Metafora:** l'intero progetto è guidato da una singola metafora condivisa mirata a rappresentare l'architettura generale in modo intuitivo, semplificare le discussioni tecniche e assicurare che il cliente **comprenda comodamente** i fondamenti del sistema.
- **Design semplice:** il design corretto per la creazione di un software è sempre e solo quello che fa superare i test, non ha sezioni di codice duplicate, esprime le intenzioni dei programmatori chiaramente e ha il numero più piccolo in assoluto di classi e metodi. La regola è **fare la cosa più semplice che può funzionare**, pagando il costo strutturale solo per ciò che si usa effettivamente e usando strumenti leggeri come le CRC card per documentare il design.
- **Testing continuo:** bisogna testare costantemente tutto ciò che può potenzialmente generare errori. I test costituiscono una specifica dei requisiti in formato eseguibile. Si dividono in due categorie: **test funzionali**, scritti dall'utente in base al proprio punto di vista, **unit test**, scritti dai programmatori idealmente ancor prima di iniziare a codificare (TDD).
- **Possesso del codice collettivo:** chiunque nel gruppo può apportare modifiche o aggiungere codice in qualunque parte dell'applicativo. Ogni qualvolta uno sviluppatore trova un problema, ha l'autorizzazione e la responsabilità di risolverlo direttamente, dato che **tutto il team è responsabile** in modo organico per l'intero sistema.
- **Integrazione continua:** le modifiche al codice, una volta testate, vengono integrate nel repository centrale al massimo entro un giorno. Dopo ogni singola integrazione, tutti gli unit test del progetto devono passare con successo. Se anche un solo test fallisce, la coppia di sviluppatori che ha introdotto il codice deve riparare

immediatamente l'errore; qualora ciò non sia rapidamente fattibile, la modifica problematica viene buttata via per ricominciare da zero.

- **Settimana lavorativa di 40 ore:** è fissato un limite alle ore di lavoro per mantenere gli **sviluppatori mentalmente freschi**. Un eccesso di **stanchezza** può generare **errori logici**. Se si rende necessario lavorare costantemente oltre le 40 ore significa che c'è un difetto più grave alla radice, piani iterativi e frequenti servono proprio a evitare sovraccarichi di compiti.
- **Cliente in sede:** requisito chiave di XP è che il cliente lavori fisicamente nella sede di sviluppo per portare avanti le sue idee. La sua presenza è **indispensabile** per redigere test funzionali, stabilire le priorità operative, fornire l'esatto contesto di business dietro alle decisioni e rispondere tempestivamente ai quesiti dei programmatori. *"Se non si può avere il cliente in sede, forse il progetto non ha un'importanza strategica così alta"*.
- **Standard di codifica:** tutto il team deve adottare **regole uniformi** e rigide su come scrivere il codice. Costrutti inutilmente complicati sono vietati a favore della semplicità di lettura. Regole univoche per l'indentazione, posizionamento delle parentesi, scelta di nome di metodi, attributi e classi e per lo stile dei commenti, assicurando che nessuno debba perdere tempo a riformattare il lavoro di un collega.
- **Refactoring:** la pratica di **migliorare continuamente** la struttura e la pulizia del codice senza alterarne il comportamento. Si procede a piccoli passi puntando ad eliminare ogni forma di ripetizione. È un'**operazione sicura** perché viene validata in tempo reale dagli unit test che intercettano subito eventuali problemi.

Scrum

La prima guida su Scrum è stata prodotta nel 2010. Scrum è una struttura di base per aiutare le persone a generare soluzioni per problemi complessi.

Proprio per questa sua natura viene descritto come un approccio **semplice** ma **incompleto**: ossia fornisce un'impalcatura su cui è possibile utilizzare vari processi e tecniche specifiche a seconda del contesto.

Scrum promuove un ambiente ciclico in cui:

- un **Product Owner** (proprietario) elenca il lavoro da svolgere su un problema complesso tramite un **Product Backlog** (lavoro arretrato);
- lo **Scrum Team** (sviluppatori) trasforma una parte del lavoro in un incremento durante uno **Sprint** (iterazione);
- lo **Scrum Team** e chi ha interesse nel prodotto (**stakeholder**) valutano i risultati e fanno aggiustamenti per il prossimo sprint.

A livello teorico, Scrum si basa fortemente su due concetti chiave: l'**empirismo** e il concetto di **lean** (snello o agile):

- **Empirismo**: la conoscenza viene dall'esperienza e le decisioni sono prese in accordo con quel che viene osservato.
- **Lean**: mira alla massima efficienza, riducendo lo spreco per concentrarsi sull'essenziale.

Impiegando un approccio che è simultaneamente iterativo e incrementale, Scrum permette di ottenere una forte prevedibilità sui risultati e di mantenere un rigoroso **controllo del rischio**.

Per sostenere questo processo, Scrum si regge su tre pilastri fondamentali:

- **Trasparenza**: il processo e il lavoro devono essere sempre visibili sia a chi svolge il lavoro, sia a chi riceve i risultati. Le decisioni importanti sono basate sui tre **artefatti** (Product backlog, Sprint backlog, Incremento), è proprio la trasparenza di questi elementi che abilita l'ispezione.
- **Ispezione**: gli artefatti di Scrum e il progresso verso l'obiettivo devono essere ispezionati frequentemente e in modo diligente per scoprire potenziali problemi. Scrum fornisce la cadenza delle ispezioni tramite 5 **eventi**. A sua volta l'ispezione permette l'adattamento.
- **Adattamento**: se durante l'ispezione si nota qualche aspetto del processo o risultato inattendibile, è necessario aggiustare immediatamente il processo. Questo aggiustamento deve essere attuato subito per evitare ulteriori divergenze del piano.

La messa in pratica di questi principi avviene attraverso i 5 eventi di Scrum, governati da regole temporali ben precise:

1. **Sprint**: iterazione base su cui si fonda Scrum, ha una lunghezza fissata, tipicamente un mese;
2. **Sprint Planning**: evento che produce la pianificazione di tutto il lavoro che dovrà essere svolto durante lo Sprint. Dura al massimo 8 ore per uno Sprint di un mese;
3. **Daily Scrum**: incontro giornaliero di 15 minuti, svolto sempre nello stesso punto. Serve a ispezionare il progresso quotidiano e ad adattare lo **Sprint Backlog**. Migliora la comunicazione, identifica gli ostacoli, promuove il processo decisionale.
4. **Sprint Review**: ispeziona i risultato dello Sprint e determina gli adattamenti. Dura al massimo 4 ore per uno Sprint di un mese;
5. **Sprint Retrospective**: ha lo scopo di migliorare la qualità e l'efficacia del lavoro. Durante questo evento il team ispezione com'è andato

l'ultimo Sprint concentrandosi sugli individui, le iterazioni, i processi, i tool e la **Definition of Done**. Quest'ultima è una descrizione dello stato in cui deve trovarsi l'incremento quando soddisfa le misure di qualità richieste per il prodotto.

Design Pattern Observer

Conosciuto anche come Publish-Subscribe, il suo intento è definire una dipendenza di tipo **uno a molti** fra diversi oggetti.

Grazie a questa struttura, nel momento in cui un oggetto cambia il proprio stato interno, tutti i suoi oggetti dipendenti vengono notificati e aggiornati in maniera **automatica**.

