UniCT – DMI

Ingegneria del Software

ANNO ACCADEMICO 2019/20

Autori:

Luigi Seminara - Lemuel Puglisi

Revisione:

Alessio Tudisco

Sommario

Introduzione

FAQ

Cos'è il software?
Un software è un programma per computer con una documentazione associata. Un software può essere sviluppato per un particolare cliente o per un mercato generale, come ad esempio le applicazioni per smartphone.

Quali sono gli attributi di un buon software?
Un buon software dovrebbe fornire all'utente le funzionalità e le prestazioni richieste e inoltre essere mantenibile, affidabile e utilizzabile.

Cos'è l'ingegneria del software e come si differenzia dall'informatica?
L'ingegneria del software è una disciplina ingegneristica concepita con tutti gli aspetti della produzione del software: dall'ideazione iniziale al funzionamento e alla manutenzione. L'informatica si concentra su teoria e fondamenti; l'ingegneria del software si occupa delle funzionalità dello sviluppo di sistemi basati su computer: hardware, software e ingegneria dei processi.

Quali sono le attività fondamentali dell'ingegneria del software?
Specifiche del software, sviluppo del software, convalida del software ed evoluzione del software.

Quali sono i costi di ingegneria del software?
Circa il 60% dei costi del software sono costi di sviluppo, il 40% sono costi di test. Per i software personalizzati, i costi di evoluzione spesso superano i costi di sviluppo.

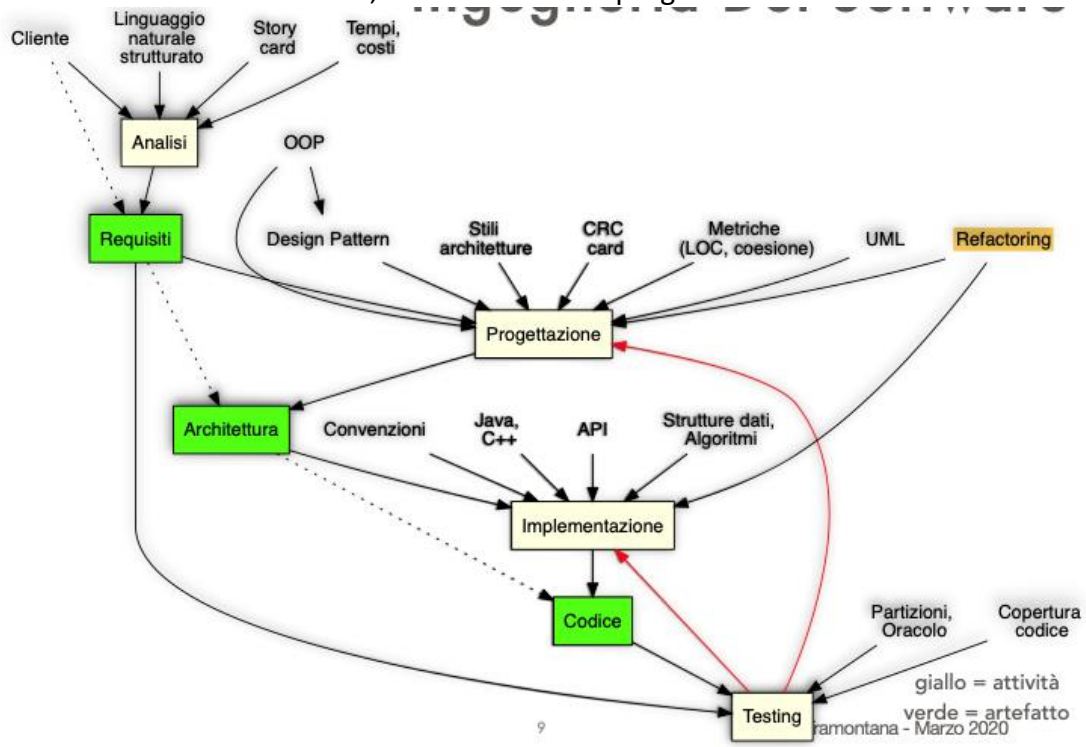
Quali sono le migliori tecniche e metodi di ingegneria del software?
Non ci sono metodi e tecniche che si possono applicare universalmente: tecniche diverse sono appropriate per diversi tipi di sistema. Ad esempio, i giochi dovrebbero sempre essere sviluppati utilizzando una serie di prototipi, mentre i sistemi di controllo critici per la sicurezza richiedono lo sviluppo di una specifica completa e analizzabile.

Ingegneria del software

L'ingegneria del software è una disciplina ingegneristica concepita con tutti gli aspetti della produzione del software dall'ideazione iniziale al funzionamento e alla manutenzione.
In questa definizione ci sono due parole chiavi:

1. **Disciplina ingegneristica.** Gli ingegneri del software applicano teorie, metodi e strumenti in modo appropriato, riconoscono inoltre che devono lavorare all'interno di vincoli organizzativi e finanziari e devono cercare soluzioni all'interno di tali vincoli.
2. **Tutti gli aspetti della produzione del software.** L'ingegneria del software non riguarda solo i processi tecnici di sviluppo del software. Include anche attività come la gestione dei progetti software e lo sviluppo di strumenti, metodi e teorie a supporto dello sviluppo del software.

Definizione di artefatto
In informatica, e in particolare in ingegneria del software, si definisce artefatto un sottoprodotto che viene realizzato durante lo sviluppo software. Sono artefatti i casi d'uso, i diagrammi delle classi, i modelli UML, il codice sorgente e la documentazione varia, che aiutano a descrivere la funzione, l'architettura e la progettazione del software.



Fasi dell'ingegneria del software

1. **Analisi e definizione dei requisiti.** I servizi, i vincoli, gli obiettivi del sistema sono stabiliti consultando gli utenti del sistema. Vengono quindi definiti in dettaglio per fungere da specifiche di sistema. In questa fase viene utilizzato un linguaggio naturale strutturato e vengono definiti i tempi e i costi del software. La documentazione creata rappresenta un artefatto denominato "requisiti del software".
2. **Progettazione e architettura del software.** In tale fase si assegnano i requisiti a sistemi hardware o software. Stabilisce un'architettura di sistema globale. La progettazione del software prevede l'identificazione e la descrizione delle astrazioni fondamentali del sistema software e delle loro relazioni. Con architettura si definiscono i componenti dell'architettura stessa e quali sono i ruoli di ogni componente (una componente è un insieme di classi che hanno una determinata funzione). L'architettura rappresenta l'artefatto di questa fase.
3. **Implementazione.** Durante questa fase, l'architettura precedentemente descritta viene realizzata attraverso dei linguaggi di programmazione, strutture dati e librerie, producendo del codice sorgente. Quest'ultimo rappresenta l'artefatto della fase.
4. **Testing.** Una volta scritto del codice è necessario testarlo affinché rispetti le richieste del cliente. I test ci permettono di far emergere dei difetti, che durante le precedenti fasi non sono stati individuati, e quindi correggerli. I test vengono scritti al di fuori del codice implementato, ovvero è codice esterno che valuta metodo per metodo la correttezza dell'output.

5. **Manutenzione correttiva-adattiva e manutenzione evolutiva.** Durante il ciclo di vita del software è pressoché certo che vi sarà bisogno di uno o più interventi di manutenzione scaturiti dalla scoperta di un difetto o la richiesta di un potenziamento delle funzionalità. Un software progettato bene è caratterizzato da una facile modificabilità che permette di incorporare nuove funzionalità in tempi rapidi e con meno sforzo rispetto a quello iniziale di sviluppo.

Qualità del software

Le tecniche dell'ingegneria del software cercano di produrre sistemi software entro i costi e i tempi preventivati e con qualità accettabile. Le principali caratteristiche di un software sono:

1. **Correttezza:** Il software deve attenersi alle richieste dell'utente. Per valutare la correttezza di un software è necessario scrivere dei test sulla base delle specifiche ricevute.
2. **Accettabilità:** Il software deve essere accettabile per il tipo di utenti per cui è progettato. Ciò significa che deve essere comprensibile, utilizzabile e compatibile con altri sistemi che usano.
3. **Affidabilità e sicurezza:** Un software affidabile non dovrebbe causare danni fisici o economici in caso di guasto del sistema. Il software deve essere sicuro in modo che gli utenti malintenzionati non possano accedere o danneggiare il sistema. Il concetto di affidabilità è legato al numero di guasti e malfunzionamenti in una certa unità di tempo.
4. **Efficienza:** Il software non dovrebbe fare spreco delle risorse di sistema come memoria e cicli del processore. L'efficienza quindi include reattività, tempo di elaborazione, utilizzo delle risorse, ecc.
5. **Manutenibilità:** Il software dovrebbe essere scritto in modo tale da poter evolversi per soddisfare le mutevoli esigenze dei clienti. Questo è un attributo critico perché la modifica del software è un requisito inevitabile di un ambiente aziendale in evoluzione.

Spaghetti code

Lo "spaghetti code" è un anti-pattern che indica un codice monolitico: un codice in cui vengono eseguite tutte le operazioni in un unico flusso, caratterizzato da una difficile comprensione e modifiche che si ripercuotono su gran parte del codice.
Tale codice non segue il paradigma della programmazione ad oggetti con la conseguente impossibilità di riusarlo o verificarne la correttezza.

Programmazione ad Oggetti (OOP)

Per un uso ottimale del paradigma OOP devono verificarsi i seguenti punti:

- Ogni metodo ha una sola piccola responsabilità;
- Il flusso di chiamate ai metodi è indipendente dai singoli algoritmi;
- Posso riusare (richiamandoli) i servizi offerti dai metodi.

Paradigma Command e Query

Il **paradigma Command e Query** può essere utilizzato per poter migliorare la OOP:

- I metodi *Query* restituiscono un risultato e non modificano lo stato del sistema.
- I metodi *Command* (o modificatori) cambiano lo stato del sistema ma non restituiscono un valore.

Ulteriori concetti iniziali

Miglioramenti sul codice

Le principali accortezze da seguire per poter produrre del buon codice o migliorarne uno già esistente sono le seguenti:

1. Separazione delle varie operazioni: ogni operazione ha un suo metodo. Si ottengono così metodo atomici o a singola responsabilità che eseguono solo un determinato compito, ciò permette di verificare la correttezza delle operazioni e riusarle.
2. Costruzione di astrazioni: si implementano metodi di livello più basso e metodi di livello più alto, che richiamano i precedenti, lo stesso si fa per le classi.
3. Impiego del paradigma "Command & Query": vengono distinte le operazioni che cambiano lo stato (*command*) da quelle che restituiscono valori (*query*) in metodi diversi. Un esempio possono essere i get() e set() degli attributi.
4. Lo stato dell'oggetto diventa osservabile, attraverso metodi opportuni, in modo da poter fare i test.

Test

Il testing ha lo scopo di assicurare che un programma soddisfi i requisiti richiesti dall'utente e correggere difetti che non sono stati individuati precedentemente. Diremo quindi che un codice è corretto se è stato possibile eseguire un test su di esso e risulti passato.
Quando si esegue il test di un software, si provano a fare due cose:

1. Dimostrare allo sviluppatore e al cliente che il software soddisfa i requisiti.
2. Trovare input o sequenze di input in cui il comportamento del software è errato o non conforme alle sue specifiche a causa di difetti. La correzione di tali difetti permette di sradicare comportamenti indesiderati del sistema come arresti anomali, interazioni indesiderate con altri sistemi, calcoli errati e corruzione di dati.

I test sono così caratterizzati:

1. Ogni test deve essere **indipendente** dagli altri: il risultato di un test non deve dipendere da quello di un altro test, inoltre il risultato del test non deve essere influenzato dall'ordina di esecuzione. Prima di ogni test si ritorna allo stato iniziale dell'applicativo.
2. I test devono essere **auto-valutanti**: il test deve ritornare un risultato che esprima l'esito della propria esecuzione.

Inoltre, i test documentano le condizioni sotto le quali il codice funziona e protegge il quest'ultimo da modifiche che potrebbero alterarne il corretto funzionamento.

Test Driven Development (TDD)

La tecnica di testing denominata **Test Driven Development (TDD)** prevede di scrivere prima tutti i test, in base ai requisiti da soddisfare, e poi il codice, ma solo lo stresso necessario per soddisfare il test. Si continua con questa modalità fino a quando non saranno soddisfatti tutti i test implementati, ovviamente il set di test deve essere così completo da rispecchiare i requisiti iniziali.

Principio di Singola Responsabilità (SRP)

I compiti sono suddivisi su vari metodi (e quindi su più classi), questo permette di ottenere coesione del codice, ovvero tutto ciò che è in una classe svolge un unico compito.
Principio di Singola Responsabilità (SRP)

Ogni classe deve avere una singola responsabilità in modo che il codice sia comprensibile e la classe possa essere riusata. Questo principio è fondamentale anche per l'ereditarietà.
Se si scrive un commento su ciò che la classe deve fare e si scrivono virgole o congiunzioni, allora probabilmente la SRP non è rispettata.

Astrazione

Quando si costruisce una classe si sta creando un'astrazione utile per la funzionalità del software. Il nome che si dà a classi e metodi è estremamente importante poiché ne descrive l'obiettivo.

Cos’è il refactoring

Il refactoring

Un precetto fondamentale dell'ingegneria del software tradizionale è la progettazione per il cambiamento: anticipare le future modifiche al software e progettarlo in modo che queste modifiche possano essere facilmente implementate. Tuttavia nel tempo si è scartato questo principio a favore dello sviluppo incrementale sulla base del fatto che la progettazione per il cambiamento è spesso uno sforzo sprecato, in quanto le modifiche previste non si materializzano quasi mai oppure le richiede di modifica ricevute risultano completamente diverse.

Un problema fondamentale dello sviluppo incrementale è che i cambiamenti locali tendono a degradare la struttura del software: ulteriori modifiche al software risultano sempre più difficili da implementare, il codice viene spesso duplicato, parti del software vengono riutilizzate in modo inappropriato e vi è l'introduzione di un debito tecnico.

Debito tecnico

Il debito tecnico è una metafora usata per descrivere le possibili complicazioni che subentrano in un progetto software, qualora non venissero adottate adeguate azioni volte a mantenerne bassa la complessità. Lo sforzo per recuperare un progetto sviluppato senza una corretta metodologia può aumentare anche considerevolmente nel tempo, se non si interviene tempestivamente.

Il refactoring consiste nel cercare e implementare possibili miglioramenti nel codice, in quanto raramente vi è una progettazione perfetta fin dall'inizio, così da migliorare la struttura e la leggibilità del software, evitando il deterioramento strutturale che si verifica naturalmente quando il software viene modificato. Spesso, dopo aver effettuato un refactor, la dimensione del codice si riduce, le strutture più complesse si trasformano in strutture più semplici e facili da manutenere.

Possiamo definire il "refactoring" come un 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.

Quando il refactoring fa parte del processo di sviluppo, il software risulta essere spesso di facile comprensione e modifica quando vengono proposti nuovi requisiti, inoltre promuove una comprensione profonda della code base.

Come si fa refactoring?

Una ottima pratica è quella di affiancare testing e refactoring: prima si scrivono i test, poi si effettua il refactoring, così da essere certi che le modifiche soddisfino ancora i requisiti e che non alterino il comportamento desiderato.

Tecnica di refactoring: Estrai metodo

Un frammento di codice può essere raggruppato come segue:

```
void printOwing() {
  printBanner();

  // Print details.
  System.out.println("name: " + name);
  System.out.println("amount: " + getOutstanding());
}
```

La tecnica consiste nel fare diventare quel frammento un metodo il cui nome spiega il suo scopo:

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails(double outstanding) {
  System.out.println("name: " + name);
  System.out.println("amount: " + outstanding);
}
```

Più righe vengono trovate in un metodo, più difficile è capire cosa fa il metodo. Questa è la ragione principale di questo refactoring.

Motivazioni

- Cercare metodi lunghi o codice che necessita di commenti per essere comprensibile;
- Ridurre la lunghezza dei metodi, poiché metodi piccoli sono con più probabilità utilizzabili richiamandoli da altri metodi;
- Scegliere un buon nome per i metodi piccoli, che esprime ciò che il metodo fa;
- I metodi di alto livello, quelli che invocano i metodi piccoli, sono più facili da comprendere, sembrano essere costituiti da una sequenza di commenti;
- Fare overriding (ridefinire nella sottoclasse) è più semplice se si hanno metodi piccoli.

Meccanismi

- Creare un nuovo metodo il cui nome comunica l'intenzione del metodo;
- Copiare il codice estratto dal metodo sorgente al nuovo metodo;
- Se il codice estratto fa uso di variabili locali nel metodo sorgente, far diventare tali variabili parametri del nuovo metodo;
- Se alcune variabili sono usate solo all'interno del codice estratto farle diventare variabili temporanee del nuovo metodo;
- Se una variabile locale al metodo sorgente è modificata dal codice estratto, tentare di trasformare il metodo estratto in un metodo query e assegnare il risultato alla variabile locale del metodo sorgente;
- Sostituire nel metodo sorgente il codice estratto con una chiamata al metodo nuovo.

Tecnica di refactoring: Sostituisci Temp con Query

Problema: Una variabile temporanea (temp) è usata per tenere il risultato di un'espressione.

```
double calculateTotal() {
  double basePrice = quantity * itemPrice;
  if (basePrice > 1000) {
    return basePrice * 0.95;
  }
  else {
    return basePrice * 0.98;
  }
}
```

Soluzione: Estrarre l'espressione ed inserirla in un nuovo metodo query, il cui nome indica l'espressione che esegue. Sostituire tutti i riferimenti a temp con la chiamata al metodo che incapsula l'espressione. Il nuovo metodo può essere richiamato anche altrove.

```
double calculateTotal() {
  if (basePrice() > 1000) {
    return basePrice() * 0.95;
  }
  else {
    return basePrice() * 0.98;
  }
}
double basePrice() {
  return quantity * itemPrice;
}
```

Prestazione

Questo refactoring può sollevare la questione se questo approccio possa causare un calo delle prestazioni. La risposta onesta è: sì, poiché il codice risultante può essere gravato dall'interrogazione di un nuovo metodo. Ma con le CPU veloci di oggi e gli eccellenti compilatori, l'onere sarà quasi sempre minimo. Al contrario, il codice leggibile e la possibilità di riutilizzare questo metodo in altri punti del codice del programma - grazie a questo approccio di refactoring - sono vantaggi molto evidenti.

Motivazioni

- Le variabili temp sono temporanee e locali. Possono essere viste solo all'interno del contesto del metodo e per questo inducono ad avere metodi lunghi, perché è il solo modo per raggiungere tali variabili;
- Con la sostituzione effettuata tramite il refactoring, qualsiasi metodo può avere il dato;
- Spesso si effettua questo refactoring prima di Estrai metodo;
- Questo refactoring è facile se Temp è assegnata una sola volta.

Meccanismi

- Cercare una variabile temporanea assegnata una sola volta;
- Dichiarare temp final e compilare per verificare se è assegnata una sola volta;
- Estrarre la parte di destra dell'assegnazione e creare un metodo.

Tecnica di refactoring: Dividi variabile temporanea

Problema: Una variabile temporanea è assegnata più di una volta, ma non è una variabile assegnata in un loop o usata per accumulare valori. [Variabile multifunzione]

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

Soluzione: Si usa una variabile separata per ciascun assegnamento.

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

Motivazioni

- Le variabili temporanee hanno vari usi. Alcuni usi portano ad assegnare temp più volte, es. variabili che cambiano ad ogni passata di un ciclo (queste assegnazioni multiple sono ok).
- Se le variabili temp sono assegnate più di una volta allora hanno più di una responsabilità all'interno del metodo. Ogni variabile dovrebbe avere una sola responsabilità e dovrebbe essere sostituita con una temp per ciascuna responsabilità.
- Usare la stessa temp per due cose diverse confonde il lettore.

Meccanismi

- Cambiare il nome della variabile temp al momento della dichiarazione e della sua prima assegnazione;
- Dichiarare la nuova temp come final;
- Cambiare tutti i riferimenti a temp fino alla sua seconda assegnazione;
- Dichiarare una nuova temp per la seconda assegnazione;
- Compilare e testare;
- Ripetere se necessario per i luoghi in cui alla variabile viene assegnato un valore diverso.

Il paradigma OOP

Classi e oggetti

Modelli ad oggetti

- Descrivono il sistema in termini di classi (OOP)
- Una classe ha attributi ed operazioni comuni ad un set di oggetti
- Vari modelli (e diagrammi) ad oggetti possono essere prodotti
  - Di ereditarietà, aggregazione, interazione
- Pro del modello ad oggetti
  - Mappa naturalmente entità del mondo reale
  - Classi che rappresentano entità del dominio sono *riusabili*
- Contro
  - Entità astratte sono più difficilmente modellabili
  - L'identificazione di classi è un processo difficile che richiede una comprensione profonda del dominio applicativo

Identificazione classi

- Dall'elenco dei requisiti
  - Analisi grammaticale del testo
    - Nomi --> classi o attributi
    - Verbi --> operazioni
  - Individuare oggetti fisici
    - Questi suggeriscono classi corrispondenti
  - Raggruppare in modo coeso operazioni tra loro e dati tra loro
    - Questi gruppi suggeriranno delle classi

Una classe è quindi una astrazione di un concetto. Un insieme di classi formano il vocabolario del dominio del problema.

Un'istanza di una classe prende il nome di un oggetto e ci permette di richiamare i metodi della classe che avranno come dominio l'istanza stessa, una classe può istanziare sé stessa o altre classi, ad ogni classe possono corrispondere 0 o più istanze.
Se la classe possiede metodi statici, questi possono essere invocati senza avere necessariamente una sua istanza.

Le classi sono statiche, quindi la semantica e le relazioni sono fissate prima dell'esecuzione del programma, inoltre l'individuazione delle relazioni fra le varie classi prende il nome di **analisi statica delle relazioni**.

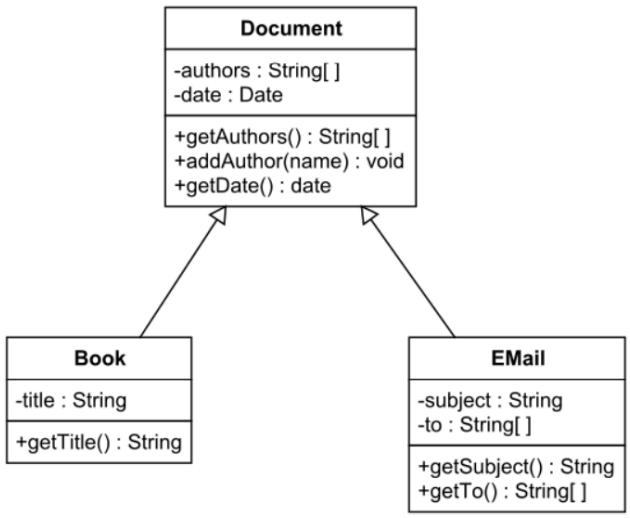
Gli oggetti insieme interagiscono per soddisfare i requisiti del problema. La creazione di un oggetto di una specifica classe avviane attraverso l'operatore *new*, mentre la distruzione, per quanto riguarda Java, avviene quando ogni riferimento all'oggetto viene perso, dando così il permesso al *garbage collector* di deallocare l'oggetto.

Riuso delle classi

Durante lo sviluppo di un software può capitare la necessità di avere due classi simili, magari l'una un caso particolare dell'altra (per esempio classe Macchina e classe Sportiva), in cui varia qualche metodo o attributo.
La soluzione più banale e inefficiente consiste nel ricopiare il codice, pratica da evitare quando possibile.
La migliore soluzione per tale problema è l'utilizzo dell'ereditarietà tra classi.

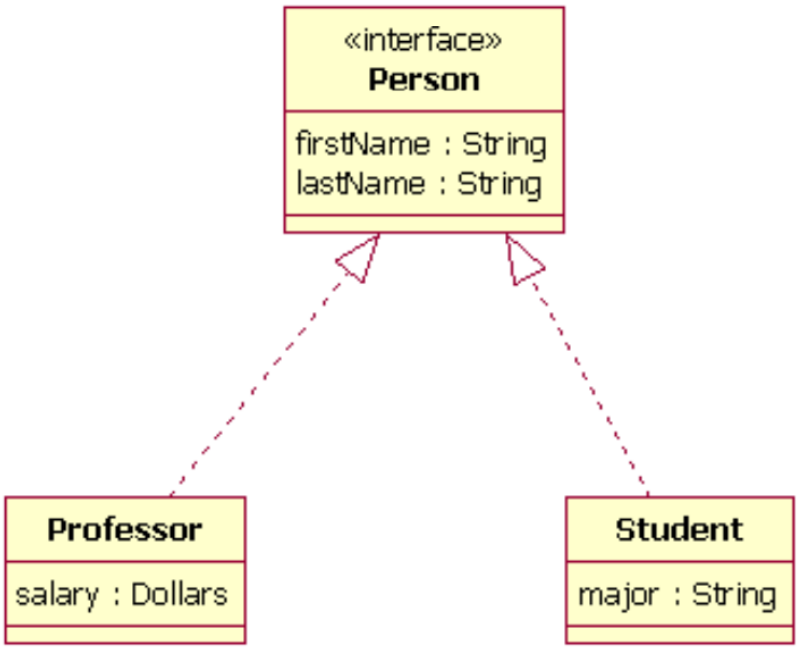
Ereditarietà delle classi

Attraverso l'ereditarietà tra classi è possibile definire una nuova classe indicando solo cosa ha in più o differisce da una classe esistente. La sottoclasse erediterà tutti i metodi e gli attributi della superclasse e potrà utilizzarli come se fossero definiti localmente. Può inoltre ridefinire (overriding) i metodi della superclasse, ma non può eliminarli. Ciò che è private nella superclasse non sarà visibile nella sottoclasse, a differenza degli attributi e metodi definiti con visibilità *protected* o *public*.
Nei diagrammi UML, l'ereditarietà viene indicata tramite una freccia che parte dalla sottoclasse e punta alla superclasse, la cui punta non è riempita di colore.



Interfacce

Una interfaccia è costrutto necessario a fornire uno schema condiviso di una classe. In una interfaccia tutti i metodi sono astratti, ovvero primi di implementazione, bensì sono solo definiti. Elenca le firme dei metodi public. Non vi sono costruttori. Non è possibile istanziare una interfaccia.



Una classe che utilizza una interfaccia deve necessariamente fornire delle implementazioni ai metodi dichiarati nell'interfaccia stessa. Può eventualmente aggiungere altri metodi. L'interfaccia è utile poiché i client sanno cosa possono invocare utilizzando oggetti che utilizzano una determinata interfaccia.
Un oggetto può utilizzare più interfacce.

Nei diagrammi UML, se una classe utilizza una interfaccia è necessario tracciare una freccia tratteggiata che parte dalla classe e punta all'interfaccia, la cui punta non è riempita di colore. Va inoltre indicato che si tratta di una interfaccia anteponendo la key <<interface>>.

Classi astratte

Una classe astratta è una classe parzialmente implementata. Alcuni metodi sono implementati, altri no (e sono definiti come abstract). Un metodo abstract è utile perché:

- Altri metodi della classe possono invocarlo.
- I client si aspettano di poterlo invocare.
- Forza le sottoclassi (concrete) ad implementare il metodo abstract.

La classe astratta non può essere istanziata. Le classi concrete ereditate da una classe astratta devono implementare tutti i metodi astratti, altrimenti saranno considerate anch'esse classi astratte. Nell'UML di una classe astratta va anteposta la key <<abstract>>.

Compatibilità di tipo

L'ereditarietà permette di stabilire una classificazione di tipi. Una sottoclasse è un sottotipo di una superclasse, ovvero ha una relazione "is a" con la superclasse.

Prendiamo come esempio una classe Studente ed una classe Persona. Lo studente è una persona, quindi la prima è sottoclasse della seconda. Sarà possibile quindi cambiare tutte le occorrenze di Persona con Studente, ed il programma compilerà ancora. Non è possibile fare il contrario: una persona non è necessariamente uno studente. Questo vuol dire che una classe studente potrebbe avere metodi aggiuntivi non implementati nella classe persona.

Polimorfismo

Nella programmazione ad oggetti, il polimorfismo è il legame di una singola interfaccia ad entità di tipo differente. L'effetto intrinseco del polimorfismo è dato dai diversi comportamenti che può avere la chiamata ad un metodo comune a due classi che derivano dalla stessa interfaccia.
Esempio: Differenza di comportamento del metodo insert() fra un ArrayList e una LinkedList, entrambe facenti uso dell'interfaccia List.

Late Binding o Dispatch dinamico

Attraverso il polimorfismo è possibile assegnare ad una entità il tipo di una interfaccia o di una superclasse ed assegnare a run-time la sottoclasse. È possibile per le sottoclassi eseguire l'override dei metodi della superclasse.
Questa pratica è accettata dal compilatore, ma come si saprà a runtime a quale procedura saltare? Il compito è assegnato al dispatch dinamico (o run-time binding), che, per ogni metodo comune, esegue tale processo:

1. Controllo se il metodo risiede nella sottoclasse. Se sì, salto al metodo.
2. Altrimenti, eseguo il metodo definito nella superclasse.

Unified Modeling Language (UML)

Cos'è L'UML

L'UML è una famiglia di notazioni grafiche che aiutano a descrivere e progettare sistemi software, in particolare i sistemi software costruiti usando lo stile orientato agli oggetti.

Notazioni UML per classi e interfacce

Notazione UML per classi e interfacce

- Esistono varie notazioni per la classe e l'interfaccia (Java)
- Le notazioni indicano
  - Nome classe; nome classe e attributi; nome classe, attributi e metodi
    - Per la visibilità di attributi e metodi: + public, # protected, – private
    - I nomi delle interfacce sono in corsivo
- Uno stereotipo (es. «interface») indica una variazione di un elemento UML, che ha tutte le proprietà dell'elemento di partenza

Interfaccia

```
public interface Persona {
   public String getNome();
   public boolean isEmpty();
}
```

- Forma degli attributi

```
visibilità nome: tipo                    es. – prezzo : int
visibilità nome: tipo = valore iniziale  es. – prezzo : int = 10
visib nome[molteplicità]: tipo           es. – prezzo[5] : int
```

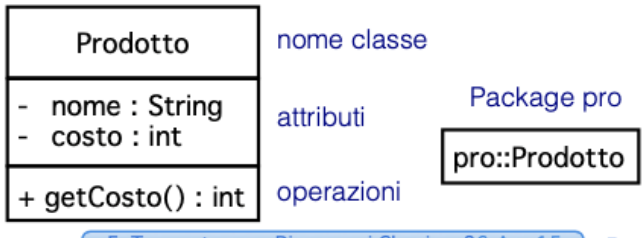
- Forma dei metodi

```
visibilità nome(par: tipo): tipo di ritorno  es. + getCosto() : int
```

  - I metodi statici sono sottolineati

Classe