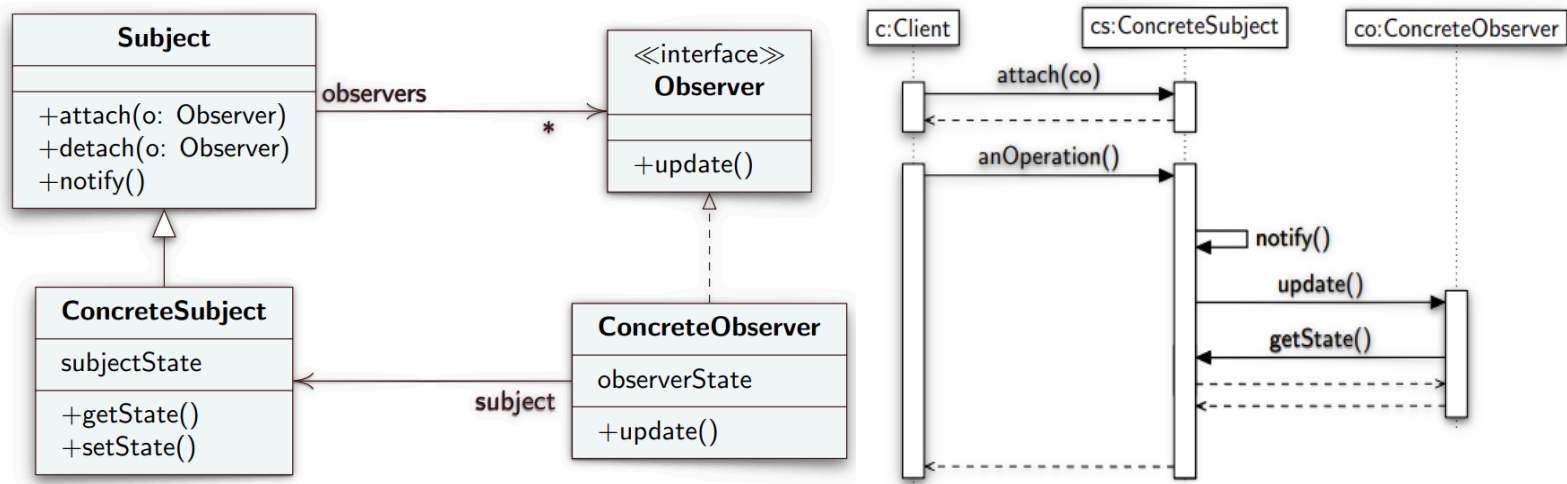
La motivazione di base di questo pattern risiede nella necessità di mantenere la **consistenza** tra oggetti correlati all'interno di un sistema partizionato in classi cooperanti. Un esempio classico sono uno strumento di presentazione separato dai dati dell'applicazione. Le classi che tengono i dati e quelle di presentazione sono separate e riusabili. Uno spreadsheet e un diagramma sono dipendenti dall'oggetto che contiene i dati e dovrebbero essere notificati dei cambiamenti dei dati.

Per stabilire le relazioni fra oggetti dipendenti, il pattern individua due ruoli chiave: il Subject e l'Observer. Un'astrazione può avere due aspetti dipendenti e **incapsularli** in oggetti separati permette di **riutilizzarli in modo indipendente**. Questo pattern è applicabile quando la modifica di un oggetto richiede il cambiamento di altri oggetti ma non si conosce il numero preciso di oggetti da cambiare, permettendo al Subject di notificare tali cambiamenti senza dover conoscere l'identità specifica di chi riceve la notifica. Questo garantisce che gli oggetti non siano strettamente accoppiati.

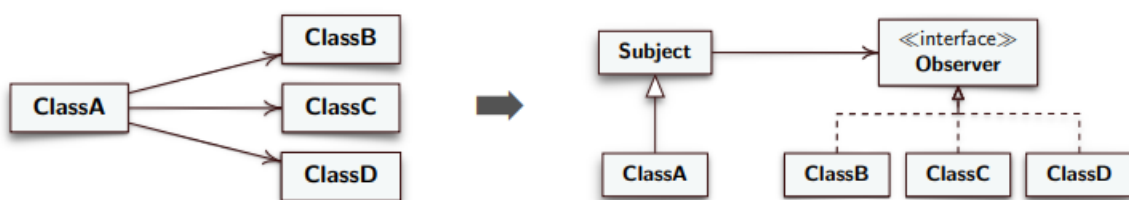
Il pattern è strutturato con i seguenti elementi:

- il **Subject** conosce i suoi osservatori che possono essere in numero variabile, fornisce le operazioni per aggiungere e togliere oggetti (**attach()**, **detach()**) e per notificarli (**notify()**).
- l'**Observer** definisce un'interfaccia comune a tutti gli oggetti che devono ricevere le notifiche, solitamente esponendo un'operazione **update()**.
- il **ConcreteSubject** è l'oggetto reale che mantiene lo stato di interesse e si occupa di inviare la notifica agli osservatori quando tale stato muta.
- il **ConcreteObserver** mantiene un riferimento al ConcreteSubject e conserva al suo interno uno stato che deve rimanere sempre allineato con quello del Subject. Implementa inoltre l'interfaccia

Observer e, una volta ricevuta la notifica di cambiamento, interroga il Subject per recuperare il nuovo dato aggiornato.



Quando lo stato dell'oggetto **ConcreteSubject** cambia, l'oggetto **ConcreteSubject** chiama **notify()** che chiamerà **update()** sugli oggetti **ConcreteObserver** per aggiornarli.



L'adozione dell'Observer porta con sé importanti conseguenze architetturali. Il vantaggio principale è il basso accoppiamento: il **Subject** interagisce solo con l'interfaccia astratta **Observer**, ignorando le classi concrete degli osservatori. Così facendo il sistema diventa flessibile e facile da modificare. La notifica inviata da un **ConcreteSubject** è mandata a tutti gli oggetti che si sono iscritti ma non sa quanti sono. Gli osservatori possono essere rimossi in qualunque momento. L'osservatore sceglie se gestire o ignorare la notifica.

L'aggiornamento da parte del Subject può innescare una reazione a catena di operazioni sugli **Observer** e altri oggetti per gli aggiornamenti. La notifica non dice agli Observer cosa è cambiato nel **ConcreteSubject**, è un evento che indica il completamento di un'operazione del **ConcreteSubject**.

Bisogna però tenere a mente le seguenti considerazioni:

- **Ottimizzazione della memoria:** Quando in un sistema si hanno numerosi Subject ma pochi Observer, far mantenere a ogni Subject i riferimenti ai propri osservatori può occupare troppa memoria; per ridurre lo spazio impegnato si può ricorrere a un'unica tabella associativa condivisa tra tutti i Subject.
- **Gestione di osservazioni multiple:** Un singolo Observer ha la possibilità di osservare più oggetti contemporaneamente. Per permettere all'Observer di capire esattamente chi ha generato la notifica, è opportuno passare il riferimento del Subject stesso come parametro all'interno del metodo `update()`.
- **Controllo del flusso di notifiche:** Un Subject non è obbligato a chiamare il metodo `notify()` immediatamente dopo ogni singola modifica, ma può attendere che si verifichi un certo numero di cambiamenti per evitare di inviare continue notifiche agli Observer, ottimizzando così le prestazioni del sistema.
- **Mantenimento della consistenza bidirezionale:** Per garantire che Observer e Subject rimangano sempre coerenti, se avviene un cambiamento all'interno dell'Observer, questo deve essere comunicato al Subject (ad esempio utilizzando un metodo `setState()`).
- **Gestione del ciclo di vita degli oggetti:** Quando si decide di eliminare definitivamente un Subject dalla memoria, è necessario avvisare prima tutti i suoi Observer registrati, affinché questi possano cancellare il riferimento che li lega al Subject ormai distrutto e prevenire errori a runtime.
- **Modelli Push vs. Pull:** Il trasferimento dei dati durante la notifica può avvenire in due modi: seguendo il modello *push*, il Subject passa le informazioni aggiuntive sui cambiamenti direttamente come argomenti della chiamata `update()`. Al contrario, nel modello *pull*, il Subject invia solo la notifica e aspetta che sia l'Observer ad attivarsi per leggere il nuovo stato.
- **Notifiche mirate (Filtraggio degli eventi):** In fase di registrazione, gli Observer possono specificare a quali particolari eventi sono interessati. Grazie a questa impostazione, tramite il metodo `update()` verrà inviato loro solamente ciò che è effettivamente cambiato di loro interesse, evitando trasferimenti di dati inutili.

Observer e Java 9

Il problema risolto dall'Observer è così diffuso che storicamente Java forniva una sua soluzione nella libreria `java.util` tramite la classe `Observable` (che funge da Subject) e l'interfaccia `Observer`. La classe `Observable` utilizzava un flag interno modificabile tramite `setChanged()` per indicare l'avvenuto mutamento di stato prima di chiamare `notifyObservers()`. Tuttavia, questa implementazione storica presenta un limite legato al multithreading: quando il Subject chiama `notify()`, l'esecuzione del metodo `update()` avviene sul thread del chiamante, bloccandolo di fatto fino al termine dell'aggiornamento di tutti i `ConcreteObserver`.

Per risolvere questo blocco sincrono e evitare che un publisher inondi di dati un subscriber, da **Java 9** (2017) i vecchi tipi sono stati deprecati in favore dei **Reactive Streams**, all'interno del pacchetto `java.util.concurrent`, introducono un approccio asincrono tramite nuove interfacce:

- **Publisher**: fornisce gli elementi e definisce il metodo `subscribe()`. La classe concreta `SubmissionPublisher` implementa questo ruolo dedicando un thread asincrono per l'invio dei messaggi, bloccandosi solo se il ricevente non è pronto.
- **Subscriber**: riceve gli elementi implementando i metodi `onNext()` (per i nuovi dati), `onError()` e `onComplete()`.
- **Subscription**: rappresenta il collegamento tra Publisher e Subscriber, permettendo all'osservatore di controllare il flusso di dati richiedendo uno specifico numero di messaggi (`request(n)`) o fermando l'invio (`cancel()`).

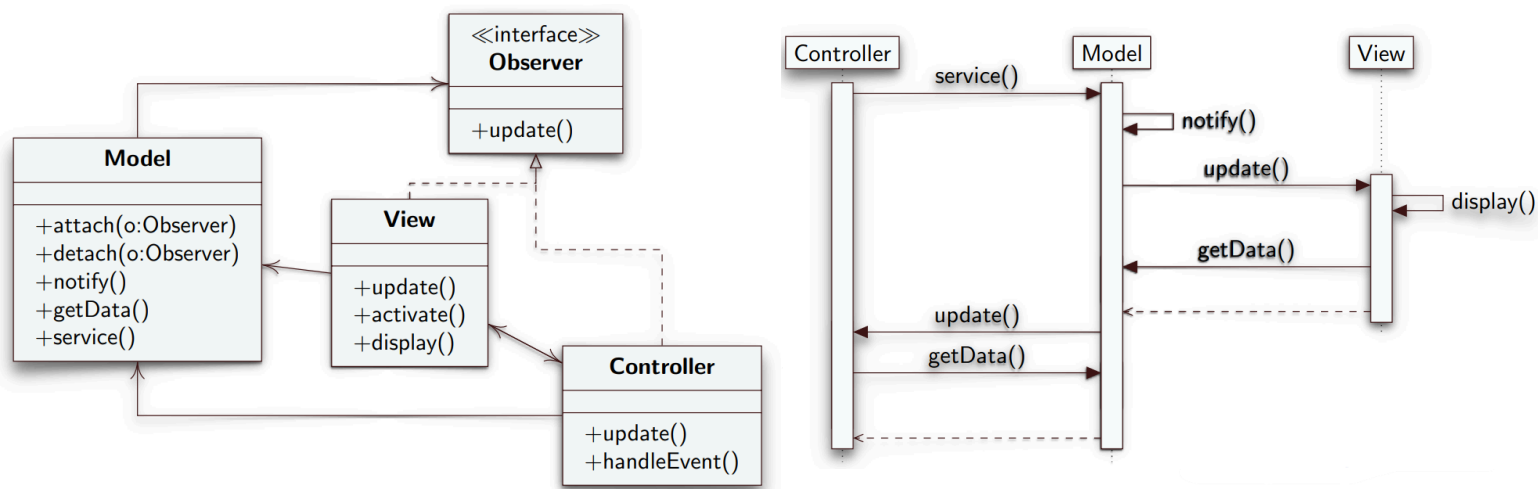
Model View Controller

L'MVC è un pattern architetturale fondamentale per le applicazioni interattive, nato per isolare la logica di business dall'interfaccia utente.

Le interfacce utente sono soggette a comuni cambiamenti legati a piattaforme o dispositivi diversi ed è vitale che lo stesso nucleo di dati possa essere rappresentato in modi differenti (ad esempio con grafici diversi) reagendo istantaneamente alle manipolazioni dell'utente senza intaccare le funzionalità principali del sistema.

Il Model view Controller si divide in tre componenti:

- il **Model** incapsula i dati e le funzionalità principali. È totalmente indipendente da input o output e funge da Subject, registra View e Controller e li avvisa in caso di modifiche ai dati.
- la **View** si occupa di mostrare i dati all'utente. Funge da Observer del Model, quando riceve una notifica legge i nuovi dati dal Model e li aggiorna. Ogni View è solitamente associata ad un proprio Controller.
- il **Controller** gestisce l'interazione con l'utente ricevendo input da periferiche sotto forma di eventi. Può essere anch'esso un Observer del Model. Il suo compito è tradurre le azioni dell'utente in richieste di servizio dirette al Model (per modificare dati) o in comandi per la View (per modificarne l'aspetto).



Design Pattern State

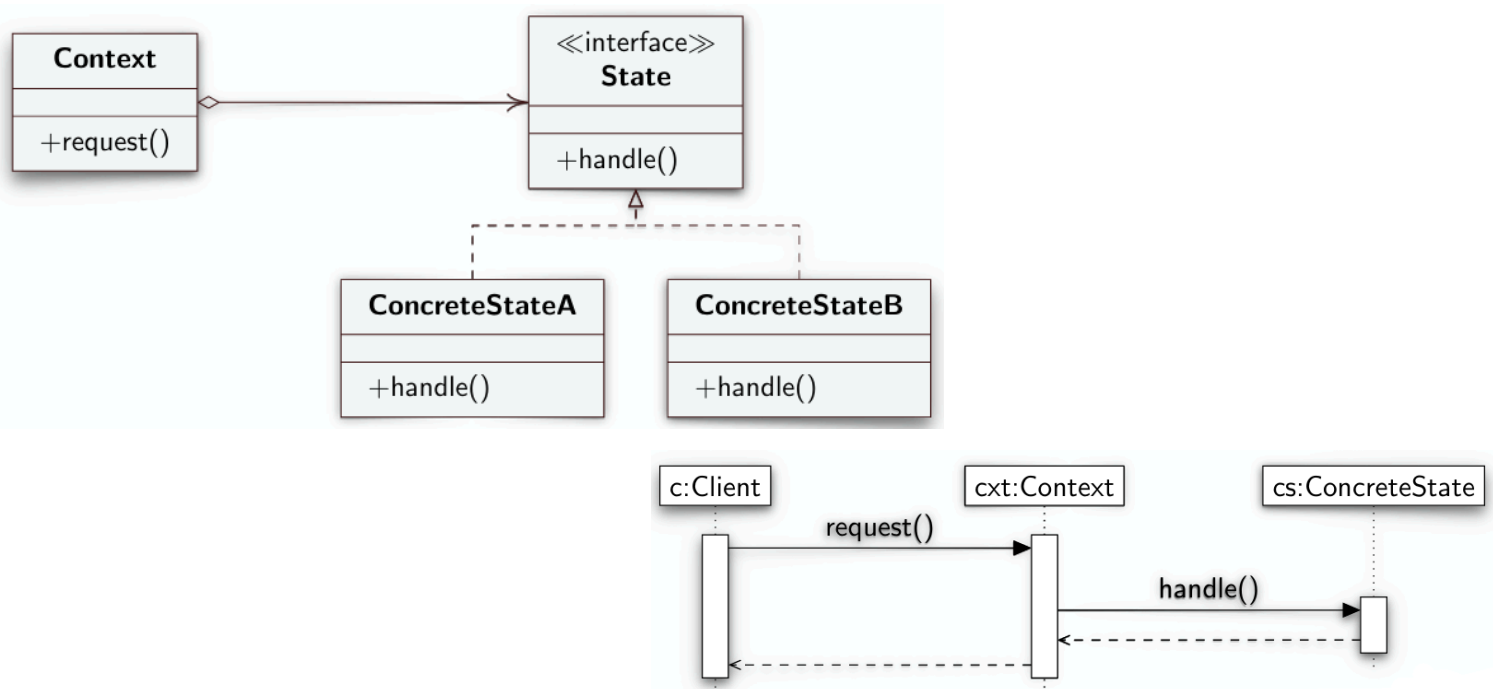
State è un pattern comportamentale il cui intento principale è permettere ad un oggetto di **alterare il suo comportamento** nel momento in cui il suo **stato interno cambia**.