```
public class Prodotto {
  private String nome;
  private int costo;
  public int getCosto() {
    return costo;
  }
}
```

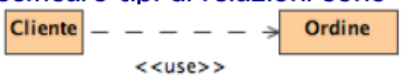


1*Package indica la suddivisione delle classi in cartelle*

Costrutti di estensibilità

Costrutti di estensibilità

- Vincoli (Constraints)
  - Si usano per indicare condizioni o restrizioni e sono rappresentati da espressioni entro parentesi graffe
  - Es. accanto ad un attributo: {il valore è multiplo di 10}
- Stereotipi
  - Si usano per definire nuovi elementi o per specificare tipi di relazioni sono rappresentati da testo entro « »
- Dipendenze e stereotipi predefiniti
  - Associazioni (o dipendenze) tra classi: «use», «call», instantiate», «destroy»
  - «use» indica che un elemento (Ordine) è richiesto per il corretto funzionamento di un altro (Cliente)
    - Es. necessario a compile time perché è un parametro di un metodo
  - Generalizzazioni tra classi: «implements»
- Tagged Value
  - Coppia di stringhe che indica un dato

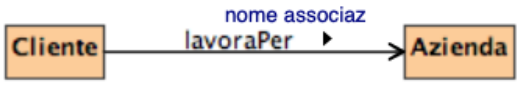


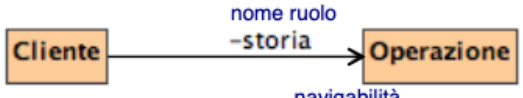
ed il suo valore entro { } Es. dentro una classe: {nome=John}

Diagramma UML delle classi

Diagramma UML delle classi

- Il diagramma delle classi mostra le classi, le loro caratteristiche e le loro relazioni (ereditarietà, implementazione, associazione, uso)
- Una associazione descrive una connessione tra istanze delle classi e specifica
  - Molteplicità: quante istanze di una classe possono essere in relazione con una istanza dell'altra classe (* indica illimitate)
    - Se indicato come attributo, es. `nome[0..1] : Persona`
  - Nome ruolo: usato per attraversare l'associazione (è il nome dell'attributo all'interno della classe di partenza)
  - Navigabilità: verso di attraversamento
  - Nome: spiega l'associazione e il verso

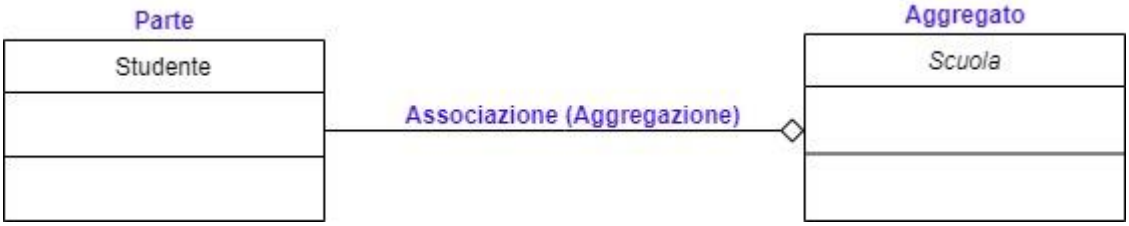




Aggregazione vs Composizione

L'associazione è un concetto abbastanza generico, di fatto *l'aggregazione* e la *composizione* sono casi particolari di associazione.

- Si parla di *aggregazione* o *relazione non* forte quando la classe "figlia" ha senso di esistere anche senza la classe "madre". Ad esempio, supponiamo di avere una classe Scuola (aggregato), ed una classe Studente (parte). Eliminando la Scuola, gli Studenti avrebbero ancora senso, ovvero il ciclo di vita della parte non dipende da quello dell'aggregato. Inoltre può esistere un aggregato privo di part e più aggregati potrebbero far usa della stessa parte.



- Si parla di *composizione* o *relazione forte* quando classe figlia non ha senso di esistere senza la classe madre. Di fatto si dice che la classe madre si compone della classe figlia. Ad esempio, ipotizziamo una classe Macchina (composito) ed una classe Motore (parte): senza la Macchina, il motore non servirebbe più nulla. Il ciclo di vita della parte dipende da quello del composito, ovvero quest'ultimo si occupa della sua creazione e distruzione.

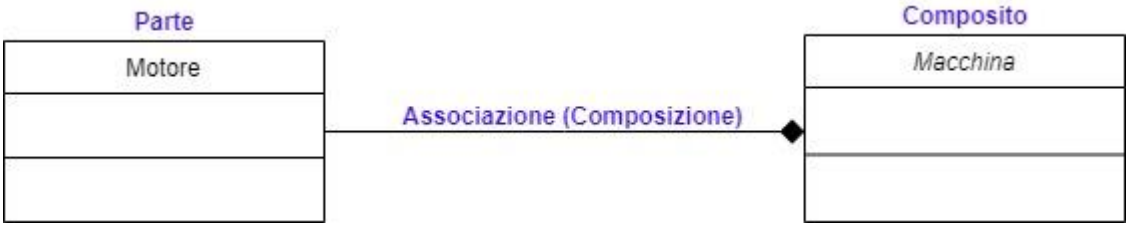


Diagramma di Ereditarietà

Diagramma UML di ereditarietà

- Organizza le classi in una gerarchia
  - Le classi in alto nella gerarchia (superclassi) mostrano le proprietà comuni delle classi in basso (sottoclassi)
  - Le classi ereditano gli attributi e i servizi da una o più super-classi

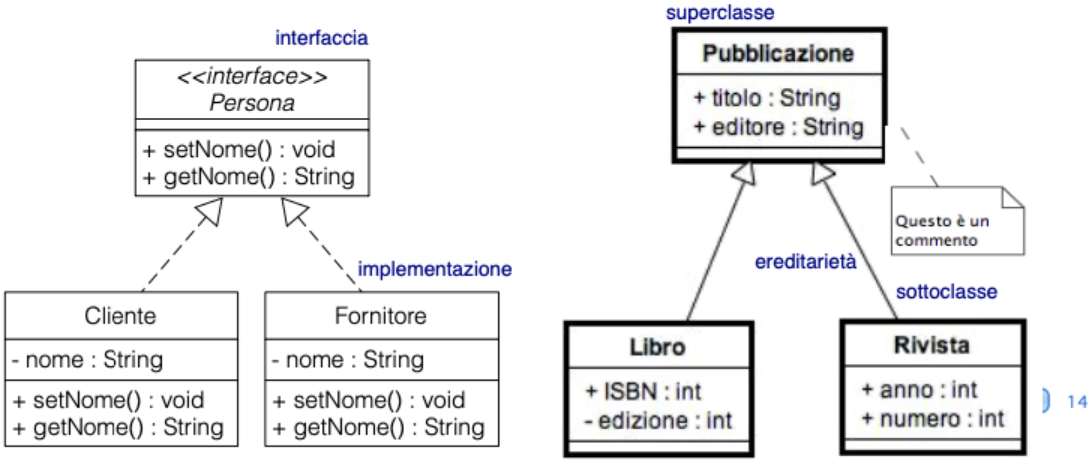


Diagramma di Collaborazione

Diagramma UML di collaborazione

- Mostra interazioni tra oggetti
  - Il flusso dei messaggi è indicato da frecce accanto alle associazioni fra istanze che partecipano all'interazione
  - I messaggi sono mostrati da etichette sulle frecce ed hanno
    - Un numero sequenziale che indica l'ordine temporale con cui avvengono
    - Il metodo chiamato
    - Un valore di ritorno (opzionale)

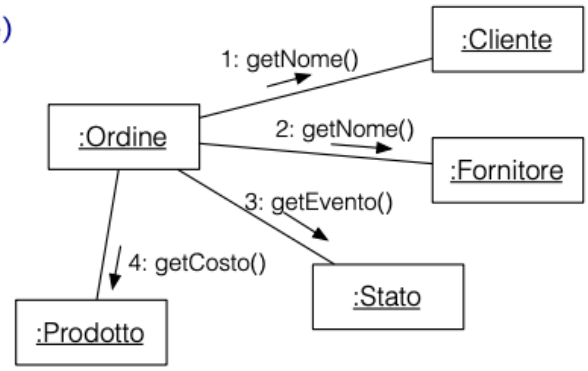
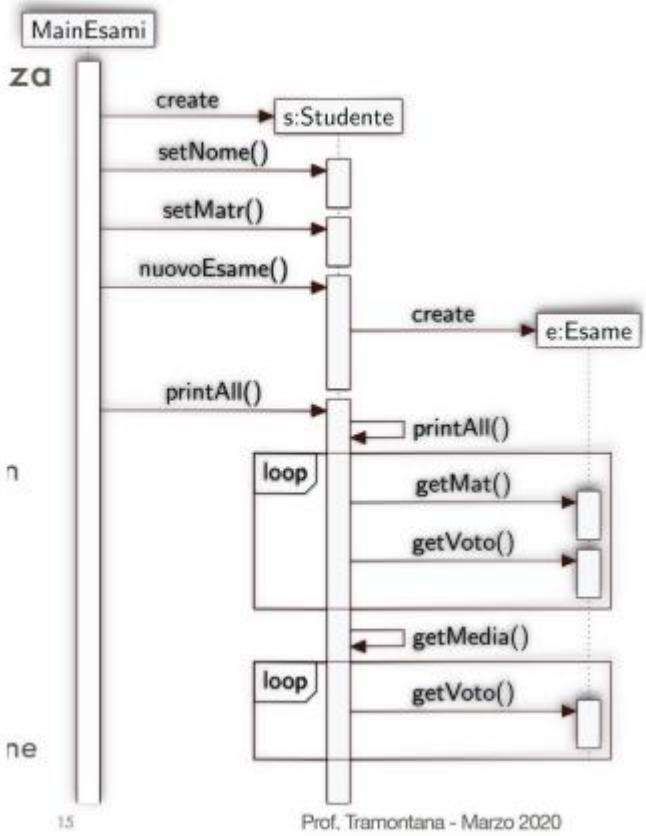


Diagramma di collaborazione per Stampa Ordini

Diagramma di Sequenza

I diagrammi di interazione descrivono come gruppi di oggetti collaborano in alcuni comportamenti. Il diagramma di sequenza è un diagramma di interazione che cattura il comportamento di un singolo scenario, mostrando un numero di oggetti di esempio e i messaggi che vengono passati tra questi oggetti nel caso d'uso.
Un esempio di diagramma di sequenza è il seguente:



- L'asse temporale è inteso in verticale verso il basso;
- In alto in orizzontale risiedono gli oggetti;
- In ciascuna colonna, ogni oggetto esistente è indicato con una linea tratteggiata, detta linea della vita;
- Quando un oggetto è attivo, la linea tratteggiata viene sovrapposta da una barra di attivazione;
- Una chiamata di metodo è indicata da una freccia piena che va dalla barra di attivazione dell'oggetto chiamante alla barra di attivazione dell'oggetto chiamato;

Cicli, condizioni e simili

Un problema comune con i diagrammi di sequenza è come mostrare il comportamento in loop e condizionale. L'obbiettivo dei diagrammi di sequenza è mostrare l'interazione fra gli oggetti piuttosto che la logica di controllo, perciò, nonostante sia possibile inglobare parti del diagramma in sezioni per indicare un ciclo o una condizione, è preferibile usare un diagramma di attività o addirittura con il codice stesso per rappresentarla.

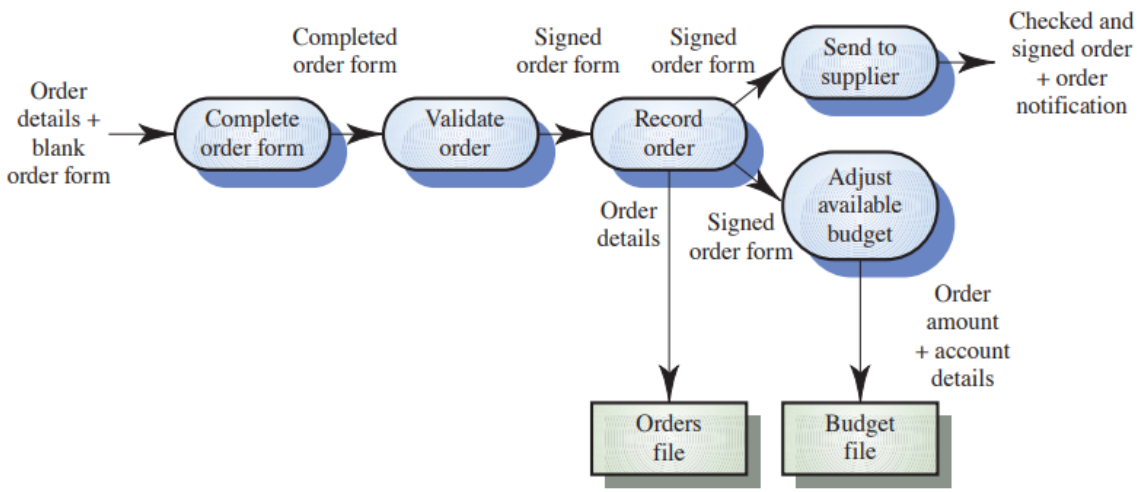
Modello di Comportamento (State Machine UML)

Modelli di comportamento

- Sono usati per descrivere il comportamento globale del sistema
  - **Data processing model** (ovvero Data Flow Diagram, DFD)
    - Mostrano i passi per l'elaborazione di dati che attraversano il sistema
    - Notazione intuitiva comprensibile ai clienti
    - Mostrano lo scambio di informazioni tra sistemi e sottosistemi
    - Simili al diagramma delle attività UML
  - **State machine model** (in UML sono i diagrammi degli Stati)
    - Modellano il comportamento in risposta a eventi interni o esterni
    - Mostrano ***stati*** del sistema come ***nodi*** ed ***eventi*** come ***archi*** tra i nodi
    - Quando un evento si verifica, il sistema passa da uno stato a un altro
    - Utili per sistemi real-time, poiché spesso pilotati da eventi
- Entrambi sono richiesti per ottenere la descrizione del sistema

Riferimenti
Pressman, capitoli 6.5.2, 8.5.2, 8.5.3, 8.6, 8.8
Sommerville, capitoli 7.1, 7.2

Esempio Modello DFD per ordini

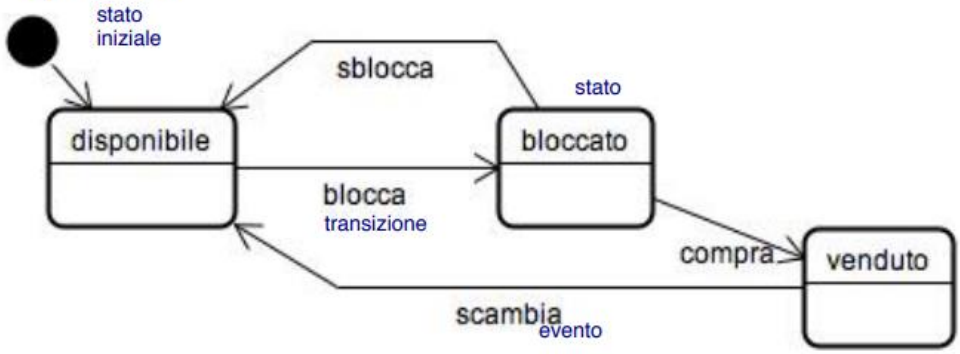


- Rettangoli arrotondati: passi di elaborazione
- Frecce: flussi
- Rettangoli: archivi o sorgenti dati

Diagramma degli Stati

Diagramma UML degli Stati

- Diagramma UML degli stati che mostra la vita di un biglietto per uno spettacolo



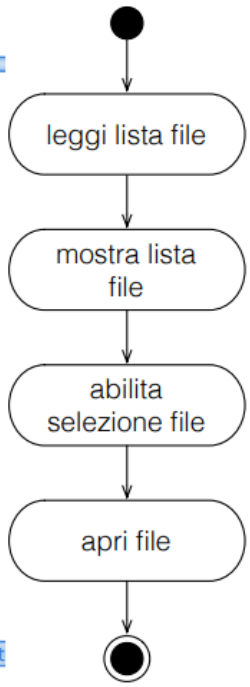
- Rettangoli con angoli arrotondati: stati
- Frecce: transizioni tra stati
- Cerchi pieni: stati iniziale e finale (con circonferenza)

Diagramma delle attività

Diagramma delle attività

- Utilizzato per la modellazione di processi e workflow
- Mostra dettagli di funzionamento interno del sistema software da realizzare
- E' una vista sull'esecuzione delle attività
- Mostra dipendenze tra attività (o passi)

Attività: Apri file da browser



I Design pattern

Cos'è un Design pattern?

I design pattern sono strutture software, o anche micro-architetture, costituite da un piccolo numero di classi che descrivono soluzioni di successo a problemi comuni. Un design pattern nomina, astrae ed identifica gli aspetti chiave di un problema di progettazione, le classi e le istanze che vi partecipano, i loro ruoli e come interagiscono, ovvero la distribuzione delle responsabilità.

Definizione Tramontana: *Un Design Pattern descrive un problema di progettazione ricorrente che si incontra in specifici contesti e presenta una soluzione collaudata generica ma specializzabile.*

L'utilizzo dei Design Pattern aiuta i principianti ad agire come se fossero esperti, supporta gli esperti nella progettazione su grande scala ed evita di re-inventare concetti e soluzioni, riducendo il costo di sviluppo.

Un design pattern non è un codice che è possibile incollare e utilizzare, è di fatto una soluzione generica che necessità di essere adattata alla realtà del proprio programma.

Design Pattern vs Algoritmo

I pattern sono spesso confusi con gli algoritmi, poiché entrambi i concetti descrivono soluzioni tipiche di alcuni problemi noti. In realtà un algoritmo definisce un chiaro insieme di azioni da eseguire per raggiungere qualche obiettivo; un design pattern, invece, fornisce una descrizione di livello più alto di una soluzione. Il codice dello stesso pattern design applicato a due programmi diversi può risultare diverso.

Presentazione formale di un design pattern

La maggior parte dei design pttern sono descritti in modo molto formale in modo che le persone possano riprodurli in molti contesti. Queste sono le sezioni presenti nella descrizione di un design pattern:

- **Nome** del design pattern. Permette di identificare il design pattern con una parola e di lavorare con un alto livello di astrazione, indica lo scopo del pattern. Rende più facile pensare ai progetti e la comunicazione con gli altri.
- **L'intento** del modello descrive brevemente le funzionalità e lo scopo.
- **Il problema**, suddiviso in due parti:
  - **Motivazione,** descrive il problema per cui conviene applicare il pattern;
  - **Soluzione,** le condizioni necessarie per applicare il pattern;
  - Si parla di **forze** (come in fisica) per indicare obbiettivi e vincoli (spesso contrastanti) che si incontrano nel contesto del design pattern;
- **La soluzione** che descrive gli elementi (classi) costitutivi del pattern con le relazioni e responsabilità, senza però addentrarsi in una specifica implementazione. Il concetto è di presentare un problema astratto e la relativa configurazione di elementi adatta a risolverlo.
- **La struttura:** una rappresentazione attraverso diagrammi UML, classi e relazione fra esse, della generica soluzione proposta;

- **Le conseguenze** indicano risultati, compromessi, vantaggi e svantaggi nell'uso del design pattern.

Altre parti della descrizione di un design pattern possono essere:

- **Esempi di utilizzo** illustrano dove il design pattern è stato usato.
- **L'esempio di codice** in uno dei linguaggi di programmazione popolari si semplifica la comprensione dell'idea alla base del design pattern.

Classificazione dei Design Pattern

Classificazione per complessità e scala di applicabilità

I design pattern differiscono per complessità, livello di dettaglio e scala di applicabilità all'intero del sistema progettato:

- I design pattern più elementari e di basso livello sono spesso chiamati "*modi di dire*" e di solito si applicano solo a un singolo linguaggio di programmazione.
- I design pattern più universali e di alto livello sono modelli architettonici e possono essere utilizzati per progettare l'architettura di un'intera applicazione.

Classificazione per scopo

Tutti i design pattern possono essere classificati in base al loro scopo:

- **I design pattern creazionali** forniscono meccanismi di creazione di oggetti che aumentano la flessibilità e il riutilizzo del codice esistente. [Riguardano la creazione di istanze]
  - *Singleton, Factory Method, Abstract Factory, Object Pool, Builder, Prototype*
- **I design pattern strutturali** spiegano come assemblare oggetti e classi in strutture più grandi, mantenendo le strutture flessibili ed efficienti. [Riguardano la scelta della struttura]
  - *Adapter, Facade, Composite, Decorator, Bridge, Flyweight, Proxy*
- **I design pattern comportamentali** si occupano di fornire una comunicazione efficace e dell'assegnazione delle responsabilità tra gli oggetti. [Riguardano la scelta dell'incapsulamento]
  - *Iterator, Template Method, Mediator, Observer, State, Strategy, Chain of Responsability, Command, Interpreter, Memento, Visitor*

I Design pattern creazionali

I design pattern creazionali forniscono meccanismi di creazione di oggetti che aumentano la flessibilità e il riutilizzo del codice esistente.

Singleton

- **Intento:** Garantire che una classe abbia solo un'istanza, fornendo al contempo un punto di accesso globale a questa istanza.
- **Problema:** Vi è la necessità che alcune classi abbiano esattamente una sola istanza durante il run-time dell'applicazione. I campi di interesse più comuni per questa necessità può essere l'accesso ad alcune risorse condivise, come ad esempio un database o un file.
  Un approccio semplicistico per garantire un accesso globale ci porterebbe a istanziare un oggetto in una variabile globale, ma ciò non impedirebbe un ulteriore istanziamento.
  Inoltre, una volta creato un oggetto, ogni tentativo di accesso deve fare riferimento a tale istanza.
  La classe stessa dovrebbe quindi fornire e tenere traccia del punto d'accesso globale.
- **Soluzione:** il Singleton presenta le seguenti caratteristiche:
  - La classe contiene una istanza statica e privata di sé stessa;
  - La classe è responsabile della creazione dell'unica istanza, attraverso un approccio preventivo o lazy (= lazy initialization):
    - Con un approccio preventivo, l'istanza viene creata arbitrariamente all'avvio dell'applicativo;
    - Con un approccio lazy, l'istanza viene creata al primo tentativo di accesso;
  - Il costruttore della classe è reso privato per impedire la creazione di una istanza al di fuori della classe stessa da parte di terzi;
  - La classe contiene metodo pubblico e statico denominato getInstance() che fornisce il punto d'accesso, ovvero restituisce l'istanza. Nel caso di implementazione lazy, tale metodo si occuperà di creare l'istanza al primo tentativo di accesso.
- **Struttura**: Diagramma delle classi

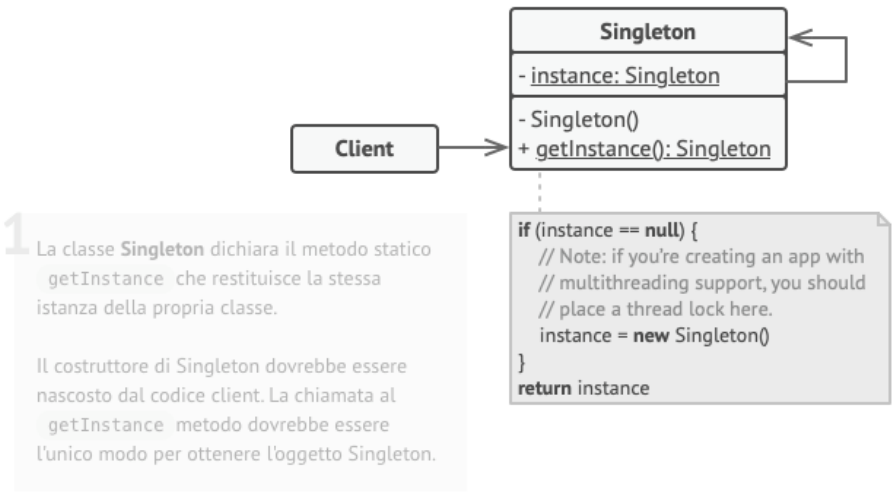
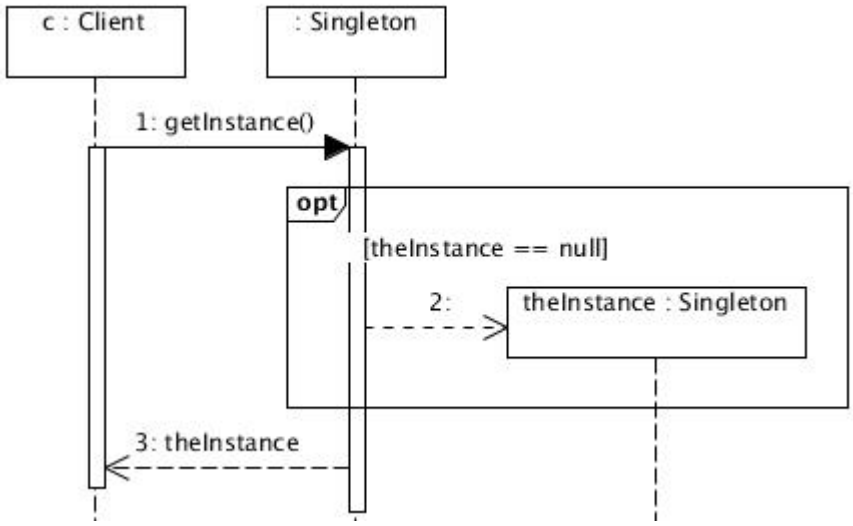


Diagramma di sequenza:



- **Conseguenze:**
  - ✓ Assicura che la classe abbia una sola istanza;
  - ✓ Il passaggio fra Singleton e non è alquanto semplice poiché basta modificare il metodo getInstance() per ottenere il comportamento desiderato (comportamento singleton o generatore di nuove istanze);
  - ✓ La classe Singleton ha pieno controllo di come e quando i client accedono all'istanza, fornendo lei stessa l'accesso globale a tutti gli interessati;
  - ✓ Previene l'uso di variabili globali per l'utilizzo della sola istanza condivisa;
  - ✓ Il singleton è più flessibile rispetto alla creazione di soli metodi e attributi statici, poiché può cambiare facilmente il numero di istanze consentite, mentre l'approccio static prevede la modifica di tutte le invocazioni.
  - × In un approccio lazy viola il principio di singola responsabilità poiché il getInstance() risolve due problemi per volta (creazione e restituzione dell'istanza);
  - × In un ambiente multi-thread richiede un trattamento particolare, poiché getInstance() fornisce una risorsa condivisa;
- **Dettagli Implementativi:**
  - Individuare un oggetto che necessità di essere unico in run-time;
  - Aggiungere un riferimento statico e privato della classe alla classe stessa;