All'esterno questo cambiamento è così radicale da far sembrare che l'oggetto abbia effettivamente cambiato la propria classe di appartenenza a run-time.

Questo pattern risolve un problema architetturale molto comune, ossia gestire oggetti il cui comportamento dipende fortemente dal loro stato e deve adattarsi dinamicamente durante l'esecuzione. Molto spesso **in assenza di questo pattern**, lo stato viene rappresentato dal valore di una o più variabili. Di conseguenza le operazioni della classe finiscono per contenere **strutture condizionali ripetute** in maniera simile in vari metodi, rendendo il codice difficile da leggere e mantenere.

Per risolvere questo problema, il pattern State estrae ogni singolo ramo condizionale e lo incapsula in una classe separate. A livello strutturale si divide in tre parti:

- **Context**: la classe che definisce l'interfaccia utilizzata dal client. Al suo interno mantiene un'istanza di una classe **ConcreteState** che rappresenta lo stato in cui si trova il Context in quel momento.
- **State**: l'interfaccia che dichiara e incapsula i comportamenti associati ad un particolare Context.
- **ConcreteState**: sono le sottoclassi concrete che implementano l'interfaccia **State**. Ognuna di queste classi contiene esclusivamente il comportamento associato a uno specifico stato del Context.



Il funzionamento del pattern si basa sulla **delega**: quando il Context riceve una richiesta dipendente dallo stato da parte di un client, non la gestisce in prima persona ma la passa all'oggetto **ConcreteState** corrente. Qualora lo stato avesse bisogno di informazioni aggiuntive per operare, il Context può passare sé stesso come argomento al **ConcreteState**.

Per le classi client, l'unica interfaccia di comunicazione rimane il Context, mantenendo l'architettura pulita. Infine, la logica di transizione, ovvero la decisione su quale debba essere lo stato successivo e sotto quali circostanze avvenga il passaggio può essere gestita indifferentemente dal Context stesso o delegata ai singoli **ConcreteState**.

L'utilizzo di questo pattern porta numerosi vantaggi progettuali. Il primo fra tutti è che il comportamento specifico di ogni stato è localizzato in una sola classe **ConcreteState**, **partizionando** in modo netto **i vari comportamenti**. Questo rende estremamente facile **aggiungere nuovi stati** e nuove transizioni al sistema, è sufficiente creare nuove sottoclassi senza dover modificare il codice esistente.

Inoltre, separare la logica che gestisce il cambiamento di stato in classe dedicate impone una **struttura più chiara** e **aiuta a prevenire** la creazione di **stati inconsistenti**, poiché i cambiamenti vengono decisi in modo centralizzato e non sparsi tra mille istruzioni condizionali nella classe principale. L'unico svantaggio è il numero totale di classi, esso sarà più alto ma le singole classi risulteranno molto più semplici, focalizzate e coese.

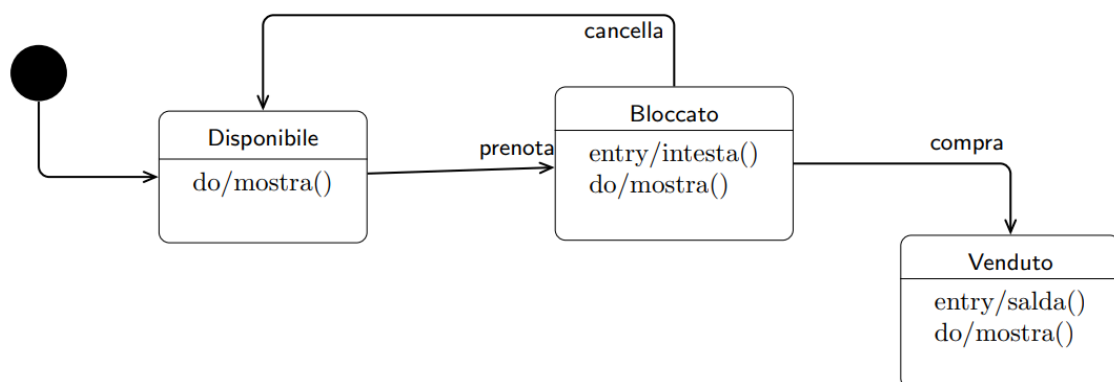
Di seguito vediamo un esempio di applicazione del pattern:

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

Possiamo dividere i requisiti in tre categorie:

- **Classi**: biglietto;
- **Attributi**: codice, prezzo, data, nome;
- **Operazioni**: prenotare, acquistare, annullare.

La classe Biglietto potrà avere 3 stati: **disponibile**, **bloccato** (prenotato), **venduto**.



Quindi la classe **Biglietto** dovrà implementare i metodi: **prenota()**, **cancella()**, **compra()** e **mostra()**. Ciascuna operazione controllerà lo stato in cui si trova il biglietto.

Per soddisfare i requisiti di questo software si può pensare di fare uno *switch-case* in ogni metodo per controllare lo stato del biglietto. Tuttavia così si creano rami condizionali **duplicati** e **difficili da modificare** e rendono il codice poco flessibile.

La soluzione che propone il design pattern State si basa sulla tecnica di Refactoring **Replace Conditional with Polymorphism**. Questo tipo di soluzione rende il codice più semplice da comprendere e modificare, ciascuno stato avvia quando occorre una transizione e questo permette di eliminare l'avvio delle transizioni da Context, e quindi gli switch su esso. L'interfaccia usata dai **ConcreteState** permette di ritornare il riferimento a un nuovo state (è detta **fluent**).



Design Pattern Mediator

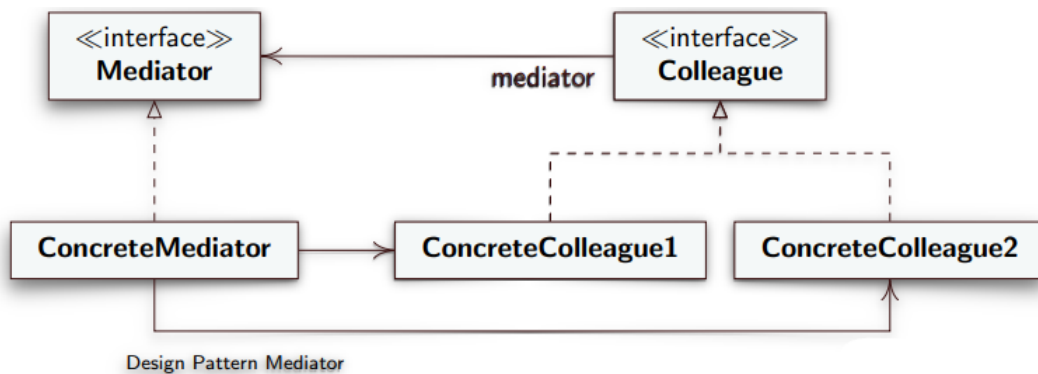
Questo pattern ha come intento principale quello di definire un oggetto capace di **incapsulare** le modalità con cui un gruppo di oggetti interagisce all'interno di un sistema. Il Mediator promuove il **loose coupling** (basso accoppiamento): evita che gli oggetti interagiscano direttamente e permette di modificare le loro interazioni indipendentemente da essi.