  - Rendere privato il costruttore;
  - Aggiungere il metodo pubblico e statico getInstance(), implementato con l'approccio che si preferisce;
  - Sostituisce nel codice tutte le inizializzazioni di un oggetto della classe desiderata con il richiamo al metodo getInstance();

Variante Singleton: il Multiton
Tale variante consiste nel fatto che invece di istanziare un solo oggetto è possibile istanziarne un numero limitato.

Principio delle conseguenze locali
Un cambiamento in qualche punto del codice non dovrebbe causare problemi in altri punti.

Factory Method

- **Intento:** Definire una interfaccia per creare un oggetto, ma lasciare che le sottoclassi decidano quale classe istanziare. Factory Method permette ad una classe di rimandare l'istanziazione alle sottoclassi. Sarà quindi possibile distinguere: chi vuole l'istanza, chi può dare l'istanza, chi crea l'istanza.
- **Problema:** I framework utilizzano classi astratte per definire e mantenere le relazioni tra gli oggetti. Il framework deve creare oggetti ma conosce solo classi astratte che non può istanziare.
    - **Definizione di framework**: Un framework è definito da un insieme di classi astratte e dalle relazioni tra esse. Istanziare un framework significa fornire un'implementazione delle classi astratte. L'insieme delle classi concrete eredita le relazioni tra le classi.

  Il Factory Method offre una soluzione fornendo un metodo, detto factory o fabbricatore, responsabile per l'istanziazione che incapsula la conoscenza su quale classe creare.
  Utilizzare il Factory Method quando:
    - Una classe non può anticipare la classe dell'oggetto da istanziare;
    - Una classe vuole che le sue sottoclassi specifichino gli oggetti creati;
    - Una classe delega la responsabilità a una delle diverse sottoclassi di supporto, e si vuole localizzare la conoscenza rispetto a quale sottoclasse di supporto è delegato.

  Questo pattern consente di separare il Client dal Framework permettendo di modificare i dettagli implementativi senza dover modificare il Client.
- **Soluzione:** Il Factory Method presenta le seguenti caratteristiche:
    - **Product:** L'interfaccia comune degli oggetti creati dal factoryMethod();
    - **ConcreteProduct:** Una delle varie implementazioni del *product;*
    - **Creator:** Un'interfaccia che dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Il creator può avere un'implementazione di default del factoryMethod() che ritorna un certo ConcreteProduct.
    - **ConcreteCreator:** Implementa il factoryMethod() o ne fa override. Sceglie quale ConcreteProduct istanziare e ritorna tale istanza.
- **Struttura:**
  Diagramma delle classi:

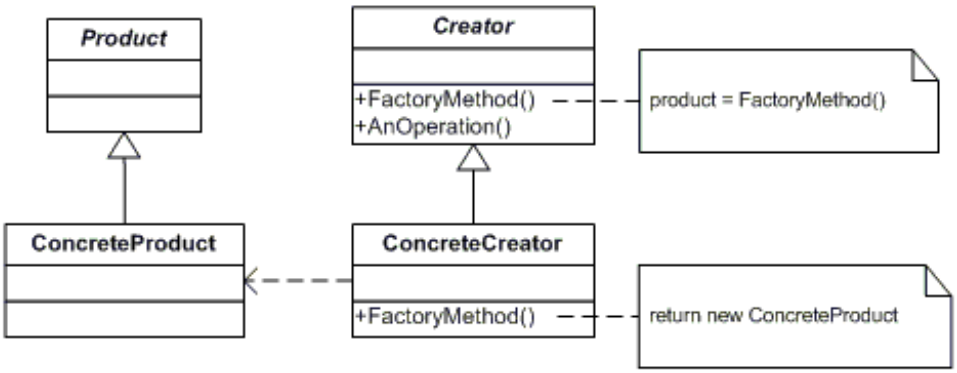
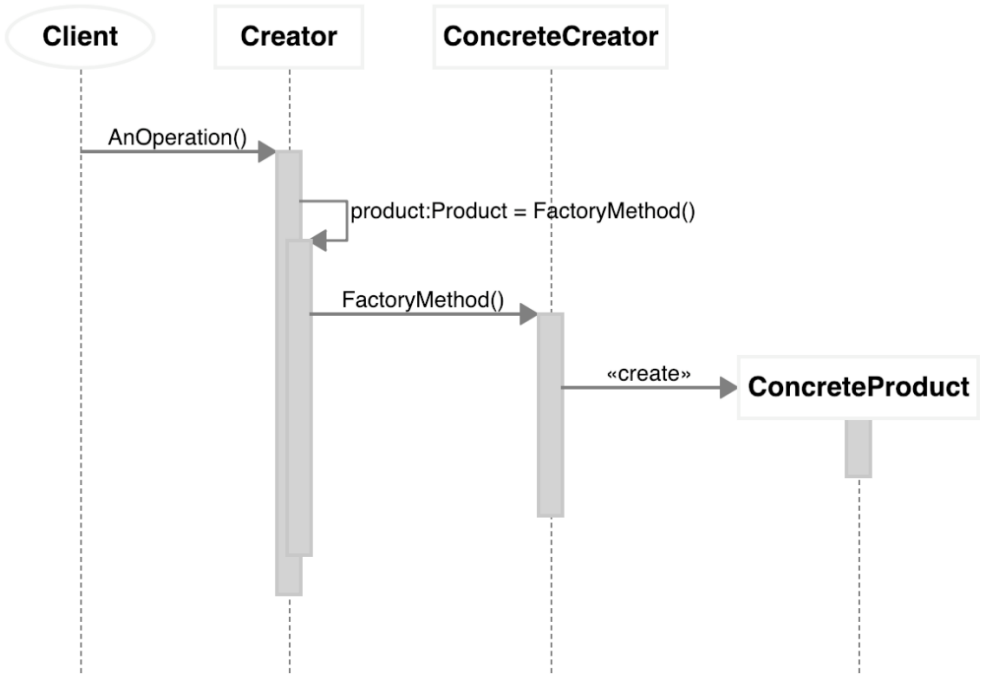


Diagramma di sequenza:



- **Varianti:**
  - I ruoli di Creator e ConcreteCreator possono essere svolti nella stessa classe;
  - Il factoryMethod() può essere un metodo statico;
  - Il factoryMethod() può disporre di un parametro che permette al client di suggerire quale classe ConcreteProduct istanziare e tornare;
  - Il factoryMethod usa la *Riflessione Computazionale* per eliminare le dipendenze dai ConcreteProduct, la classe istanziata sarà nota a runtime.
- **Conseguenze**
  - ✓ Il codice conosce solo l'interfaccia Product e può lavorare con qualsiasi ConcreteProduct. I ConcreteProduct sono facilmente intercambiabili;
  - ✓ Se si implementa una sottoclasse (ConcreteCreator) di Creator per ciascun ConcreteProduct da istanziare si ha una proliferazione di classi;
    - × Potrebbe rivelarsi uno svantaggio la variante in cui Creator e ConcreateCreator sono unificati tende a risolvere tale problematica;
  - ✓ Fornisce un appiglio per le sottoclassi, creare un oggetto da dentro una classe con il factory method è più flessibile della creazione diretta;
- **Dettagli Implementativi:**
  - Tutti i ConcreteProduct implementano la stessa interfaccia (Product), la quale deve dichiarare metodi che hanno senso in ogni prodotto;
  - Si aggiunge un factoryMethod() vuoto all'interno dell'interfaccia Creator, il cui tipo restituito corrispondere all'interfaccia comune del Product;
  - All'interno dell'interfaccia Creator sono presenti tutti i riferimenti dei costruttori dei ConcreteProduct;
  - Si crea un ConcreteCreator per ogni ConcreteProduct che implementerà il metodo factoryMethod() in modo da ritornare il corretto ConcreteProduct;

Object pool [non approfondito]

Un object pool è un deposito di istanze già create, una istanza sarà estratta dal pool quando un client ne fa richiesta, quest'ultimo dovrà restituire l'istanza quando non gli sarà più utile.
Il pool può avere una dimensione variabile, ovvero cresce nel tempo, o fissa: qualora sia di dimensioni fissa e se non ci sono più oggetti disponibili al momento della richiesta, non ne se creano di nuovi.
L'obbiettivo della object pool è ridurre i tempi necessari per ottenere istanze di oggetti complessi.
L'object pool potrebbe essere concepito per essere unico, perciò per l'implementazione si potrebbe usare il Singleton.
Il design pattern Factory Method può implementare un object pool:

- I client fanno richieste, il ConcreateCreator invece di istanziare un nuovo oggetto preleva un'istanza dalla pool;
- I client dovranno dire quando l'istanza non è più in uso, quest'ultima verrà quindi resettata allo stato di partenza e sarà pronta per essere riutilizzata;

Dependency Injection (Una tecnica non un design pattern!)

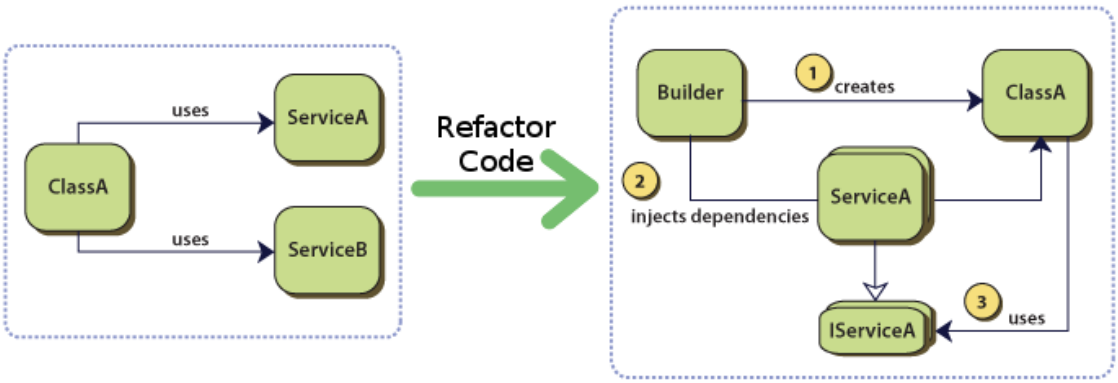
Ipotizzando che una classe C utilizzi un servizio di una classe S, possiamo dire che C dipende da S. Ipotizzando ulteriormente che esistano più implementazioni, e quindi sottoclassi, del servizio S e vi sia la necessità che C non debba dipendere da una specifica implementazione di S. Per tale scopo possiamo usare il pattern Factory Method per "iniettare" la dipendenza desiderata al momento della creazione dell'istanza.

Situazione

- Una classe C usa un servizio S (ovvero C dipende da S);
- Esistono tante implementazioni di S (ovvero S1, S2), la classe C non deve dipendere dalle implementazioni S1, S2;
- Al momento di creare l'istanza C, indico all'istanza C con quale implementazione di S deve operare.

Esempio

- Una classe TextEditor usa un servizio SpellingCheck
- Ci sono tante classi che implementano il servizio SpellingCheck, in base alla lingua usata: SpCheckEnglish, SpCheckItalian, etc.
- TextEditor deve poter essere collegato ad una delle classi che implementano SpellingCheck.



2Builder = Factory

I Design pattern strutturali

I design pattern strutturali spiegano come assemblare oggetti e classi in strutture più grandi, mantenendo le strutture flessibili ed efficienti.

Adapter

- **Intento:** Permettere ad oggetti con interfacce incompatibili di interagire, ovvero convertendo l'interfaccia di una classe in un'altra interfaccia che il client conosce.
- **Problema:** Alcune volte una classe di una libreria non può essere usata poiché incompatibile con l'interfaccia che l'applicazione usa. Ovvero il nome metodo, i parametri, il tipo di ritorno che l'applicazione prevede non sono corrispondenti a quelli offerti da una classe di libreria.
  Si ha forti necessità di usare la libreria, ma non è possibile (o consigliabile) cambiare il codice sorgente della libreria, né tantomeno apportare le modifiche di adeguamento all'applicazione.
- **Soluzione:** È possibile creare un *Adapter*, ovvero un oggetto speciale che converte, o adatta, l'interfaccia di un oggetto, che chiameremo *Adaptee,* in un'interfaccia *target,* utilizzabile da un altro oggetto o client.
  L'adapter converte, o adatta, le chiamate che fa un client all'interfaccia della classe della libreria. Il chiamante non conosce direttamene l'adaptee in quanto usa l'adapter come se fosse l'oggetto della libreria.
  L'adapter tiene un riferimento all'oggetto della libreria (adaptee) e sa come invocarlo, ovvero implementa delle chiamate verso i metodi di adaptee.
  L'Adapter deve delegare la maggior parte del lavoro all'oggetto della libreria, gestendo solo l'interfaccia o la conversione del formato dei dati
  La comunicazione avviene come segue:
  1. L'adapter fornisce una interfaccia target al client
  2. Il client richiama i metodi dell'interfaccia target
  3. L'adapter converte la richiesta adattandola alla classe adaptee
  4. L'adapter richiama correttamente i metodi di adaptee

  Il design pattern Adapter prevede le seguenti caratteristiche:
  - **Target:** L'interfaccia che l'oggetto o il client conosce e sa usare;
  - **Adaptee:** L'oggetto della libreria che necessità di un adattamento;
  - **Adapter:** L'oggetto speciale che fa tramite o traduttore tra le due interfacce, a seconda dell'implementazione si hanno due varianti del pattern design;
    - **Adapter a due vie**: la classe Adapter fornisce un'interfaccia sia al client sia al adaptee, ovvero adatta in entrambe le direzioni;
- **Struttura:**
  - La variante **object adapter** è un'implementazione che utilizza il *principio di composizione*: *l'adapter* implementa l'interfaccia target e contiene un'istanza dell'oggetto *adaptee*.

Diagramma delle classi:

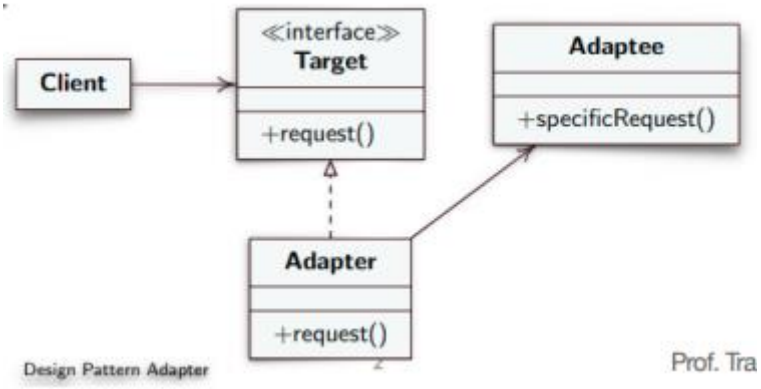
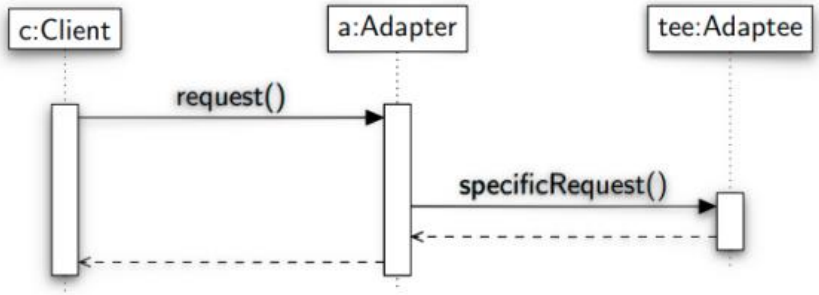


Diagramma di sequenza:



- La variante **class adapter** è un'implementazione che utilizza l'*ereditarietà multipla*: *l'adapter* estende sia l'interfaccia target la classe dell'oggetto *adaptee*. *Costituisce un adapter a due vie*.
  Diagramma delle classi:

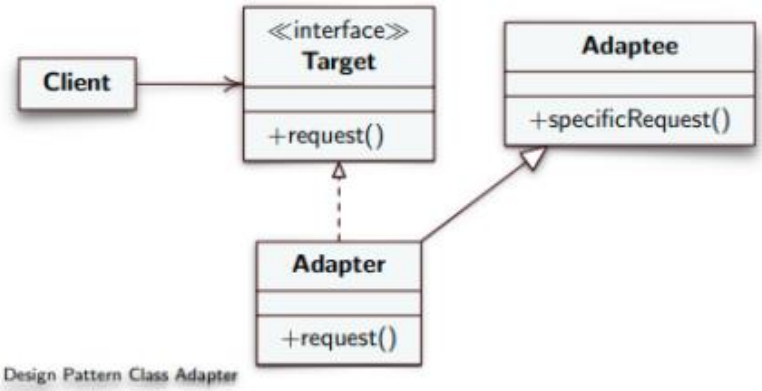
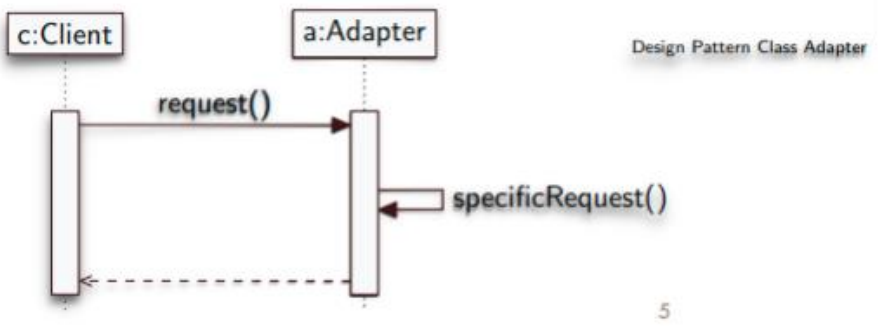


Diagramma di sequenza:



- **Conseguenze**:
  - ✓ Il client e la classe libreria rimangono indipendenti;
  - ✓ L'Adapter può cambiare il comportamento dell'Adaptee;
  - ✓ Si possono aggiungere test di precondizioni e postcondizioni:
    - *Precondizione*: è una condizione che deve essere soddisfatta prima di poter eseguire il metodo della libreria;
    - *Postcondizione*: è una condizione che deve essere soddisfatta dopo l'esecuzione del metodo della libreria per indicare la buona riuscita;
  - ✓ La variante object adapter può implementare la tecnica di *Lazy Initialization*.
  - ✓ I client utilizzano l'Adapter tramite l'interfaccia Target, ciò consente di modificare o estendere gli adattatori senza influire sul codice Client;
  - × Il design pattern Adapter aggiunge un livello di indirettezza.
  - × Nella variante object adapter ad ogni invocazione del client ne scaturisce un'altra dall'adapter da cui si ha un possibile rallentamento (trascurabile);
  - × Il codice diventa più difficile da comprendere;
- **Dettagli Implementativi**:
  - Assicurarsi di avere due classi incompatibili: una classe di servizio non è utilizzabile ma una o più classi del client trarrebbero vantaggio dal suo utilizzo;
  - Dichiarare l'interfaccia client (Target) e descrivere in che modo il client vorrebbe comunicare;
  - Creare un oggetto Adapter che implementa l'interfaccia Target e contiene un riferimento alla classe di servizio;
  - Implementare tutti i metodi dell'interfaccia Target;

Facade

- **Intento:** Fornire un'interfaccia unica e semplificata al posto di un insieme di interfacce di un sottosistema (consistente di un insieme di classi).
  Definire un'interfaccia di alto livello semplificata che rende il sottosistema più facile da usare.
- **Problema:** Spesso si hanno tante classi che svolgono funzioni correlate e l'insieme delle interfacce può essere complesso e può essere difficile capire qual è l'interfaccia essenziale ai client per l'insieme di classi.
  Un obiettivo di progettazione comune è ridurre al minimo la comunicazione e le dipendenze tra il client ed il sottosistema, ma anche fra i sottosistemi stessi.

  Utilizzare il Facade quando:
  - Si desidera fornire un'interfaccia unica, semplice ma limitata a un sottosistema complesso. Solo i clienti che necessitano di maggiore personalizzazione dovranno guardare oltre Facade.
  - Ci sono molte dipendenze tra i client e le classi di implementazione di un'astrazione. Il Facade separa il sottosistema dai client e dagli altri sottosistemi, promuovendo l'indipendenza e la portabilità del sottosistema.
  - Si desidera stratificare i sottosistemi. Si utilizza Facade per definire un punto di ingresso per ciascun livello di sottosistema.
- **Soluzione:** Si fornisce un'unica interfaccia semplificata ai client e nasconde gli oggetti del sottosistema, questo riduce la complessità dell'interfaccia e quindi delle chiamate. Il client interagisce solo con l'oggetto Facade, quest'ultimo invoca i metodi degli oggetti che nasconde.
- **Struttura:**
  Diagramma delle classi:

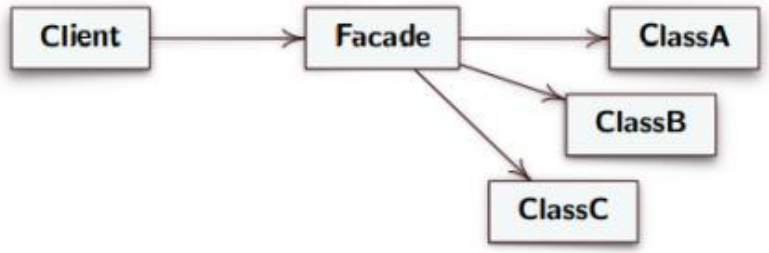
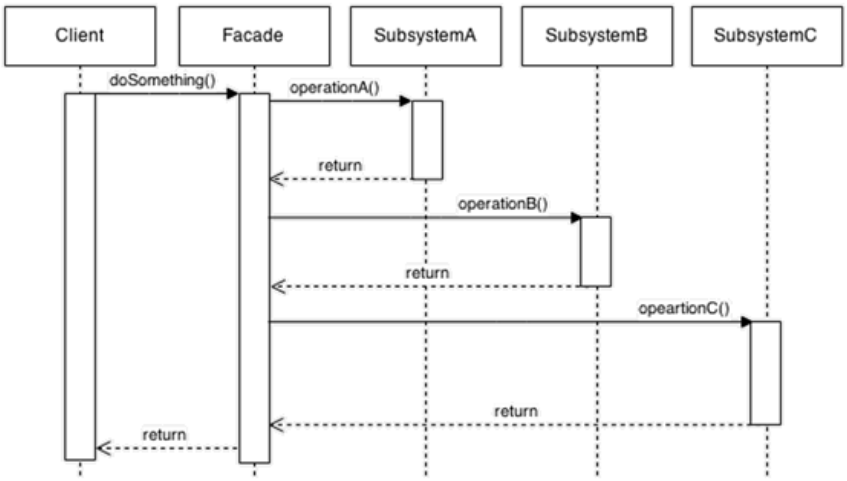


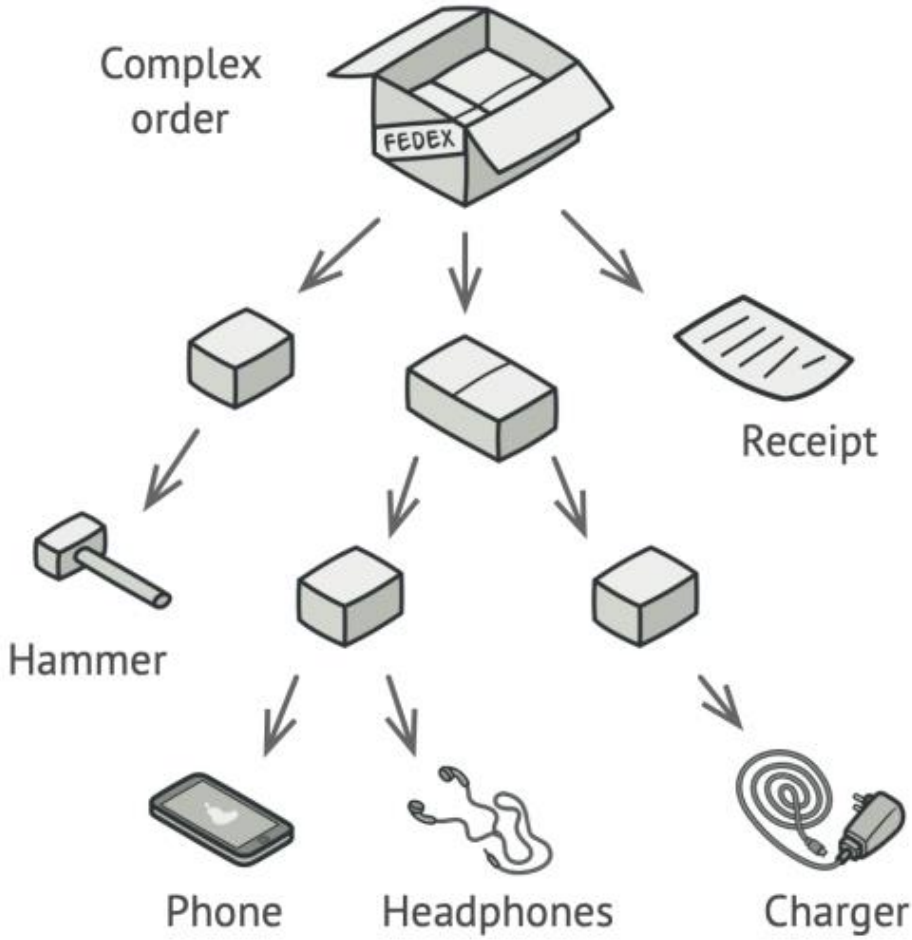
  Diagramma di sequenza:



- **Conseguenze**
  - ✓ Nasconde ai client l'implementazione del sottosistema;
  - ✓ Riduce il numero di oggetti che i client devono gestire e facilitano l'utilizzo del sistema;
  - ✓ Promuove *l'accoppiamento debole* tra sottosistema e client;
    - Detto anche *accoppiamento lasco*, indica che il client ha poca o nessuna conoscenza di come sia fatto il sottosistema;
  - ✓ Riduce le dipendenze di compilazione in sistemi grandi: una modifica a una classe del sottosistema richiede la sola ricompilazione del sottosistema fino alla facade, lasciando inalterati i client;
  - × Non impedisce alle applicazioni di usare le classi del sottosistema se necessario.
- **Dettagli Implementativi:**
  - Verificare se è possibile fornire un'interfaccia più semplice rispetto a quella fornita da un sottosistema esistente e che renda il codice del client indipendente da molte delle classi del sottosistema;
  - Dichiarare e implementare questa interfaccia in una nuova classe Facade, la quale deve reindirizzare le chiamate dal codice client agli oggetti appropriati del sottosistema.
  - La Facade potrebbe anche occuparsi dell'inizializzazione dei componenti del sottosistema;
  - Per ottenere l'indipendenza del client dal sottosistema è necessario che il client comunichi unicamente con la Facade;
  - Se la Facade diventa troppo grande, è consigliabile dividerla in Facade più semplici;

Composite

- **Intento:** Comporre oggetti in strutture ad albero e quindi operare con esse come se fossero singoli oggetti.
- **Problema:** Ipotizzando di voler calcolare il prezzo finale di un ordine, costituito da prodotti finali, aventi un proprio prezzo, e pacchetti contenenti altri prodotti finali o pacchetti, il cui prezzo finale è, riproponendo il quesito iniziale, la somma dei prezzi del suo contenuto. (Segue la struttura ad albero che caratterizza l'ordine)



Utilizzare il Composite quando:

  - È necessario raggruppare elementi semplici tra loro per formare elementi più grandi;
  - Nell'implementazione c'è distinzione tra classi per elementi semplici e classi per contenitori di questi elementi semplici, il codice che interessa queste classi deve trattarli in modo differente, rendendosi così più complicato;
  - Vi è necessità di descrivere una composizione ricorsiva, in modo che i client non debbano fare distinzione tra tipi di elementi. I client tratteranno tutti gli oggetti della struttura uniformemente;
    - È ciò che in sostanza permette il Composite;
- **Soluzione:** Il Composite prevede le seguenti caratteristiche:
  - **Component**: Un'interfaccia (o classe abstract) che rappresenta elementi semplici e i contenitori. Definisce le operazioni (*operation*) comuni agli oggetti semplici e ai contenitori.

- **Leaf**: una classe che rappresenta elementi semplici, detti children. Tale classe implementa l'interfaccia Component e contiene la logica delle *operation* relative agli oggetti semplici;
- **Composite**: una classe che rappresenta elementi contenitori. Tale classe implementa l'interfaccia Component e contiene la logica delle *operation* relative agli oggetti contenitori, che consiste nel consultare, riguardo l'operazione decisa, tutti gli elementi che contiene (elementi semplici o ulteriori contenitori);
- **Concetto di collaborazione**: Il client interagisce indistintamente con Leaf e Composite, ma quando un metodo viene chiamato su un Leaf, esso verrà eseguito direttamente; mentre quando un metodo è eseguito su un Composite, esso verrà eseguito su tutti gli elementi che lo compongono;

- **Struttura:**
  A seconda della dichiarazione dei metodi *add()* e *remove()*, tipici degli elementi contenitori, si ottengono due varianti con caratteristiche opposte:
  - **Tendente alla sicurezza:** quando tali metodi vengono dichiarati nella classe *Composite* per impedire che gli elementi semplici possano vederli o richiamarci (vi è già un controllo a compile-time). Tale approccio perde di trasparenza, in quando elementi semplici e contenitori dovranno essere distinti;
  - **Tendente alla trasparenza:** quando tali metodi vengono dichiarati nell'interfaccia *Component***,** con la conseguenza necessità di un'implementazione banale o nullafacente nella classe Leaf. Tale approccio per sicurezza poiché gli elementi semplici vedono e potrebbero richiamare metodi che non sono concepiti per essi;
    Diagramma delle classi:

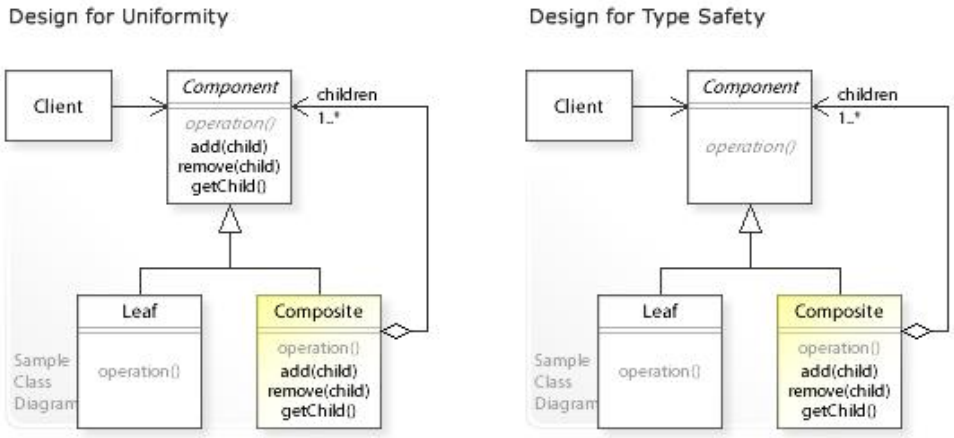
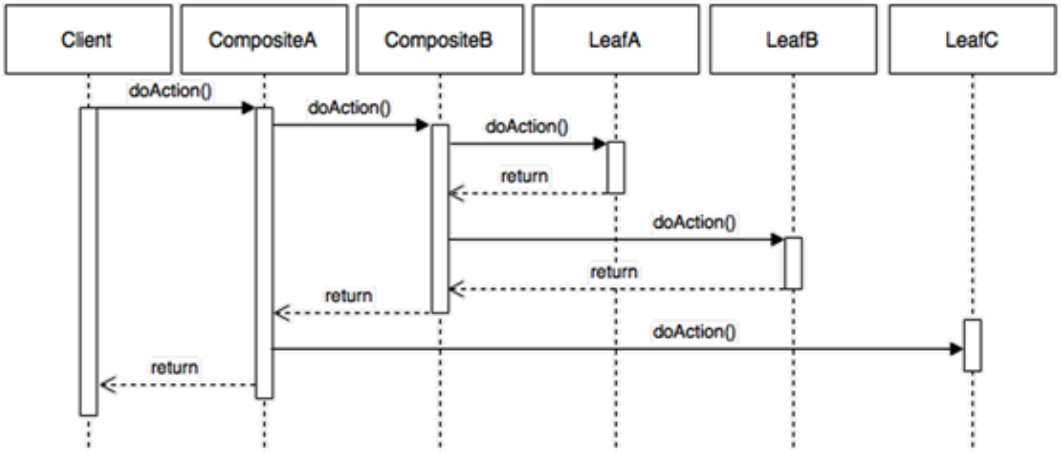


Diagramma di sequenza:

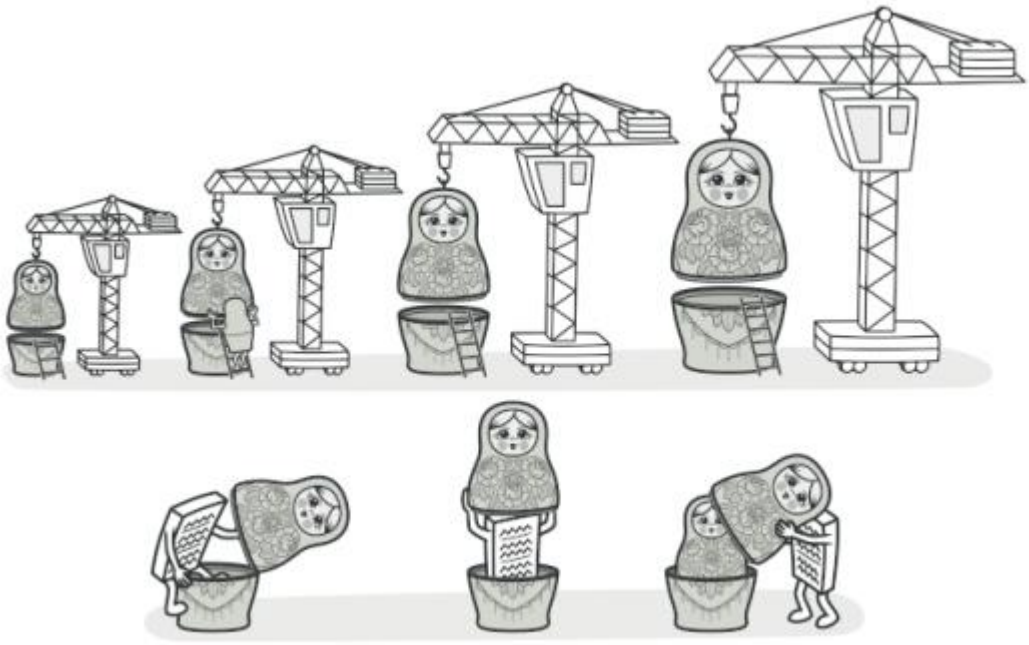


- **Conseguenze**
  - ✓ Gli elementi semplici potrebbero essere contenuti in contenitori che a sua volta sono contenuti in altri contenitori (*composizione ricorsiva*);
  - ✓ Non vi è alcun problema se un client che si aspetta un elemento semplice ottiene un contenitore, in quanto esso non li distingue e vengono trattati uniformemente;
  - ✓ Nuovi tipi di elementi (semplici o contenitori) sono di facile integrazione in strutture dati o client esistenti;
  - × Può rendere il progetto eccessivamente generalistico: non vi è modo di limitare il tipo di elementi contenuti in un Composite se non con controlli a run-time;
- **Dettagli Implementativi:**
  - Si definisce un metodo *getComposite*() all'interno di *Component* che restituisce *null*, nella classe *Composite* esso verrà sovrascritto per ritornare se stesso;
  - La lista degli elementi che compongono un Composite è di tipo *Component* ed è definita all'interno della classe Composite per evitare lo spreco di memoria, dovuta a un'allocazione inutile su Leaf qualora la lista fosse definita in Component;
  - L'ordinamento degli elementi della lista di un Composite potrebbe essere importante;
  - Il Composite potrebbe implementare una sorta di cache per ottimizzare le prestazioni, utile qualora una operazione debba effettuare una ricerca. In tal caso gli elementi Left devono disporre di un metodo per invalidare la cache;

Decorator

- **Intento:** Aggiungere responsabilità ad un oggetto dinamicamente, fornendo un'alternativa flessibile rispetto all'uso dell'ereditarietà per estendere funzionalità.
- **Problema:** Si vuole aggiungere una responsabilità, nuove implementazioni, ad uno specifico oggetto, istanza, e non all'intera classe, inoltre non si vuole modificare quest'ultima.
  L'aggiunta di una responsabilità deve risultate totalmente trasparente, ovvero non deve cambiare il modo con cui il client opera con l'oggetto.

  Il design pattern Decorator provvede non solo ad aggiungere responsabilità dinamicamente, ma anche a sottrarle. Inoltre è possibile annidare questi comportamenti aggiuntivi, assumendo ideologicamente la forma di una matrioska.



  È lampante la differenza fra questo approccio e l'ereditarietà delle classi, quest'ultima infatti
  - Una sottoclasse può estendere solo una classe per volta;
  - Può solo aggiungere un comportamento e non rimuoverlo;
  - Non gode della flessibilità e della ricorsività del Decorator, una definizione massiccia di sottoclassi può rappresentare l'anti-pattern Class Explosion;

  Esempio: Si voglia stilizzare un determinato bottone senza dover modificare la classe; in tal caso è necessario creare decoratori come bordo, colore, larghezza e applicarli al bottone "base" rappresentato dal ConcreteComponent;
- **Soluzione:** Il Decorator prevede le seguenti caratteristiche:
  - **Component**: Un'interfaccia comune sia per gli oggetti a cui è possibile aggiungere responsabilità sia ai decorator;
  - **ConcreteComponent**: una classe che implementa l'interfaccia Component e definisce l'oggetto a cui è possibile aggiungere responsabilità;
  - **Decorator**: una classe che implementa l'interfaccia Component, implementandone i metodi, e mantiene un riferimento a un oggetto

Component. Inoltra le chiamate ricevute all'oggetto Component che contiene;

- **ConcreteDecorator**: una classe che estende la classe Decorator e implementa la nuova responsabilità tramite ovverriding, eseguendo la nuova responsabilità prima o dopo aver chiamato il metodo genitore;

- **Struttura:**
  Diagramma delle classi:

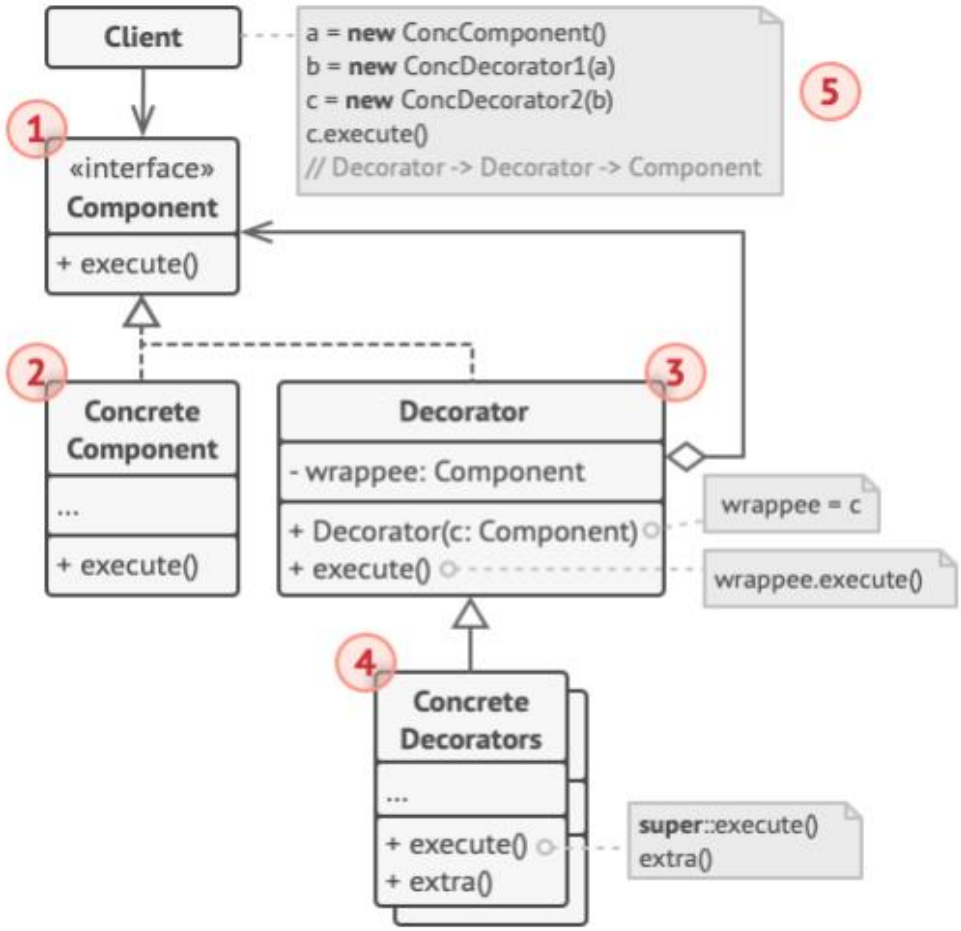
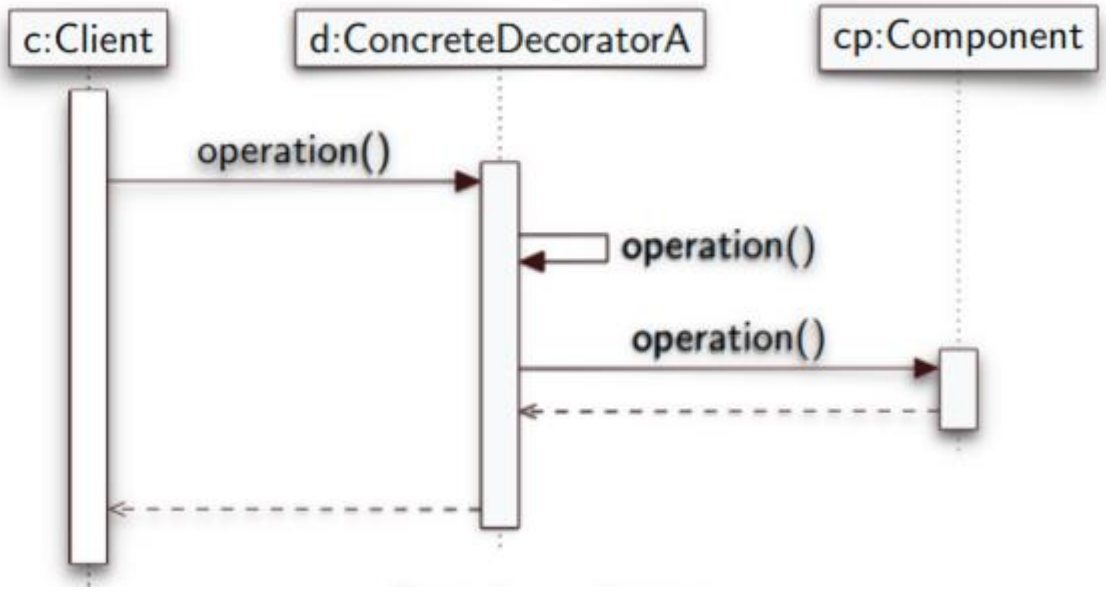


Diagramma di sequenza:



- **Conseguenze**
  - ✓ Si ha più flessibilità rispetto all'ereditarietà poiché si possono aggiungere una o più responsabilità dinamicamente;
  - ✓ Una responsabilità può essere aggiunta più volte su oggetti differenti utilizzando lo stesso ConcreteDecorator;
  - ✓ Alleggerisce le responsabilità alle classi ConcreteComponent, rispettando il principio *SRP;*
  - × Rimuovere una responsabilità in una posizione intermedia nella pila delle responsabilità è alquanto macchinoso;
  - × Il comportamento di una responsabilità potrebbe essere influenzato dalla sua posizione nella pila delle responsabilità;
  - × Si avranno tanti piccoli oggetti, che differiscono nel modo in cui sono interconnessi e quindi si potrebbe avere una proliferazione di classi che rendono difficile il debug di un programma;
- **Dettagli Implementativi**:
  - Il comportamento basilare di un oggetto deve essere definito nella classe ConcreteComponent;
  - Tutte le classi devono implementare l'interfaccia Component;
  - Il client è responsabile della creazione dei decoratori e della loro composizione per i proprio obbiettivi;

Bridge

- **Intento:** Disaccoppiare una astrazione dalla sua implementazione affinché queste possano essere sviluppate indipendentemente.
- **Problema:** Quando un'astrazione può avere diverse implementazioni si utilizza spesso l'ereditarietà: una classe astratta definisce l'interfaccia dell'astrazione mentre diverse sottoclassi concrete la implementano in modi diversi. Tale approccio non è flessibile in quanto collega in modo permanente l'astrazione all'implementazione.

  Esempio: Si vogliono disegnare delle figure, rettangoli e cerchi, su diversi ambienti che operano in maniera diversa; si hanno le seguenti classi:
  - L'interfaccia generica Shape e le due sottoclassi Rettangolo e Cerchio, queste rappresentano l'astrazione;
  - Gli ambienti P1 e P2 che operano rispettivamente tramite drawLine() e drawCicle();

  Per poter disegnare le figure abbiamo bisogno di ogni figura per ogni ambiente disponibile, situazione che con l'aggiunta di nuove figure o ambienti causerebbe l'anti-pattern Class Explosion;
- **Soluzione:** Il Bridge prevede le seguenti caratteristiche:
  - **Abstraction**: Un'interfaccia per il client che contiene un riferimento ad un oggetto di tipo *Implementor*. Fornisce operazioni ad alto livello e inoltra le richieste del client all'oggeto Implementor che contiene;
  - **RefinedAbstraction**: una classe che implementa l'interfaccia Abstraction;
  - **Implementor**: un'interfaccia per le classi dell'implementazione, a differenza dell'Abstraction tale interfaccia fornisce *operazioni primitive*;
  - **ConcreteImplementor**: una classe che implementa l'interfaccia Implementor e definisce le *operazioni primitive*;
- **Struttura:**
  Diagramma delle classi:

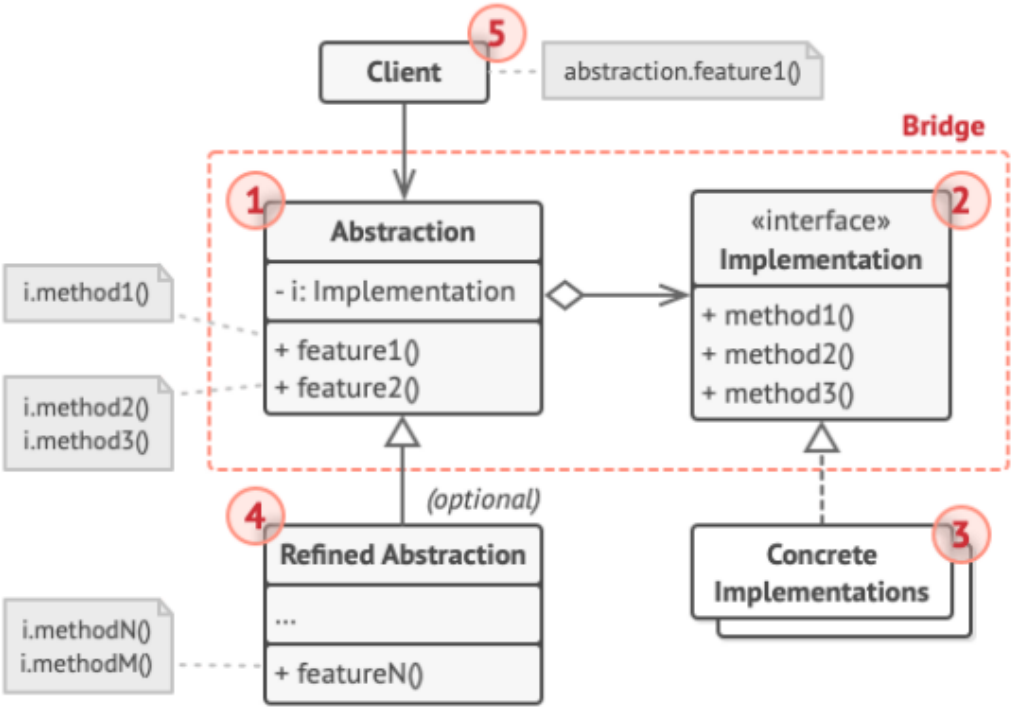
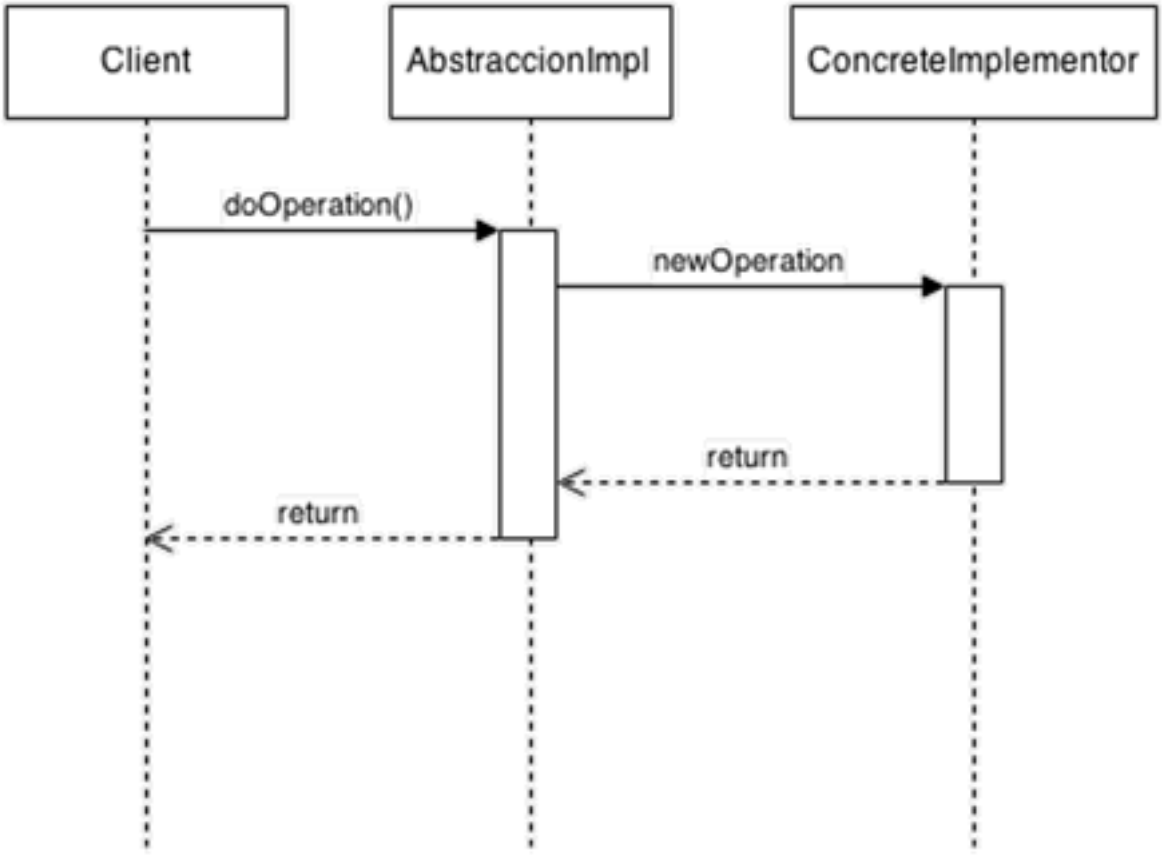


Diagramma di sequenza:



- **Conseguenze**
  - ✓ Si ha un disaccoppiamento fra implementazione e astrazione;
  - ✓ L'implementazione può variare a run-time;
  - ✓ Le gerarchie di Abstraction e Implementor possono evolvere in modo indipendente;
  - ✓ È possibile modificare l'astrazione o l'implementazione senza dover ricompilare l'intero sistema;
  - ✓ Solo certi strati del software devono conoscere Abstraction e Implementor;
- **Dettagli Implementativi**:
  - Se si hanno più varianti di logica ad alto livello è necessario creare più RefinedAbstraction;
  - Un'abstraction deve delegare la maggior parte del lavoro all'Implementor che contiene;

I Design pattern comportamentali

I modelli comportamentali si occupano di una comunicazione efficace e dell'assegnazione delle responsabilità tra gli oggetti.

State

- **Intento:** Permettere ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. Far sembrare che l'oggetto abbia cambiato la sua classe.
- **Problema:** Tale pattern è strettamente correlato al concetto di *macchina a stati finiti*:
  - Il comportamento di un oggetto dipende dal su stato e il comportamento deve cambiare a run-time in base al suo stato.
  - Le operazioni da svolgere hanno vari grandi rami condizionali che dipendono dallo stato dell'oggetto.
  - Lo stato è generalmente rappresentato da una o più costanti enumerative.
  - Spesso diverse operazioni conterranno la stessa struttura condizionale.
- **Soluzione:** Mette ogni ramo condizionale in una classe separata. Ciò consente di trattare lo stato dell'oggetto come un oggetto a sé stante che può variare indipendentemente dagli altri oggetti.
  Per tale scopo il design pattern State prevede le seguenti caratteristiche:
  - **Context:** l'interfaccia con cui comunica il client. Mantiene un riferimento di tipo State che contiene un particolare ConcreteState, alla quale il Context delegherà la maggior parte del lavoro;
  - **State:** un'interfaccia che dichiara tutti i metodi relativi agli stati;
  - **ConcreteState:** una classe che implementa l'interfaccia State e implementa tutta la logica appropriata dei metodi a seconda dello stato che descrive;
  - Le **transazioni di stato** del Context vengono effettuate dal Context stesso o dallo stato corrente a seconda delle circostanze;
- **Struttura:**
  Diagramma delle classi:

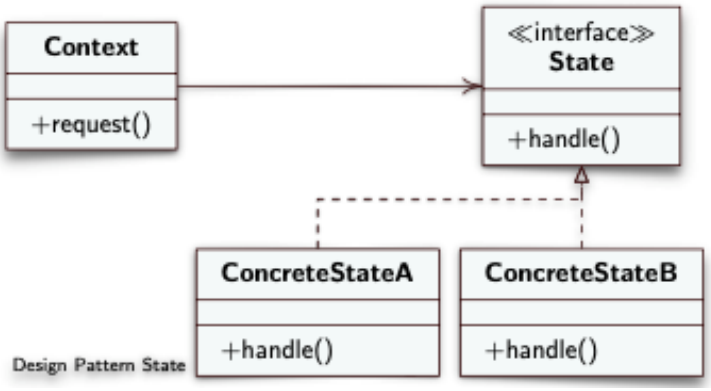
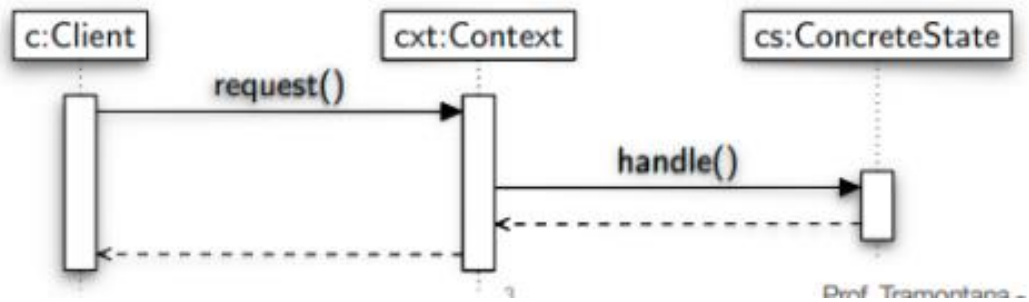


  Diagramma di sequenza:



- **Conseguenze:**
  - ✓ Il comportamento di Context è suddiviso negli svariati ConcreteState;
  - ✓ Incapsulare le azioni di uno stato in una classe impone una struttura e rende più chiaro lo scopo del codice;
  - ✓ Essendo Il comportamento associato ad uno stato localizzato in una sola classe *ConcreteState* è possibile aggiungere nuovi stati e transizioni semplicemente creando nuove sottoclassi di State;
  - ✓Ogni ConcreteState conosce il comportamento di un certo stato, questo permette di avere un codice più facile da leggere;
  - × Il numero di classi totale è maggiore, ma sono più semplici;
  - × L'uso del design pattern State potrebbe risultare un'esagerazione qualora il Context preveda pochi stati;
- **Dettagli Implementavi:**
  - Il ruolo di Context può essere affidato a una classe esistente, se ha già delle dipendenze dallo stato, o a una nuova classe, se il codice specifico dello stato è distribuito su più classi;
  - Viene dichiarata l'interfaccia State e implementati diversi ConcreteState a seconda degli stati che possiede il Context;
  - La regola basilare per poter selezionare il codice che appartiene a uno stato è leggere le condizioni che lo inglobano; una volta selezionato è sufficiente spostarlo all'interno del metodo del ConcreteState specifico e sostituire la porzione di codice con un richiamo del metodo da parte dello stato corrente;

Observer (o Publish-Subscribe)

- **Intento:** Implementare un meccanismo di notifica ad oggetti multipli rispetto allo stato di un oggetto osservato.
- **Problema:** Un effetto collaterale comune del partizionamento di un sistema in una raccolta di classi cooperanti è la necessità di mantenere la consistenza tra gli oggetti correlati. Tale consistenza non può essere ottenuta rendendo le classi strettamente accoppiate, poiché ciò ridurrebbe la loro riusabilità.
  Esempio: Immaginiamo la situazione in cui un cliente è interessato all'uscita di un nuovo modello di un prodotto venduto in un negozio, quindi va ogni giorno in sede a fare un controllo. La maggior parte dei viaggi fatti dal cliente saranno vani. Una soluzione efficace potrebbe consistere nell'invio di una newsletter non appena il prodotto diventa disponibile in negozio.

  L'Observer descrive, quindi, come stabilire relazioni fra oggetti dipendenti.
  Utilizzare l'Observer quando:
  - Un'astrazione ha due aspetti (es. dati e presentazione), uno dipendente dall'altro, incapsulare tali aspetti in oggetti separati permette di riusarli indipendentemente;
  - Un oggetto deve notificare un evento ad altri oggetti senza conoscere quest'ultimi, evitando così l'accoppiamento forte fra gli oggetti;
- **Soluzione:** L'oggetto i cui cambiamenti di stato vengono osservati viene chiamato *Subject* o, dato che andrà a notificare i propri cambiamenti, può anche essere chiamato *Publisher*. Tutti gli altri oggetti che vorranno tenere traccia dei cambiamenti del subject (quindi essere notificati) verranno chiamati *Observer* o, dato che sottoscrivono ad un flusso di notifiche, *Subscriber*.
  Il design pattern Observer prevede le seguenti caratteristiche:
  - **Subject:** una classe che contiene una lista dei suoi Observer, implementa i metodi per poter aggiungere, rimuovere e notificare gli Observer;
  - **Observer:** un'interfaccia comune a tutti gli oggetti che sono interessati a ricevere notifiche da un subject, a tale scopo definisce il metodo *update*();