Nello sviluppo del software distribuire le responsabilità tra molteplici oggetti può generare un'intricata rete di connessioni. Nel caso peggiore, l'architettura degenera al punto che ogni oggetto finisce per conoscere le interfacce di tutti gli altri, rendendo il sistema come se fosse un blocco monolitico. La frammentazione delle comunicazioni rende molto difficile mantenere o modificare il comportamento all'interno del sistema. La soluzione proposta dal pattern consiste nell'**incapsulare** questo comportamento collettivo all'interno di un oggetto separato detto **mediatore**, che funge da intermediario esclusivo e spezza le dipendenze dirette.

Per isolare in modo efficace le comunicazioni complesse, il pattern utilizza una struttura rigida basata sulla separazione dei ruoli:

- **Mediator**: interfaccia generica che funge da punto d'incontro e di comunicazione per tutti gli oggetti connessi.
- **ConcreteMediator**: classe concreta che implementa l'interfaccia del Mediatore e racchiude la logica cooperativa reale. Si occupa solo di mantenere i riferimenti ai vari componenti e di coordinare attivamente le loro interazioni.

- **Colleague**: interfaccia o classe astratta che definisce il tipo generico per tutti i componenti del sistema che devono cooperare tra di loro.
- **ConcreteColleague**: classe concreta che definisce i componenti effettivi del sistema. Ciascun **ConcreteColleague** conosce esclusivamente il proprio **Mediator** di riferimento, invia le notifiche e le richieste di azione solo al Mediatore, riceve dal mediatore comandi e dati di cui ha bisogno.



L'adozione del Mediator è particolarmente indicata quando ci si trova di fronte a un insieme di oggetti che comunicano in modo strutturato ma eccessivamente complesso, generando **interdipendenze** difficili da comprendere e gestire. È una scelta architetturale molto valida anche quando si desidera **riutilizzare** un oggetto specifico che risulta **troppo accoppiato** agli altri componenti, oppure quando un comportamento distribuito tra diverse classi **necessita di essere aggiornato** o modificato senza dover ricorrere continuamente alla creazione di nuove sottoclassi.

Il pattern sposta la complessità intrinseca della gestione delle dipendenze direttamente all'interno del Mediatore. Questo rende le classi **Colleague** molto più pulite, facili da implementare e mantenere ed estremamente più riusabili. Tuttavia applicare il Mediator porta con sé anche degli svantaggi come il fatto che il codice del **ConcreteMediator** diventa quasi sempre difficilmente riusabile altrove, essendo fortemente personalizzato e cucito addosso alle logiche di una specifica applicazione.

Design Pattern Command

L'intento di questo pattern è quello di incapsulare una specifica richiesta all'interno di un oggetto. Questo approccio permette di parametrizzare i client con richieste sempre differenti, di gestire facilmente code o log di richieste e di supportare l'annullamento delle operazioni.

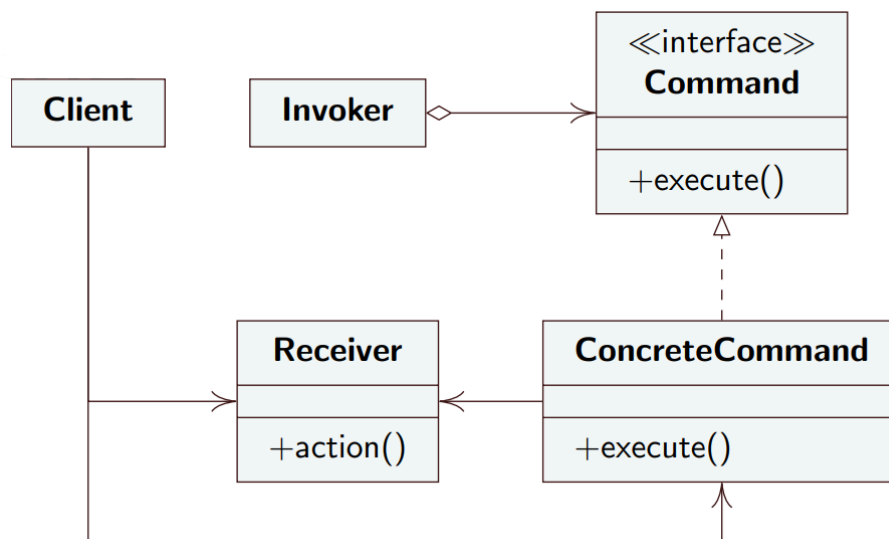
La motivazione alla base di questo pattern nasce da una problematica comune: a volte può essere necessario inviare una richiesta a determinati

oggetti senza conoscere nulla dell'operazione richiesta o del reale ricevente che dovrà eseguirla.

Attraverso il pattern Command, la richiesta viene trasformata in un **oggetto a sé stante**, quest'oggetto può essere **immagazzinato** nel sistema o **passato** come altri oggetti. La chiave di questo pattern è la classe astratta **Command**, la quale dichiara un'interfaccia standard per eseguire le operazioni. Saranno poi le varie sottoclassi a specificare l'esatta coppia formata da "ricevente" e dall'"azione" da compiere, immagazzinando il **ricevente** come **attributo** interno e implementando il metodo che materialmente effettua la richiesta. Il ricevente conosce già ciò che è necessario per servire la richiesta.

Il pattern si suddivide in 5 componenti:

- **Command**: interfaccia base per eseguire un'operazione;
- **ConcreteCommand**: classe concreta che definisce il collegamento tra Receiver e un'azione, implementa l'operazione **execute()** chiamando le corrispondenti azioni sul Receiver;
- **Client**: crea un oggetto **concreteCommand** e imposta il suo Receiver;
- **Invoker**: classe concreta che chiede al comando di effettuare la richiesta;
- **Receiver**: classe concreta che conosce come effettuare le operazioni associate alla richiesta. Qualunque classe può essere un Receiver.



Il **Client** crea un oggetto **ConcreteCommand** e specifica il suo oggetto **Receiver**. Un oggetto **Invoker** tiene l'oggetto **ConcreteCommand**. L'oggetto **Invoker** effettua una richiesta chiamando **execute()** sull'oggetto **Command**. Quando i comandi sono reversibili, il **ConcreteCommand** conserva lo stato per cancellare il comando e ritornare a

prima della chiamata a `execute()`. L'oggetto `ConcreteCommand` chiama le operazioni sul suo Receiver per portare a termine la richiesta.

Le conseguenze di questo pattern sono estremamente positive per l'architettura del software. Il vantaggio più evidente è il forte **disaccoppiamento**: il pattern separa nettamente l'oggetto che invoca l'azione (invoker) da quello che sa effettivamente come eseguirla (receiver). Poiché i **comandi** diventano oggetti veri e propri, possono essere manipolati, passati come **parametri** ed estesi esattamente come gli altri oggetti. Inoltre, si possono aggregare comandi semplici per creare un comando composto, noto anche come `MacroCommand`. In generale, questi comandi composti non sono altro che un'istanza di un altro design pattern ben noto, il design pattern **Composite**. Infine, l'architettura risulta estremamente aperta all'estensione: è facilissimo aggiungere nuovi comandi al sistema, poiché non è richiesto alcun cambiamento alle classi preesistenti.

Implementare un `Command` offre un'ampia gamma di possibilità. Un comando potrebbe essere molto semplice e limitarsi a definire un legame diretto tra un ricevente e l'azione da eseguire, oppure potrebbe implementare tutta la logica al suo interno, facendo così del tutto a meno del ricevente.

i `Command` possono supportare il meccanismo di **undo**, ma per farlo devono possedere la capacità di rendere reversibile la loro stessa esecuzione. Per ottenere questo, la classe `ConcreteCommand` potrebbe aver bisogno di conservare uno stato aggiuntivo. Allo stesso tempo, il ricevente deve fornire tutte le operazioni necessarie per permettere al command di farlo ritornare allo stato precedente l'esecuzione. Se l'applicazione richiede un solo livello di **undo**, al command basterà conservare l'ultima attività eseguita; se invece si vogliono implementare più livelli di **undo** (e magari anche di **redo**), sarà necessario strutturare il sistema per mantenere una vera e propria lista (storico) dei comandi eseguiti.

Progettare per Preservare

Vediamo ora come applicare strategicamente dei Design Pattern per garantire flessibilità, disaccoppiamento e riuso ottimale del codice all'interno di architetture software complesse.

Istanziazione degli Oggetti e le Dipendenze

Quando si progetta un sistema software, un principio fondamentale per preservare la flessibilità e la manutenibilità è evitare di creare un oggetto specificando esplicitamente la classe.

Un'istanziazione esplicita orienta il sistema verso una specifica **implementazione** e non verso un'**interfaccia**, rendendo molto più complicati cambiamenti futuri.

Gli oggetti dovrebbero essere creati in modo **diretto**, avvalendosi di design pattern creazionali specifici per questo scopo come il **Factory Method** o il **Prototype**.

Allo stesso modo, le dipendenze da operazioni specifiche sono da evitare, in quanto vincolano la logica del programma a un unico rigido modo di soddisfare una determinata richiesta.

Rimuovere queste dipendenze permette di **modificare dinamicamente** il modo in cui le richieste vengono soddisfatte sia a compile-time che a run-time. Per ottenere questo disaccoppiamento logico tra mittente e destinatario della richiesta, si rivelano particolarmente utili pattern comportamentali come **Chain of Responsibility** e **Command**.

Isolamento da Piattaforme Esterne e Rappresentazioni interne

Un altro aspetto architetturale cruciale riguarda le dipendenze dalle piattaforme hardware e software. L'uso intensivo di API esterne rende molto più difficile il porting del sistema software verso altri ambienti e complicando l'aggiornamento sulla piattaforma stessa.

La progettazione deve **limitare il più possibile** i legami e le dipendenze dalla piattaforma sottostante, sfruttando l'astrazione garantita da pattern come **Factory** e **Bridge**.

Analogamente anche le dipendenze dalle rappresentazioni o dalle implementazioni interne (ovvero il modo strutturale in cui un oggetto è rappresentato in memoria, conservato o implementato logicamente) rendono le classi client estremamente vulnerabili.

Se la struttura interna dell'oggetto cambia, le classi client subiscono **modifiche obbligate**. Nascondere le informazioni di implementazione alle

classi client evita disastrosi cambiamenti a cascata, adoperando pattern in grado di incapsulare i dettagli come **Factory**, **Bridge**, **Memento** e **Proxy**.

Gestione degli Algoritmi e il Vantaggio del Lasco Accoppiamento

Durante il ciclo di vita e lo sviluppo del software, gli algoritmi vengono sistematicamente estesi, ottimizzati o del tutto rimpiazzati per favorire il riuso.

Se gli oggetti dipendono da essi, ogni modifica all'algoritmo causerà cambiamenti agli oggetti stessi. È necessario **isolare architettonicamente gli algoritmi** soggetti a cambiamenti tramite l'uso mirato di pattern come **Builder**, **Iterator**, **Strategy/State**, **Template Method** e **Visitor**.

A livello di design generale mantenere uno stretto accoppiamento fra le varie classi tende inesorabilmente a generare sistemi monoliti e difficili da scalare.

Le classi fortemente accoppiate tra loro risultano ardue sia da modificare che da riutilizzare singolarmente senza dover comprendere e alterare anche le altre classi. Promuovere un **accoppiamento debole** incrementa la possibilità che una classe possa essere riutilizzata da sola rende il sistema più facile da comprendere, modifica ed estendere. I pattern che facilitano questo disaccoppiamento includono **Factory**, **Bridge**, **Chain of Responsibility**, **Command**, **Facade**, **Mediator** e **Observer**.

Composizione vs Ereditarietà e Gestione del Codice Chiuso

Per quanto concerne l'estensione delle funzionalità del codice, l'uso dell'ereditarietà tramite sottoclassi rappresenta un riuso di tipo "scatola bianca". Questo approccio può condurre a un grande numero di classi create esclusivamente per gestire le nuove estensioni.

Richiede quindi una profonda comprensione della superclasse e fare l'override di un'operazione potrebbe richiederne un altro a catena portando un aumento esponenziale di classi. È molto meglio **limitare il numero di livelli** di una gerarchia di classi e **preferire la composizione di oggetti** rispetto all'ereditarietà. Molti design pattern favoriscono questo tipo di progettazione, permettendo di personalizzare le funzionalità definendo una sola sottoclasse e componendo dinamicamente le sue istanze con altre, come avviene utilizzando **Bridge**, **Chain of Responsibility**, **Composite**, **Decorator**, **Observer** e **Strategy**.

Vi sono anche scenari pratici in cui risulta impossibile modificare classi esistenti. Questo accade quando il codice sorgente non è disponibile

(librerie di terze parti) o quando un intervento richiederebbe modificare troppe classi dipendenti.

In questi specifici casi di blocco, la progettazione si affida a pattern risolutivi che permettono di estendere o adattare i comportamenti senza alterare il codice originale, come come **Adapter**, **Decorator** e **Visitor**.

Introduzione alla Manutenzione e Costi del Software

Il processo di manutenzione del software è definito come l'**introduzione di modifiche** a un prodotto dopo la consegna al cliente. Molto spesso questo termine viene confuso col termine di **evoluzione**, sebbene quest'ultimo venga inteso come un singolo passo all'interno di un processo di manutenzione più ampio che include il rilascio di patch e la rimozione del sistema.

Da un punto di vista statistico e aziendale, i costi di manutenzione assorbono una quota enorme, stimata tra il 67% e l'80% dei costi totali del software. Questi cambiamenti si possono classificare in 4 gruppi:

- **correttivi**: rimozione degli errori (17%);
- **adattivi**: adeguare il software a un nuovo ambiente (18%);
- **perfettivi**: miglioramento del sistema e aggiunta di funzionalità (60%);
- **preventivi**: modifiche interne mirate a prevenire problemi futuri (5%).

La categoria di cambiamenti perfettivi copre la maggior parte delle modifiche che si svolgono all'interno di un software, per questo è buona pratica anticipare la necessità di cambiamenti già durante la fase di design, avvalendosi di tecniche come la parametrizzazione e l'incapsulamento.

Dinamiche di Evoluzione e Leggi di Lehman

Le dinamiche di evoluzione descrivono i processi reali di cambiamento che interessano un sistema e poggiano su solide **basi empiriche**: studi condotti da Lehman e Belady. Queste ricerche, confermate anche da studi più recenti, hanno analizzato grandi sistemi sviluppati da grandi aziende, portando alla formulazione delle note "**Leggi di Lehman**":

1. **Cambiamento continuo [1974]**: i sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti.
2. **Aumento della complessità [1974]**: quando un sistema evolve, la sua struttura aumenta di complessità, a meno che il lavoro venga svolto per preservare o semplificare la sua struttura.

3. **Auto-regolazione [1974]:** attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti.
4. **Stabilità organizzativa [1978]:** durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo.
5. **Conservazione di familiarità [1978]:** in media, l'incremento di crescita di un sistema tende a rimanere costante o a diminuire.
6. **Continua crescita [1978]:** il contenuto di funzioni di un sistema deve continuamente essere incrementato per mantenere la soddisfazione del cliente.
7. **Diminuzione della qualità [1994]:** la qualità di un sistema diminuisce e non viene rigorosamente gestita e adattata durante i cambiamenti.

È importante notare che l'**applicabilità** di queste leggi è verificata soprattutto per grandi sistemi e organizzazioni, rimane incerto come esse si adattino a prodotti di piccole dimensioni, a sistemi che integrano componenti pre-costruiti (CTOS) o a piccole aziende.

Fattori di Costo, Modelli e Tipi di Modifiche

Il costo della manutenzione è fortemente influenzato da chi la esegue: se il team che ha sviluppato originariamente il software viene coinvolto anche nella fase di manutenzione i **costi si riducono**. Al contrario, se gli sviluppatori non hanno responsabilità contrattuali a lungo termine, non avranno incentivi a fare una progettazione iniziale facilmente modificabile, causando un **degrado della struttura** man mano che si introducono cambiamenti.

Per gestire queste fasi si adottano diversi modelli di manutenzione:

- **Quick-fix:** si basa sull'applicazione di cambiamenti e correzioni in maniera veloce, intervenendo direttamente a livello di codice.
- **Miglioramento iterativo:** prevede che i cambiamenti siano eseguiti in base ad un'analisi accurata del sistema esistente, con lo scopo di controllare la complessità e preservare l'integrità del design.
- **Riuso:** consiste nello stabilire i requisiti del nuovo sistema cercando di riutilizzare il più possibile componenti del vecchio sistema.

Per quanto riguarda le azioni pratiche sul codice, abbiamo tre tipi di modifiche:

- **Re-factoring o Re-structuring:** processo che migliora la struttura interna del codice senza alterarne il comportamento esterno.
- **Reverse engineering:** consiste nell'analizzare un sistema, partendo dal suo interno per estrarre informazioni cruciali sulla sua architettura o sul suo comportamento.
- **Re-engineering:** una vera e propria alterazione del sistema attuata per ricostruirlo e trasformarlo in un'altra forma.

Metriche Tradizionali e Complessità Ciclomatica

Le metriche software definiscono un insieme di misurazioni fondamentali per un sistema software.

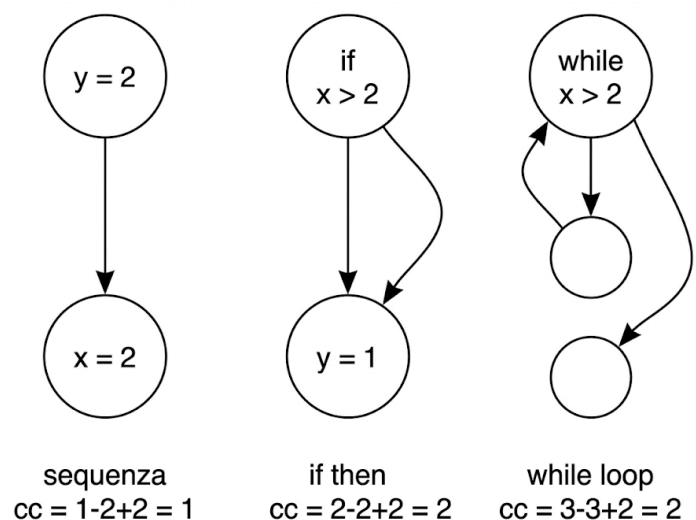
I suoi obiettivi sono:

- **monitorare** il prodotto mentre si costruisce;
- **identificare** i livelli soglia per ciascuna metrica;
- **rimediare** in caso i livelli non siano soddisfacenti.

È importante ricordare che solo gli attributi interni (come la dimensione del codice) possono essere misurati direttamente, attributi esterni (come la qualità) vengono ricavati indirettamente.

Una delle metriche tradizionali più importanti è la **Complessità Ciclomatica**, che valuta la complessità di un algoritmo indicando il numero di percorsi di esecuzione indipendenti.

È un dato molto importante perché corrisponde al numero minimo di **test necessari** per valutare un algoritmo. La formula per calcolare la CC partendo da un grado del flusso di controllo è: $\text{archi} - \text{nodi} + 2$. Ad esempio, un blocco di istruzioni in semplice "sequenza" ha una complessità ciclomatica pari a 1 (1 arco - 2 nodi + 2 = 1), richiedendo quindi un solo test. Una struttura condizionale "if-then" ha invece una CC pari a 2 (2 archi - 2 nodi + 2 = 2), richiedendo due test per coprire entrambi i rami logici. Allo stesso modo, un ciclo "while loop" presenta una CC di 2 (3 archi - 3 nodi + 2 = 2).



Altre metriche tradizionali riguardano la dimensione del software, calcolata in **LOC** (Line of Code) o **NCNB** (Non Comment Non Blank) e la **Comment Percentage**, che valuta la proporzione dei commenti rispetto alle linee di codice, il valore ottimale si attesta intorno il 30%.

Metriche CK per i Sistemi a Oggetti

Quando si analizzano sistemi sviluppati con il paradigma orientato agli oggetti, si ricorre alla suite di Chidamber & Kemerer, nota come metriche CK. Tra le quali troviamo:

- **WMC** (weighted Method per Class): rappresenta la somma delle complessità dei metodi per classe. Se tutti i metodi pesano allo stesso modo, corrisponde al semplice conteggio dei metodi. Un valore elevato di WMC indica che la classe ha un'alta specificità e richiederà un grande sforzo di manutenzione, diminuendone di conseguenza la riusabilità.
- **DIT** (Depth of Inheritance Tree): misura la profondità dell'albero di ereditarietà, calcolando il numero massimo di livelli che separano una classe dalla radice gerarchica. Un alto DIT favorisce il riuso del codice grazie all'ereditarietà, ma rende il sistema complessivamente più complesso e il comportamento della specifica classe molto più difficile da comprendere, vista l'alta quantità di metodi ereditati.
- **NOC** (Number of Children of a Class): conta il numero di sottoclassi dirette che derivano da una classe. Un NOC elevato è un indicatore positivo per il riuso, evidenziando una forte influenza di tale classe sull'intero sistema.
- **CBO** (Coupling Between Object Classes): quantifica le dipendenze, ovvero il numero di altre classi con cui una interagisce. Un CBO alto è un campanello di allarme: forte dipendenza, scarsa possibilità di riuso.
- **RFC** (Response For a Class): conta il numero totale di metodi che potrebbero essere eseguiti in risposta alla ricezione di un messaggio. Se questo valore è alto, ci si trova davanti a una grande complessità che si traduce in una notevole difficoltà nella fase di comprensione e di testing.
- **LCOM** (Lack of Cohesion of Methods): valuta la mancanza di coesione interna di una classe. Un basso valore di LCOM indica un buon design, che la classe è semplice e ha una singola responsabilità ben definita, aumenta le possibilità di riuso. L'LCOM viene calcolato attraverso una formula che mette in relazione il numero di attributi (campi), il numero di metodi e quanti di questi metodi utilizzano i vari campi.

$$LCOM = 1 - \frac{\sum_i^a m_{Ai}}{n \cdot m}$$

Con 3 campi e 3 metodi
 - ciascun metodo usa 3 campi, allora $(3+3+3)/(3 \cdot 3) = 1$
 - ciascun metodo usa 2 campi, allora $(2+2+2)/(3 \cdot 3) = 2/3$
 - un solo metodo usa i 3 campi, allora $(1+1+1)/(3 \cdot 3) = 1/3$

Problema dei Difetti di un Software

Il problema dei difetti nel software è molto comune e rappresenta delle sfide cruciali. Per scoprire che un programma contiene un difetto non basta semplicemente guardare il codice ma bisogna confrontare il comportamento del software con la sua **specificità**. La specifica descrive esattamente cosa il programma dovrebbe fare, di conseguenza, quando il software mostra un comportamento anomalo, ci sta comunicando la presenza di difetti che non dovrebbero passare inosservati.

Verifica e Validazione (V&V)

Per gestire la problematica dei difetti si ricorre al processo di Verifica e Validazione, il cui obiettivo è assicurare che il sistema soddisfi i reali bisogni dei suoi utenti.

La **verifica** risponde alla domanda “stiamo costruendo il prodotto nel modo giusto?”, accertandosi che il sistema sia conforme alle sue specifiche tecniche.

La **validazione** (o convalida) risponde alla domanda “stiamo costruendo il giusto prodotto?”, ovvero controlla che il sistema faccia effettivamente ciò che l'utente ha richiesto. Questo processo ha l'obiettivo di scoprire i difetti e valutare l'usabilità in situazione operativa e dovrebbe essere applicato in ogni singola fase dello sviluppo.

Classificazione dei Difetti

I difetti possono essere raggruppati in base alla fase di sviluppo in cui sorgono:

- **Difetti di specifiche:** si presentano quando la descrizione di ciò che il prodotto fa è ambigua, contraddittoria o imprecisa.
- **Difetti di progettazione:** sono causati da componenti o interazioni progettate in modo non corretto. Le cause possono riguardare algoritmi (ad esempio divisione per zero), strutture dati (ad esempio campo mancante, tipo sbagliato) o interfacce dei moduli (parametri di tipo inconsistente). A volte è difficile distinguerli dai difetti di codice.
- **Difetti di codice:** sono errori derivanti dall'implementazione, dovuti a poca comprensione del progetto o dei costrutti del linguaggio (ad esempio overflow, conversione del tipo, priorità delle operazioni aritmetiche, variabili non inizializzate o non usate).
- **Difetti di test:** persino i casi di test e i piani per i test possono contenere difetti.

Testing vs Debugging

È fondamentale chiarire la differenza tra le due attività.

La **fase di test** ha come obiettivo rivelare l'esistenza di difetti in un programma, può **dimostrare la presenza di errori**, ma non la loro assenza. Un test ha successo se scopre uno o più errori e dovrebbe essere condotto insieme alle verifiche sul codice statico.

La **fase di debugging** si riferisce alla **localizzazione degli errori**, procedendo con il formulare ipotesi sul comportamento del programma per poi verificarle e trovare i difetti.

Definizioni Fondamentali

- **Dati di test:** sono i dati di input che sono stati scelti specificatamente per testare il sistema.
- **Casi di test:** comprendono i dati di input per il sistema e gli output stimati per tali input nel caso in cui il sistema operi secondo le sue specifiche. Gli input includono non solo parametri, ma anche file, eccezioni e lo stato del sistema (le condizioni di esecuzione). Devono essere sviluppati sia per condizioni valide che non valide.
- **Test suite:** indica un insieme di casi di test.

Impraticabilità di Test Esaustivi e le Priorità

Solo un test esaustivo può mostrare se un programma è privo di difetti, ma all'atto pratico i test esaustivi sono **impraticabili**. Ad esempio, una funzione che prende in ingresso 2 interi (**int**), per essere testata esaustivamente dovrebbe essere eseguita $2^{32} * 2^{32}$ volte. Se la funzione esegue in $1ns$ occorrerebbero circa 600 anni per terminare!

Per questo si usano delle **priorità**: i test dovrebbero mostrare le **capacità del software** più che i singoli componenti, testare le vecchie funzionalità è più importante delle nuove, e le situazioni tipiche sono più importanti delle situazioni limite. Inoltre, i test devono essere ripetibili e riusabili (test di regressione).

Strategie di test

Esistono diversi approcci per eseguire dei test, ne vediamo due usati per rendere la fase di test più efficace:

- **Test Black-box:** considera il sistema una scatola nera. Viene effettuato senza conoscere la struttura o il codice del sistema, i casi di test sono progettati sulla base della descrizione del sistema (documento di specifiche) ed è possibile studiarli nelle fasi iniziali.

- **Test White-box** (glass-box o strutturale): si focalizza sulla struttura del software, per cui bisogna avere a disposizione il codice sorgente o una sua rappresentazione tramite pseudo-codice.

Posizionamento in Classi Equivalenti

Nel test black-box, un buon modo per selezionare gli input è il **partizionamento in classi equivalenti**. I dati di input e i risultati si raggruppano in classi (partizioni) in cui ci si aspetta che il programma effettui elaborazioni simili per ciascuna membro. Testare un membro di una classe equivale a testare ciascun altro valore della stessa classe, riducendo la necessità di test esaustivi.

- I casi di test dovrebbero essere scelti da ciascuna partizione. Es. Una funzione può prendere in input solo numeri da 4 a 20
 - Partizioni: numeri <4; numeri tra 4 e 20; numeri >20
 - Dati di test da scegliere: 3, 4, 12, 20, 21 (alcuni numeri sono scelti per i test di confine: boundary)

Chi fa il test deve considerare sia classi valide che non valide (input inaspettati o errati).

Test del Percorso

Questo approccio strutturale mira ad assicurare che ogni percorso all'interno del programma sia eseguito almeno una volta. È utile rappresentare il programma tramite un **grafo di flusso**. La formula per calcolare il numero di test necessari è la Complessità Ciclomatica. Con questo test tutti i percorsi sono eseguiti ma non tutte le loro combinazioni.

Test Sotto Stress

Eseguire il sistema **oltre il massimo carico** previsto consente di rendere evidenti difetti pesanti. Il sistema, anche oltre i limiti, non dovrebbe fallire in modo catastrofico. Questi test indagano su **perdite di servizio o di dati** inaccettabili e sono particolarmente rilevanti per i sistemi distribuiti dipendenti dalle condizioni della rete. I risultati possono poi servire per completare le specifiche.

Copertura del Codice (Code Coverage)

La copertura è una misura in percentuale della parte del codice sorgente eseguita da una test suite, utile per capire **quando smettere di fare test** e per **evidenziare se manca qualcosa**. Il programma viene diviso in unità e si calcola numero di unità eseguite/numero totale di unità.

Avere una copertura del 100% non significa non avere difetti, poiché esso dipende dai dati di input scelti. Esistono vari criteri:

- **Copertura di funzioni:** percentuale di funzioni eseguite.
- **Copertura di costrutti o statement.**
- **Copertura di archi del grafo di controllo.**
- **Copertura di rami:** ad esempio blocchi if ed else.
- **Copertura di condizioni:** espressioni booleane valutate sia true che false.
- **Copertura di decisioni:** combinazione di funzioni e rami, dove ogni punto di ingresso/uscita è eseguito e ogni espressione booleana ha dato tutti i risultati possibili.

Trend di Difetti Scoperti

La metrica *bug trend* misura la frequenza con cui i difetti vengono trovati. Quando questa frequenza tende a zero significa che non ci sono più difetti o che c'è un drammatico aumento dei costi di ricerca. Le pratiche standard per mantenere alta la qualità prevedono l'esecuzione frequente dei test lavorando con nuove versioni (nightly build) e l'uso di test di regressione per prevenire la ricomparsa di vecchi difetti, utilizzando il coverage e il bug trend come condizioni di stop.