  - **ConcreteSubject:** una classe che estende la classe Subject, mantiene al suo interno lo stato che interessa agli Obsever e notifica quest'ultimi attraverso i metodi ereditari dalla classe Subject;
    - Ad ogni variazione dello stato esegue il proprio metodo *notify()* che si occuperà si chiamate il metodo *update()* di ogni Observer in lista;
  - **ConcreteObserver:** una classe che implementa l'interfaccia Observer, tiene un riferimento al ConcreteSubject con il quale ottiene interagisce al momento della notifica;
    - A seconda del comportamento del metodo *update*() possiamo indentificare: una metodologia *Push* quando tale metodo ha come parametro il nuovo stato; una metodologia *Pull* quando tale metodo scatena una logica interna che eventualmente farà richiesta del nuovo stato attraverso il riferimento al ConcreteSubject interno;

- **Struttura:**
  Diagramma delle classi:

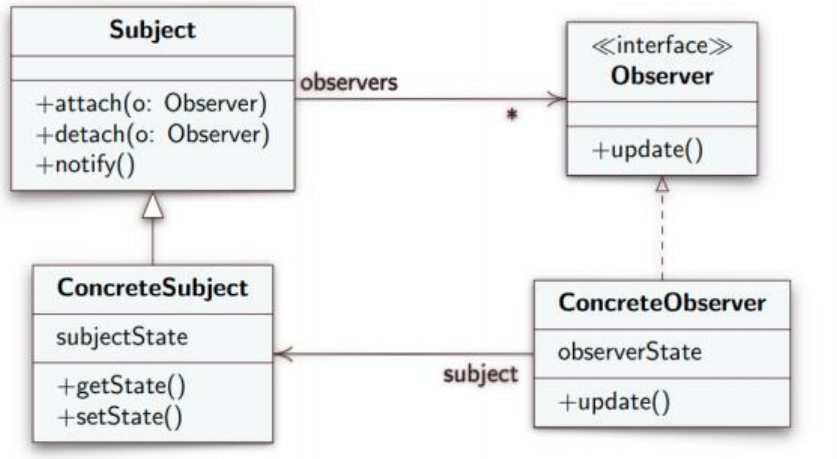
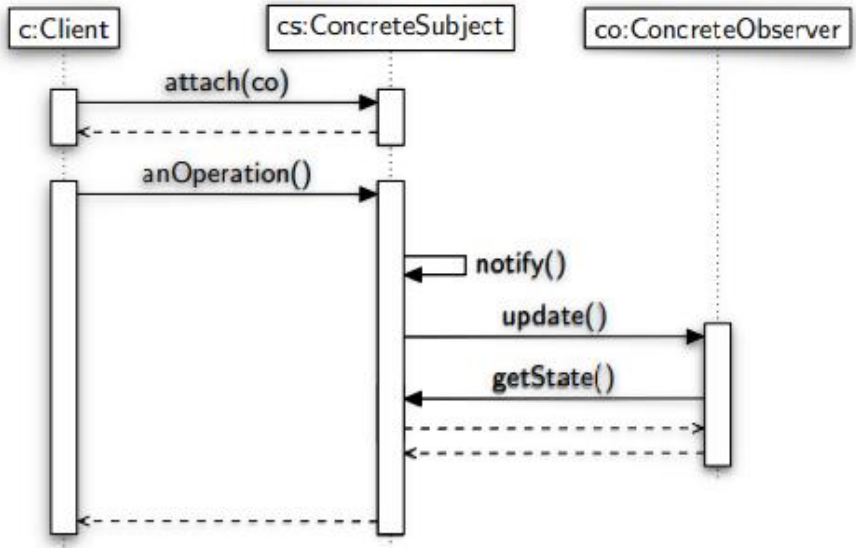


  Diagramma di sequenza:



- **Conseguenze:**
  - ✓ Il subject conosce solo la classe Observer e non ha bisogno di conoscere le classi ConcreteObserver;
  - ✓ ConcreteSubject e ConcreteObserver non sono accoppiati quindi più facili da riusare e modificare;
  - ✓ La notifica inviata da un ConcreteSubject è mandata a tutti gli ConcreteObserver in lista, ignorandone la quantità;
  - ✓ In un Subject, i ConcreteObserver possono essere aggiunti o rimossi in qualunque momento;
  - ✓ Il ConcreteObserver può scegliere se gestire o ignorare una specifica notifica;
  - ✓ L'aggiornamento da parte del Subject può far avviare tante operazioni sugli Observer per gli aggiornamenti;
- **Dettagli Implementativi**:
  - La notifica non dice agli Observer cosa è cambiato nel ConcreteState, è un evento che indica il completamento di un'operazione del ConcreteSubject;

- Gli oggetti che si sottoscrivono al subject potrebbero specificare gli eventi di interesse in modo da esser notificati solo alla loro occorrenza;
- Quando si vuole eliminare un subject, gli observer dovrebbero essere notificati per cancellare il loro riferimento ad esso;
- Un observer potrebbe voler osservare più oggetti. Per distinguerli al momento dell'update, si potrebbe passare come parametro un riferimento al Subject.

Observer in Java

Il problema affrontato del design pattern Observer è così comune che la sua soluzione è fornita nella libreria "*java.util"* con le classi *Observable* e *Observer*:

- **Observable** svolge il ruolo di Subject e notifica gli Observer tramite il metodo notifyObservers(). La classe ha una variabile flag che indica se lo stato è cambiato dalla precedente notifica, tale flag è impostata attraverso il metodo setChanged();
- **Observer** è un'interfaccia che ha solo il metodo *update*(), che può avere come argomento un oggetto Osservable per identificare chi ha scatenato l'aggiornamento;

Tuttavia, quando un Subject effettua la chiamata a notifyObservers scatena l'aggiornamento degli Observer sul suo stesso thread causando una perdita del controllo fino all'ultimazione dei processi di aggiornamento degli Observer.
Con l'avvento di Java 9, le classi Observable e Observer sono state deprecate dall'introduzione dei "*Reactive Stream*", una soluzione alternativa e migliore che non risolve il problema della perdita del controllo.

Reactive Streams

Java 9 con la libreria *"java.util.concurrent"* fornisce un set di interfacce e classi per risolvere il problema del passaggio di un insieme di item da un publisher a un subscriber senza bloccare il publisher e senza inondare il subscriber:

- **Publisher<T>**: un'interfaccia che definisce chi vuole mandare *item*, il suo metodo subscribe() è analogo al metodo attach() della classe Subject dell'Observer, prende come parametro un tipo Subscriber<T>, e l'invio degli item viene eseguito dal metodo *submit()*;
- **SubmissionPublisher:** una classe che implementa l'interfaccia Publisher, l'invio degli item ai subscriber avviene in modo asicrono. Quando un SubmissionPublisher esegue il metodo *submit()* esso crea un thread asicrono in cui richiama il metodo *OnNext()* di tutti i subscriber ad esso collegati;
- **Subscriber<T>**: un'interfaccia che definisce chi può riceve *item,* definisce i metodi onComplete(), onError(), onNext(), onSubscribe();
- **Subscription:** un'interfaccia che definisce il collegamento tra il publisher e il subscriber, presenta i metodi *cancel()* e *request(n)*:
    - Il Subscriber utilizza il metodo *cancel()* per fermare l'invio di messaggi, eliminando la sottoscrizione analogamente al detach() dell'Observer;
    - Il Subscriber utilizza il metodo *request(n)* per richiedere l'invio di messaggi;

Model View Controller (MVC) [Non segue lo schema dei pattern]

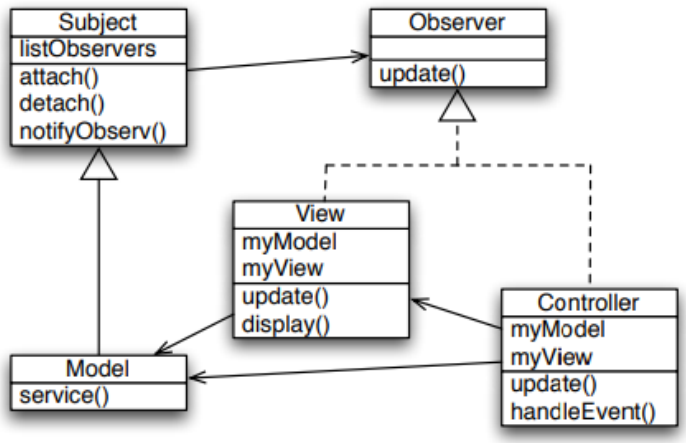
Tale pattern viene a volte chiamato design pattern e a volte pattern architetturale, inoltre possiamo dire che rappresenta una variante design pattern Observer.
Si applica in applicazioni interattive, che devono far visualizzare qualcosa, devono reagire ad eventi esterni e hanno bisogno di tenere dei dati che dipendono da queste interazioni e da algoritmi interni del sistema software.
Utilizziamo l'MVC quando:

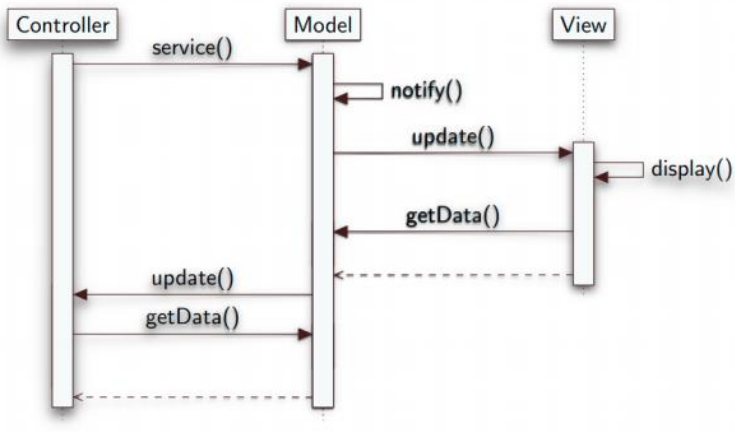
- Le interfacce utente possono variare al cambiare delle funzionalità, dei dispositivi e delle piattaforme;
- Le stesse informazioni sono presentate in finestre differenti;
- Le visualizzazioni devono subito adeguarsi alle manipolazioni sui dati;
- Le modifiche all'interfaccia devono risultare semplici;
- Il supporto ai diversi modi di visualizzazione non dovrebbe avere a che fare con le funzionalità principali;

L'MVC prevede:

- Il **Model:** incapsula le funzionalità principali e i dati dell'applicazione, è indipendente dalla rappresentazione degli input e degli output, si occupa di registrare View e Controller, avvisa quest'ultimi dei cambiamenti dei dati;
    - Nel contesto del design pattern Observer, tale elemento svolge il ruolo di Subject;
- La **View:** mostra i dati che legge dal Model all'utente, ogni view è associata a un Controller;
    - Nel contesto del design pattern Observer, tale elemento svolge il ruolo di Observer;
- Il **Controller:** riceve gli input dall'utente sotto forma di eventi, traduce quest'ultimi in richieste di servizio per Model o per View;
    - Nel contesto del design pattern Observer, tale elemento svolge il ruolo di Observer;

Diagrammi:

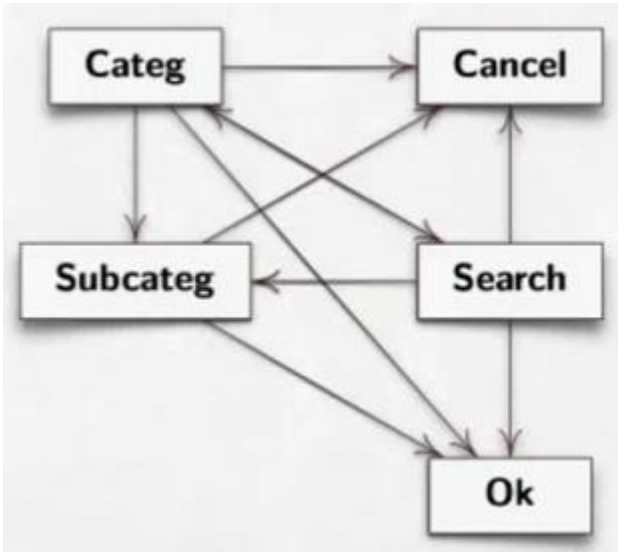




Mediator

- **Intento:** Ridurre le dipendenze, ovvero promuovere un accoppiamento lasco, tra gli oggetti definendo un oggetto mediatore che incapsula le loro interazioni. Gli oggetti comunicano fra loro indirettamente attraverso il mediatore.
- **Problema:** La programmazione ad oggetti promuove la distribuzione delle responsabilità del sistema su più oggetti che, nel caso peggiore, risultano interdipendenti.
  Il vantaggio del riuso del codice derivante dalla distribuzione delle funzionalità su più oggetti viene limitato dalle forti interconnessioni tra quest'ultimi, che instaurando un accoppiamento forte tra le componenti del sistema fanno sì che esso tenda ad essere monolitico.

  Esempio: Si supponga di dover costruire una finestra di dialogo per effettuare una ricerca composta da molti elementi come: campi di testo, caselle di controllo e pulsanti. Tali componenti sono controllati da una rispettiva classe.
  L'azione dell'utente su uno specifico elemento può causare un cambiamento sugli altri, per esempio scegliendo una ricerca su categoria è necessario che il componente per la ricerca testuale sia disattivato e quello per la sotto-categoria attivato.
  Senza l'utilizzo di un mediatore si ha che ogni componente conosce le altre, ovvero ogni classe dipende dalle altre, e che la logica delle interazioni è distribuita fra le varie classi, ottenendo così un accoppiamento forte.



  Utilizzare il Mediator quando:
  - Un insieme di oggetti comunicano in modo ben definito ma complesso. Le interdipendenze che ne risultano sono non strutturate e difficili da comprendere;
  - Riusare un oggetto è difficile, poiché esso comunica con tanti altri oggetti;
  - Un comportamento che è distribuito fra tante classi dovrebbe essere modificabile senza dover ricorrere a sottoclassi;
- **Soluzione:** Il Mediator prevede le seguenti caratteristiche:
  - **Colleague**: Un'interfaccia comune per tutti gli oggetti che necessitano di comunicare, contiene un riferimento a un Mediator che usa qualora l'oggetto volesse comunicare, legandosi al tipo generico Mediator è possibile riutilizzarlo in diversi contesti;

- **ConcreteColleague**: una classe che implementa l'interfaccia Collegue definisce la logica di business degli oggetti del sistema;
- **Mediator**: un'interfaccia che dichiara i metodi di comunicazione fra le componenti;
- **ConcreteMediator**: una classe che implementa l'interfaccia Mediator, implementa il comportamento cooperativo e coordina la comunicazione fra i vari Colleague, a tale scopo esso conosce tutti i Colleague interessati a comunicare, ovvero ne possiede i riferimenti;

- **Struttura:**

Diagramma delle classi:

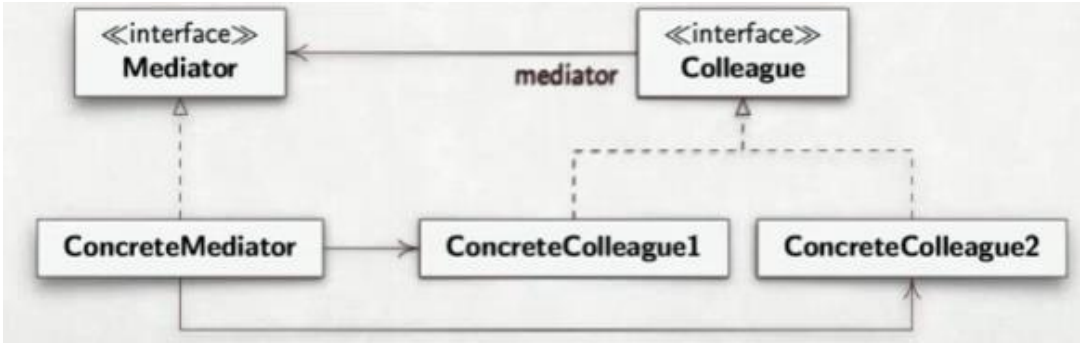
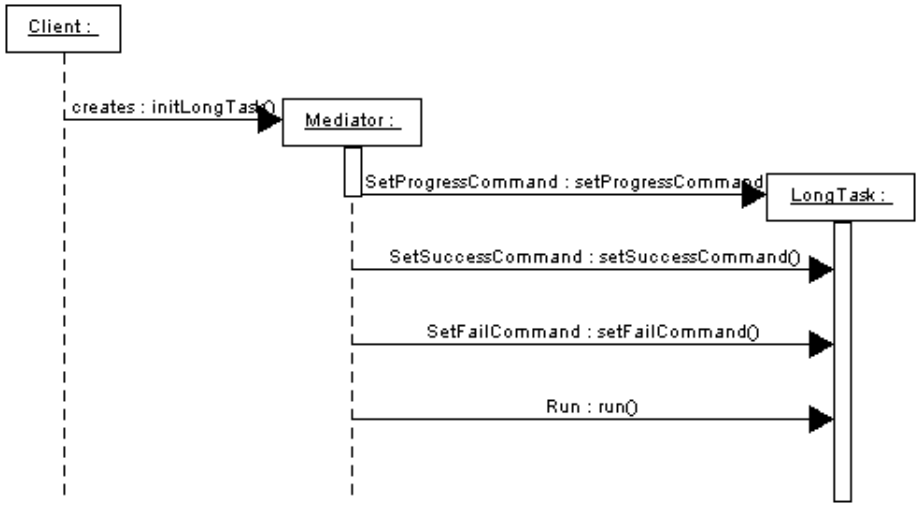


Diagramma di sequenza:



- **Conseguenze**
  - ✓ La maggior parte delle complessità che risulta nella gestione delle dipendenze è spostata dagli oggetti cooperanti al Mediator, ciò rende più facile implementare nuovi Colleague o nuovi Mediator;
  - ✓ Le classi Colleague sono più riusabili poiché la loro funzionalità fondamentale non è mischiata con il codice che gestisce le dipendenze;
  - × Il Mediator non è in genere riusabile poiché la gestione delle dipendenze implementata è specifica per l'applicazione;
- **Dettagli Implementativi**:
  - Il Mediator differisce dal Facade poiché l'astrazione di quest'ultimo prevede comunicazioni unilaterali dall'oggetto Facade alle componenti, e non bidirezionale così come prevede il Mediator;
  - La comunicazione fra Mediator e Collegues potrebbe essere implementata sfruttando il pattern design Observer;

Chain of Responsability

- **Intento:** Disaccoppiare il mandante di una richiesta con il ricevente facendo passare le richieste lungo una catena di gestori che possono decidere se elaborarla o passarla al gestore successivo, dando così la possibilità a più di un oggetto di gestire la richiesta.
- **Problema:** Si supponga di avere un applicativo che permette agli utenti di eseguire ordini, la conferma di un ordine prevede che i seguenti requisiti siano soddisfatti:
  - L'utente deve essere autenticato;
  - La merce deve essere disponibile;
  - L'utente deve essere autorizzato ad acquistare la merce desiderata;
  - Le informazioni sulle modalità di pagamento e spedizione devono essere confermate;

  Utilizzare il Chain of Responsability quando:
  - Si deve gestire una richiesta che necessità di numerosi servizi;
- **Soluzione:** Il Chain of Responsability prevede le seguenti caratteristiche:
  - **Handler**: Un'interfaccia comune a tutti gli oggetti che possono gestire una richiesta, contiene un riferimento di un altro Handler che nella catena risulta essere il successore;
  - **ConcreteHandler**: una classe che implementa l'interfaccia Handler e gestisce le richieste per cui è responsabile, ha accesso al suo Handler successore a cui inoltra la richiesta ricevuta se non è capace di gestirla;
  - **Client**: il nostro applicativo che effettua richiesta ad un ConcreteHandler;
- **Struttura:**

  Diagramma delle classi:

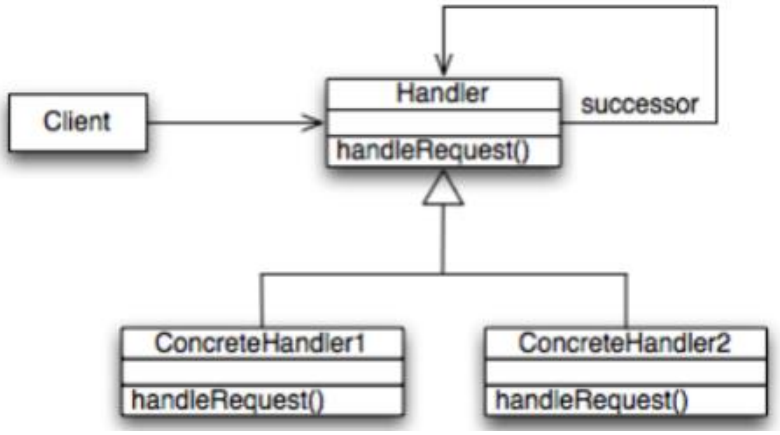
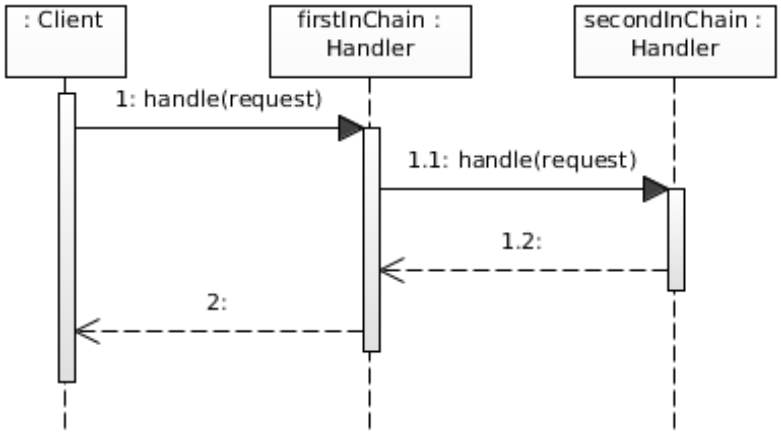


  Diagramma di sequenza:



- **Conseguenze**
  - ✓ Promuove l'accoppiamento lasco fra le componenti, chi fa la richiesta non conosce il ricevente e viceversa;
  - ✓ Si ottiene una certa flessibilità nella distribuzione delle responsabilità agli oggetti, in quanto è possibile aggiungere o rimuovere responsabilità nella gestione di una richiesta cambiando la catena in runtime;
  - × Non c'è garanzia che una richiesta venga gestita: non essendoci un ricevente specifico, la richiesta potrebbe raggiungere la fine della catena senza essere gestita;
- **Dettagli Implementativi**:
  - I vari gestori non devono conoscere la struttura della catena, necessitano di conoscere soltanto il successore a cui, eventualmente, inoltrare la richiesta;
  - Handler o ConcreteHandler possono definire i link per avere il successore della catena;
  - È possibile usare una gerarchia esistente, come il composite, come catena se la gerarchia riflette quest'ultima;

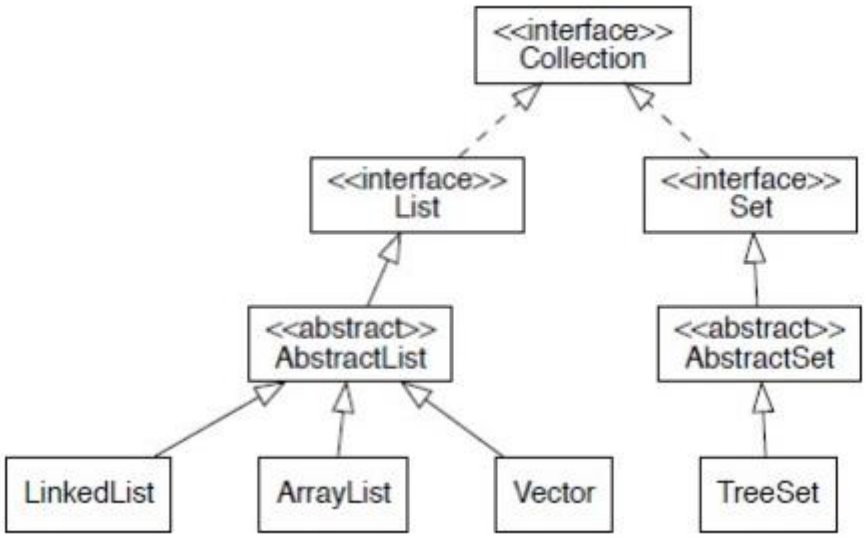
Java e la programmazione funzionale

La programmazione funzionale ci permette di ottenere dei vantaggi quando si scrive del codice, ovvero il codice è più conciso rispetto ad una programmazione imperativa e riesce a comunicare più facilmente il significato.
Un ulteriore vantaggio della programmazione funzionale è la facile parallelizzazione per tratte il massimo dall'hardware sottostante.
Java con la versione 8 ha introdotto alcune caratteristiche della programmazione funzionale.

Java Collection

Le librerie Java predispongono di tipi che implementano le strutture dati più famose. Attraverso questi tipi è possibile gestire insiemi delle classi da noi definite.

In Java una delle librerie più famose è la *Java Collections*, al suo interno è definita un'interfaccia *Collection* che definisce i metodi comuni add(), remove(), size(), contains(), containsAll(), ect…, delle principali strutture dati.



- L'interfaccia *List* estende l'interfaccia Collection definendo altri metodi tra cui: get(), set(), indexOf(). Si differenzia dall'interfaccia *Set* poiché definisce le basi del comportamento di una collezione ordinata tramite indici. Parte di questa interfaccia viene implementata attraverso la classe AbstractList, da cui derivano:
  - ArrayList, una collezione che permette un accesso rapido agli elementi poiché, come dice il nome, sfrutta una sezione contigua in memoria per salvare gli elementi, come un array;
  - LinkedList, una collezione che offre un accesso lineare agli elementi, non vi è contiguità in memoria, ma offre un inserimento e una cancellazione di complessità costante;
  - Vector, una collezione simile all'ArrayList con un approccio *synchronized*, ovvero dispone di un lock per gestire eventuali concorrenze;
- L'interfaccia *Set* definisce un insieme di elementi univoci. Per esempio l'implementazione TreeSet sfrutta una struttura ad albero;

Segue una tabella riassuntiva delle diverse complessità dei principali metodi per le concretizzazioni di List:

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

Classi e Funzioni anonime

Le classi anonime

Da Java 1.1 è possibile utilizzare le classi anonime per poter implementare un'interfaccia senza dover creare una classe apposita, fornendo le implementazioni al momento dell'inizializzazione della classe anonima.
Segue un esempio di classe anonima che implementa un'interfaccia:

```
public interface Hello {
    public void greetings (String name);
}
public class Context {
    public static void main (String[] args) {
        Hello greeter = new Hello() {
            @override
            public void greetings (String name) {
                System.out.println("Hello " + name);
            }
        };
    }
}
```

Le funzioni anonime e le espressioni lambda

Una funzione anonima è una porzione di codice eseguibile, al quale non si è dato un nome e non è stata incapsulata all'interno di un metodo.

Un'espressione lambda è una funzione anonima che: prende zero o più parametri, il cui nome è indicato a sinistra della freccia fra parentesi tonde (omettibili) e presenta un blocco di codice alla destra della freccia. Una funzione lambda è una funzione pura, ovvero il suo risultato dipende totalmente dagli input. Il tipo di ritorno è generalmente determinato automaticamente e ciò che ritorna è il risultato del blocco di codice presente alla destra della freccia.
Esempio:

```
(x, y) -> x + y
```

```
str -> System.out.println(str)
```

Definire una classe anonima tramite espressione lambda

Con Java 8 è possibile sfruttare le espressioni lambda per implementare un'interfaccia che possiede ***un solo metodo,*** il corpo dell'espressione lambda costituirà il comportamento del metodo. Prendendo come esempio l'interfaccia precedentemente descritta:

```
Hello greeter = name -> System.out.println("Hello " + name);
```

Il tipo di *name* e il tipo di ritorno saranno automaticamente presi dall'interfaccia che estende.

La programmazione funzionale e lo stile dichiarativo

La programmazione funzionale amplia quella che viene chiamata programmazione dichiarativa.

Approccio Imperativo vs Approccio Dichiarativo

Nell'approccio imperativo per capire l'obbiettivo di un metodo bisogna leggere tutte le righe di codice che lo compongono.
Nell'approccio dichiarativo semplicemente si richiamano metodo espressivi che con il loro solo nome indicato il loro obbiettivo.
Segue una comparazione degli approcci in una ricerca su lista:

```
public class Trova {

  private List<String> nomi = new Arrays.asList("Nobita", "Nobi", "Suneo",
 "Honekawa", "Shizuka", "Minamoto", "Takeshi", "Gouda");

  //In stile imperativo
  public void trovaImper() {
      boolean trovato = false;
      for (String nome : nomi)
          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(nomi.contains("Nobi")) System.out.println("Nobi trovato");
      else System.out.println("Nobi non trovato");
  }
 }
```

Lo stile funzionale

Lo stile di programmazione funzionale è dichiarativo. La programmazione funzionale aggiunge allo stile dichiarativo funzioni di ordine più alto, ovvero funzioni che hanno come parametri altre funzioni. In Java, in generale, si possono passare oggetti ai metodi, creare oggetti dentro i metodi, e ritornare oggetti dai metodi. In Java 8 si possono passare funzioni ai metodi, creare funzioni dentro i metodi e ritornare funzioni dai metodi.

Le interfacce funzionali

Ogni interfaccia con un singolo metodo astratto è detta *interfaccia funzionale* ed è denotata dalla annotazione *@FunctionalInterface*. L'annotazione consente al compilatore di validare l'interfaccia controllando che soddisfi i requisiti per essere definita funzionale, ovvero avere un solo metodo astratto.

Possiamo definire l'implementazione di una interfaccia funzionale attraverso le classi anonime, come visto precedentemente.
Esempio di interfaccia funzionale:

```
@FunctionalInterface
public class FunctionalInterfaceExample {
    public void abstractMethod ();
}
```

Predicate

La classe Predicate è un'interfaccia funzionale che implementa il metodo *test*:

```
public boolean test (T t);
```

Tale metodo prende in input un tipo generico T e restituisce un boolean.

Segue un esempio di implementazione sfruttando le espressioni lambda:

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

Function

La classe Function è una interfaccia funzionale che implementa il metodo *apply*:

```
public R apply(T arg);
```

Tale metodo necessita di due generics T e R che rappresentano rispettivamente il parametro in ingresso ed il tipo di ritorno.

Segue un esempio di implementazione sfruttando le espressioni lambda:

```
Function<String, Integer> getStringLength = str -> str.length;
```

Supplier

La classe Supplier è una interfaccia funzionale che implementa il metodo *get*:

```
public T get();
```

Tale metodo non necessita di nessun parametro in ingresso e restituisce un elemento di tipo T, funge quindi da generatore di dati.

Segue un esempio di implementazione sfruttando le espressioni lambda:

```
Supplier<String> doubleGreet = "ciao ".repeat(2);
```

Lo Stream

Lo Stream è un particolare oggetto che consiste in una sequenza di elementi di tipo generico T su cui poter eseguire alcune operazioni, ottenibile in svariati modi:

- Invocando il metodo *stream()* di una collezione;
  - Il metodo stream() è un *metodo di default dell'interfaccia Collection*, ovvero un metodo implementato nell'interfaccia ma che non può agire sullo stato dell'oggetto;
- Fornendo una collezione come parametro al metodo statico *of()* della classe Stream;
- Usando il metodo statico *iterate(seme, function)* della classe Stream che restituisce uno stream infinito ed ordinato dei risultati di una funzione applicata inizialmente a un seme;
  - È possibile limitare la dimensione dello stream attraverso il metodo *limit(n);*
  - Eccetto per il primo elemento, l'input della funzione è il risultato precedentemente elaborato;
- Usando il metodo statico *generate(function)* della classe Stream che restituisce uno stream infinito di elementi ottenuti da una funzione Supplier;
  - È possibile limitare la dimensione dello stream attraverso il metodo *limit(n);*
  - La generazione degli elementi è totalmente indipendente;

Proprietà dei metodi dello Stream

I metodi offerti da uno Stream si suddividono obbligatoriamente in due tipi:

- **Lazy**: metodi che restituiscono oggetti Stream, è quindi possibile concatenarli. Devono il loro nome al loro comportamento, non vengono eseguiti fintanto che non viene eseguito un metodo *eager* o *terminale*;
- **Eager**: metodi anche detti *terminali* poiché "terminano" la catena di metodi lazy, avviandone l'esecuzione, e restituiscono un risultato. L'esecuzione di un metodo terminale "consuma" lo stream, ovvero non sarà più utilizzabile per ulteriori metodi;

Inoltre, i metodi offerti da uno Stream possono essere classificati come:

- **Stateless:** quando non hanno uno stato interno, prendono un elemento dallo Stream e, se deve, ritorna qualcosa;
- **Stateful:** quando per operare necessità di conoscere lo stato dello Stream, ovvero conoscerne tutti gli elementi;

Varianti dello Stream

Java propone alcune varianti dello Stream:

- ParallelStream: una variante dello Stream pensata per operare in modo parallelo, ottenibile attraverso *parallelStream*(), metodo di default dell'interfaccia Collection;
- IntStream: una variante dello Stream specializzata nell'operare con i numeri interi, offre ulteriori metodi per operare con quest'ultimi:
  - *sum():* metodo eager che restituisce la somma dei numeri contenuti;
  - *rangeClosed(inizio, fine):* restituisce un IntStream contenente una sequenza di interi nell'intervallo specificato, con estremi inclusi, a incrementi di 1;

  Si ottiene mediante il metodo statico *rangeClosed()* della classe IntStream o il metodo *mapToInt()* della classe Stream;

Alcuni metodi dello Stream

filter

Il metodo *filter* è un metodo *lazy e stateless* che permette di filtrare una lista in base a un Predicate fornito come parametro.
Esempio:

```
List names = List.of("Alice", "Bob", "Alice", "Fred");

Predicate<String> isAlice = name -> name.equals("Alice");

short aliceCounter = names.stream()
                        .filter(isAlice) // potremmo inserire direttamente la
funzione lambda
                        .count();        // metodo eager
```

count

Il metodo *count* è un metodo *eager* che permette di ottenere il numero di elementi presenti nello stream.

Reduce

Il metodo *reduce* è un metodo *eager* che permette di passare da un insieme di valori a un singolo valore sulla base dei precedenti. Tale metodo prende in ingresso due parametri: un variabile di partenza e una espressione lambda, quest'ultima prende in ingresso due valori: dove il primo è un accumulatore e il secondo si riferisce all'elemento estratto dallo Stream in un dato istante.
Reduce applica la funzione lambda passata a tutti gli elementi dello stream e accumula un risultato.
Esempio:

```
List numbers = List.of(3, 4, 5);

int sum = numbers.stream()
             .reduce(10, (acc, n) -> acc + n);

// sum = [10] + 3 + 4 + 5 = 22
```

map

Il metodo *map* è un metodo *lazy e stateless* che permette di passare da uno Stream<T> iniziale a uno Stream<R> sulla base della funzione *mapper* che prende in ingresso. Tale metodo chiama la funzione mapper su ogni elemento dello Stream<T> e raccoglie i risultati nello Stream<R>.
Esempio: Si passa da Stream<People> a Stream<Integer>

```
long total = people.stream()
                    .map(p -> p.getAge())
                    .reduce(0, Integer::sum());
```

peek

Il metodo *peek* è un metodo *lazy e stateless* utilizzato in ambito di debug per controllare lo stato degli elementi dello Stream in un determinato momento.

mapToInt

Il metodo *mapToInt* è un metodo *lazy e stateless* che permette di passare da uno Stream<Integer> iniziale a uno IntStream sulla base della funzione *mapper, la quale deve restituire solo interi,* che prende in ingresso. Tale metodo chiama la funzione mapper su ogni elemento dello Stream<Integer> e raccoglie i risultati nel IntStream.

Esempio:

```
int result =
Stream.of("truth", "flows", "to", "them", "sweetly", "by", "nature")
       .mapToInt(x -> x.length()).sum();

// Output: result = 31
```

mapToObj

Il metodo *mapToObj* è un metodo *lazy e stateless* che permette di passare da uno IntStream iniziale a uno Stream di oggetti sulla base della funzione *mapper* che prende in ingresso. Tale metodo associa ad ogni numero contenuto nell'IntStream un oggetto che verrà inserito nel nuovo Stream.

Esempio:

```
// Si abbia la lista che contiene istanze di Persona, si generi uno stream
// contenente i primi 4 elementi
List<Persona> lista;

Stream<Persona> p = IntStream.rangeClosed(0, 3)
                            .mapToObj(i -> lista.get(i));
```

boxed

Il metodo *boxed* è un metodo *lazy e stateless* che permette di passare da uno IntStream iniziale a uno Stream<Integer>

Esempio:

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

sorted

Il metodo *sorted* è un metodo *lazy e stateful* che permette di ordinare gli elementi di Stream di elementi non primitivi (oggetti) in base al Comparator passato in ingresso.

Esempio:

```
students.stream()
    .filter(s -> s.getAge() < 20)
    .ordered(Comparator.comparing(Person::getName))
    .forEach(s -> System.out.println(s.getName));
```

ordered

Il metodo *ordered* è un metodo *lazy e stateful* che permette di ordinare gli elementi di Stream di elementi primitivi.

Esempio:

```
students.stream()
    .filter(s -> s.getAge() < 20)              // Stream<Person>
    .map(s -> s.getName())                     // Stream<String>
    .ordered()                                 // l'ordinamento è noto
    .forEach(str -> System.out.println(str);
```

distinct

Il metodo *distinct* è un metodo *lazy e stateful* che permette di ottenere uno Stream di soli elementi univoci.

forEach

Il metodo *forEach* è un metodo *eager* che permette di iterare fra gli elementi dello Stream applicando ad essi la funzione passata in ingresso.
Esempio:

```
students.stream()
    .filter(s -> s.getAge() < 20)              // Stream<Person>
    .map(s -> s.getName())                     // Stream<String>
    .ordered()                                 // l'ordinamento è noto
    .forEach(str -> System.out.println(str);
```

collect

Il metodo *collect* è un metodo *eager* che permette di passare da uno Stream<T> a una collezione di elementi di tipo T. Tale metodo prende in ingresso un *Collector* per stabilire che tipo di collezione creare.
Esempio: Dopo aver eseguito le operazioni desiderate si vuole ottenere una List<Integer>

```
List<Integer> ages = students.stream()
                        .map(p -> p.getAge())
                        .collect(Collector.toList());
```

findAny

Il metodo *findAny* è un metodo *eager* che permette di ottenere un qualunque elemento dello Stream. Tale metodo può essere usato dopo un filtraggio per sapere se almeno un elemento rispetta la condizione del filtro. Dato che il risultato non è garantito, il metodo ritorna un elemento di tipo Optional.
Questo metodo gode della proprietà *short circuit* perché ha la capacità fermare le iterazioni del filtraggio non appena un elemento soddisfa il filtro.
Esempio:

```
Optional<Person> res = students.stream()
                    .filter(s -> s.getAge > 30)
                    .findAny();

if (res.isPresent())
    System.out.println(res.get().getName());
```

findFirst

Il metodo *findFirst* è un metodo *eager* che permette di ottenere il primo elemento dello Stream. È simile al *firstAny* con la sola differenza che in ambito parallelo bisogna aspettare la fine di tutti i thread per operare.
Esempio:

```
Optional<Person> res = students.stream()
                    .filter(s -> s.getAge > 30)
                    .findAny();

if (res.isPresent())
    System.out.println(res.get().getName());
```

I Processi di sviluppo del software

Le attività inerenti allo sviluppo del software non riguardano solo il codice, i test o la progettazione, ma riguardano anche altri aspetti. Un'azienda che vuole sviluppare software ha bisogno di conoscere le pratiche più comuni per lo sviluppo del software.

Queste pratiche sono raccolte all'interno di un processo software, che descrive le attività o compiti che sono necessari allo sviluppo di un prodotto software e come esse siano collegate fra loro.

I nomi delle principali attività sono:

- Analisi dei requisiti (specifiche)
- Progettazione (design)
- Implementazione (codice)
- Testing (approvazione)
- Manutenzione

Analisi dei requisiti

L'analisi dei requisiti è la fase che porta a definire le specifiche; stabilire i servizi richiesti ed i vincoli del software.
In questa fase trovano uso i seguenti termini:

- Un **requisito** è una descrizione generica e granulare di ciascuna delle caratteristiche che il software deve avere. Si distinguono in due categorie:
  - **Requisiti Funzionali**: descrivono cosa deve fare il software: *… deve effettuare una ricerca …;*
  - **Requisiti Non Funzionali:** descrivono come il software dovrebbe implementare determinate caratteristiche: *… deve effettuarla in meno di 10 secondi …;*
- Una **specifica** è una descrizione rigorosa di una caratteristica del software;
- Una **feature** è un insieme di requisiti correlati fra loro per il raggiungimento di un obbiettivo;

Una cattiva analisi dei requisiti aumenterebbe la percentuale di fallimento del progetto software: si rischia di non soddisfare le necessità dei clienti e quindi di aver svolto lavoro inutile.

Lo studio dei requisiti è un processo ingegneristico diviso in più sotto-fasi:

- **Studio di fattibilità:** durante la commissione di un software è necessario comprendere a grandi linee quali sono le necessità del cliente e valutare se, con le proprie risorse, è possibile soddisfarle in un tempo ragionevole e con costi sostenibili;
- **Analisi dei requisiti:** Una persona competente rispetto allo sviluppo del software raccoglie le informazioni (requisiti) dal cliente e le inserisce in un documento provvisorio;
- **Specifica dei requisiti:** I requisiti precedentemente raccolti vengono trasformati in specifiche rigorose e dettagliate, vengono ipotizzate tutte le possibili sfaccettature dei requisiti e i comportamenti del software rispetto ad esse;
- **Convalida dei requisiti:** Si analizzano le specifiche elaborate in presenza del cliente, al fine di risolvere eventuali contraddizioni rispetto a ciò che è stato richiesto;

Progettazione

La fase di progettazione mira a stabilire qual è la struttura software che realizza le specifiche raccolte. In essa vengono stabiliti: un nome per tutte queste componenti, con granularità via via più fine, la responsabilità di ciascun componente e le relazioni tra i vari componenti.

Anche la fase di progettazione si suddivide in più sotto-fasi:

- **Suddivisione dei requisiti:** I requisiti precedentemente accordati con il cliente vengono suddivisi in gruppi in base alla loro coesione (obbiettivo);
- **Identificazione di sottosistemi**: Si individuato dei sottosistemi attraverso i gruppi precedentemente stabiliti, si ha una correlazione fra i sottosistemi e i requisiti che deve soddisfare;
- **Specifica delle responsabilità dei sottosistemi**: Si descrivono sempre più in dettaglio i sottosistemi e i suoi componenti, fino ad ottenere un componente per uno o più requisiti. Viene applicato attentamente il principio di singola responsabilità;
- **Progettazione:** Si definiscono le interfacce, le classi e i metodi, dalla documentazione fino ad ora prodotta vengono elaborati i diagrammi UML con notazione standard;

Implementazione

La fase di implementazione consiste nel produrre un programma eseguibile basandosi sulle scelte effettuate durante la fase di progettazione.

Una volta scritto il codice, è necessario cercare e rimuovere eventuali errori presenti nell'implementazione. Attraverso varie tecniche, gli sviluppatori effettuano vari test (spesso automatizzati) per scoprire bug e rimuoverli. Il processo corretto per rimuovere un errore consiste nel localizzare l'errore, rimuoverlo dal modello e dopodiché rimuoverlo dal codice; infine eseguire di nuovo il test fallito.

Testing

La fase di testing, denominata anche "verifica e validazione" del sistema software consiste nel mostrare la conformità di esso rispetto alle specifiche richieste dal cliente.

Questa fase viene condotta attraverso processi di revisione e test del sistema software. I test mirano ad eseguire il sistema software in condizioni derivate dalle specifiche di dati reali che il sistema software dovrà elaborare. Nel caso sia l'azienda richiedente a fornirci dei dati reali su cui eseguire dei test, molto spesso si stipula un accordo di riservatezza (NDA) per evitarne la diffusione. Inoltre, il software verrà eseguito su macchine conformi a quelle utilizzate dal cliente.
Se la fase di testing viene superata con successo, lo sviluppo del software può ritenersi concluso con la consegna al cliente.

Verifica vs Validazione

- La *verifica* consiste nell'assicurarsi che il software sia conforme alle proprie specifiche;
- La validazione consiste nell'assicurarsi che il software soddisfi ciò che il cliente ha realmente richiesto;

Tipologie di test

I test implementati sul sistema sono vari:

- Unit test: si testano i singoli componenti (oggetti, funzioni, componenti);
- Test di sistema: si testa l'intero sistema, dando importanza alle priorità emergenti;
- Stress Test: un test che sottopone il sistema a un carico superiore a quello previsto per valutare possibili casi di perdite di servizio o dati;
- Alpha test: un test condotto da sviluppatori con i dati reali su applicativo con difetti e parti mancanti;
- Beta test: test condotto da alcuni clienti sul prodotto quasi completo, include tutte le feature, ma con difetti;

Manutenzione

Il software rilasciato va mantenuto o evoluto flessibilmente in base agli accordi stabiliti con il cliente. Al cambiare dei requisiti (business logic, hardware, etc), il software deve evolvere per continuare ad essere utile.

Manutenzione vs Evoluzione

Si distinguono le seguenti definizioni:

- La manutenzione è il processo di introduzione di modifiche ad un prodotto software dopo la sua consegna al cliente;
- L'evoluzione indica un singolo passo di un processo di manutenzione che comprende: l'evoluzione del software, il rilascio di una patch e la rimozione del vecchio sistema;

Modelli di manutenzione

- Modello quick-fix: cambiamenti rapidi a livello di codice che portano ad un deterioramento del software;
- Miglioramento iterativo: cambiamenti preceduti da una analisi approfondita del sistema esistente, mantenendo il design. Si aggiungono requisiti, si ri-progetta il sistema (secondo il vecchio design) e si riportano le modifiche sul codice;
- Riuso: manutenzione basata sul riuso delle componenti esistenti del vecchio sistema;

I costi di manutenzione

I costi di manutenzione rappresentano circa il 67-80% dei costi del software. Possiamo raggruppare i cambiamenti in 4 categorie:

- Correttivi: correggono errori del software;
- Adattivi: adattamento a nuovi ambienti (hardware, software) o a nuove normative;
- Perfettivi: miglioramenti del software e inserimento di nuove features;
- Preventivi: modifiche atte a prevenire problemi;

La maggior parte dei cambiamenti è perfettiva. I cambiamenti perfettivi sono sintomo di un software di successo poiché il cliente è soddisfatto del prodotto e richiede altre funzionalità.

Le leggi di Lehman

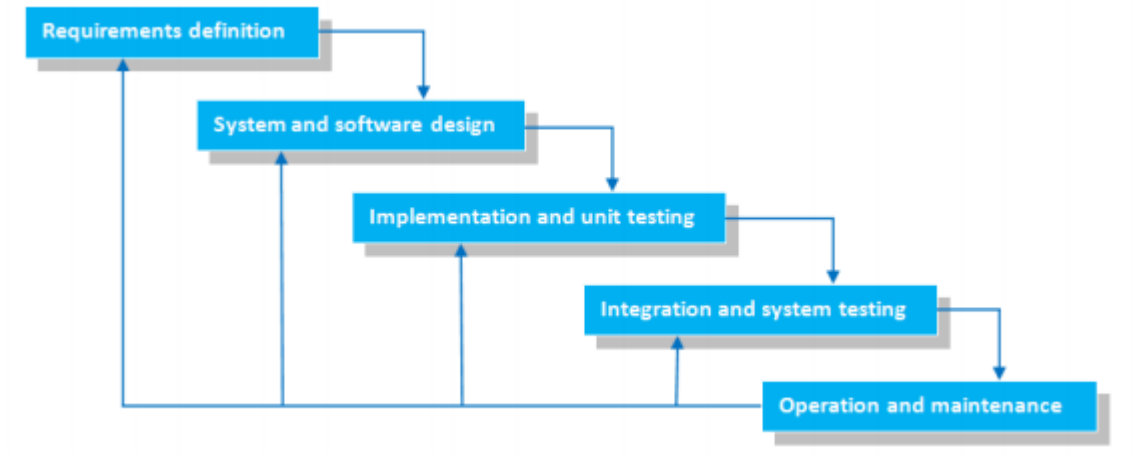
Lehman ha sviluppato delle leggi riguardo le dinamiche di evoluzione. Sono in generale applicabili a grandi sistemi sviluppati da grandi organizzazioni; non è chiaro come si adattano a piccoli prodotti.
Seguono le leggi sviluppate da Lehman tra il 1974 e il 1994:

- **Cambiamento continuo**: I sistemi hanno bisogno di essere continuamente adattati altrimenti diventano progressivamente meno soddisfacenti;
- **Aumento della complessità:** Quando un sistema evolve, la sua struttura aumenta di complessità, a meno di lavoro fatto per preservare o semplificare la sua struttura;
- **Auto-regolazione:** Attributi come dimensione, intervallo tra release e numero di errori trovati in ciascuna release sono approssimativamente invarianti;
- **Stabilità organizzativa:** Durante la vita di un sistema il suo tasso di sviluppo è circa costante e indipendente dalle risorse impiegate per lo sviluppo;
- **Conservazione di familiarità:** In media, l'incremento di crescita di un sistema tende a rimanere costante o a diminuire;
- **Continua crescita:** Il contenuto di funzioni di un sistema deve continuamente essere incrementato per mantenere la soddisfazione dell'utente;
- **Diminuzione della qualità:** La qualità di un sistema diminuisce se non viene rigorosamente gestita ed adattata durante i cambiamenti;

Alcuni processi software

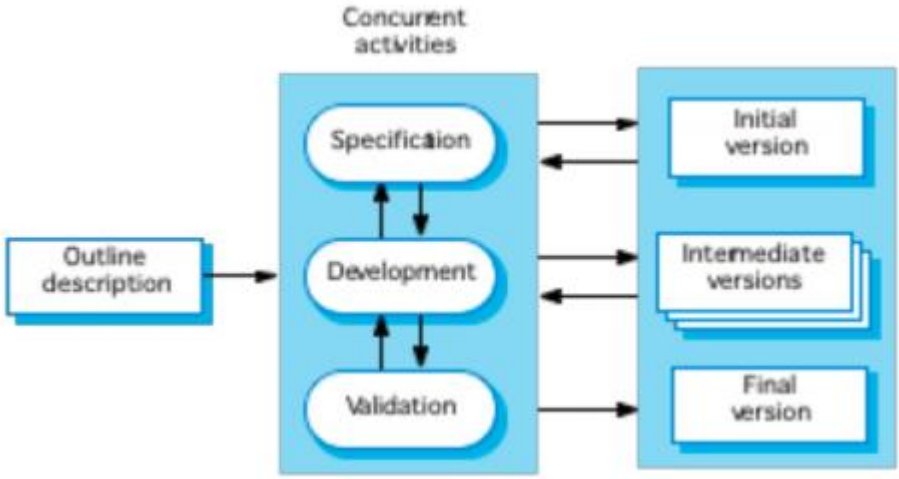
Modello a cascata (Waterfall)

Consiste nell'affrontare ciascuna fase senza tener conto delle altre, e solo quando essa sarà completata si passerà alla fase successiva, senza mai tornare indietro. Quando viene rilasciata una prima versione del software convalidato, allora si potrà tornare indietro ad una delle fasi precedenti per eventuali aggiustamenti, ma si dovrà proseguire con le fasi successive come visto precedentemente fino al rilascio di una nuova versione.
Notare che il dialogo con il cliente subentra solo nella prima fase del processo, di conseguenza la raccolta dei requisiti dovrà essere esaustiva e completa ed il cliente non potrà introdurre cambiamenti nelle fasi successive del processo. Permette di rilasciare molta documentazione per il software, a scapito del tempo di produzione. Tuttavia è un buon processo per software di qualità, di grandi dimensioni e/o con funzioni complesse e critiche.



Processo evolutivo

Il processo evolutivo prevede diverse revisioni del software prodotto sino ad arrivare a ciò che richiede il cliente.



Ne conosciamo principalmente due varianti:

- **Sviluppo per esplorazione**: Si parte con requisiti ben chiari e, lavorando di fianco al cliente, si arriva al sistema software finale per mezzo di trasformazioni successive (*evoluzioni*). Le caratteristiche vengono aggiunte man mano dal cliente;
- Build and fix: viene adottato quando si ha una comprensione limitata del sistema software da produrre, la fase di design è pressoché inesistente con conseguente documentazione poco dettagliata. Si costruisce una prima versione del software e si modifica sino a che il cliente non è soddisfatto. Il codice prodotto sarà ovviamente di bassa qualità;

Il processo evolutivo ha molti difetti, come i lunghi tempi di produzione, la produzione di sistemi difficilmente comprensibili/modificabili e non corretti e la mancanza di visione d'insieme del progetto.

Processo incrementale

Il processo incrementale divide i requisiti iniziali in più parti. Partendo dai requisiti di base (o fondamentali), si implementa una versione del software che li soddisfa. Dopodiché si passa al prossimo gruppo di requisiti e si revisiona il sistema per soddisfarli e così via. Si produce in maniera incrementale una versione del software più completa sino ad arrivare a quella finale;

Processo CBSE (o Processo COTS)

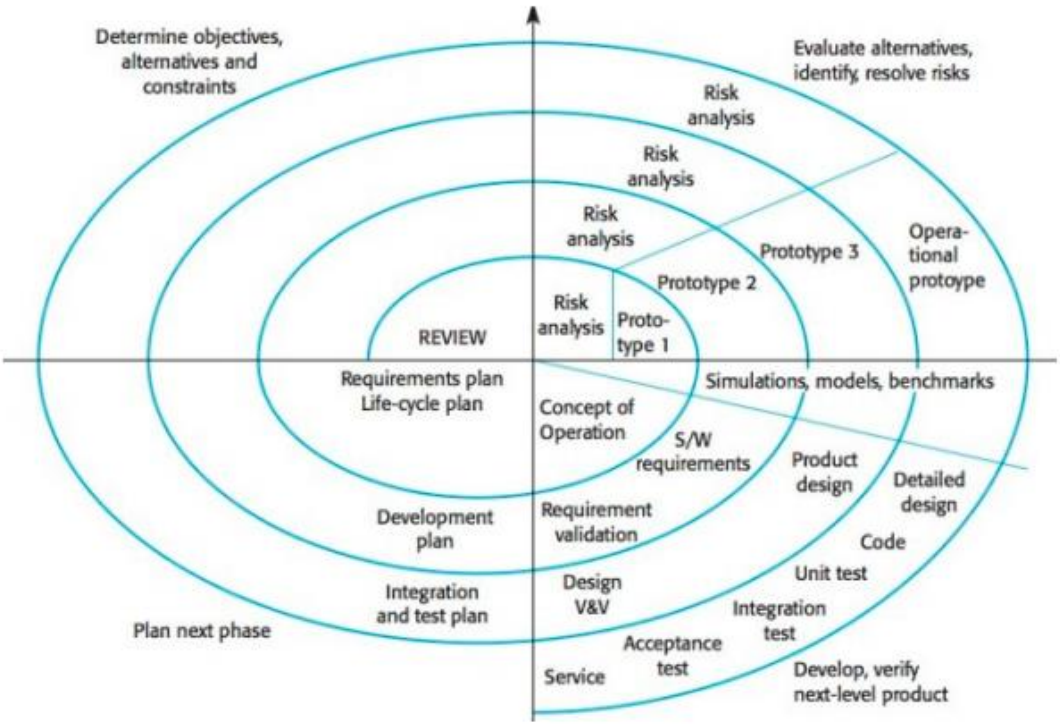
CBSE è l'acronimo di Component Based Software Engineering e consiste nello sviluppare e riutilizzare componenti indipendenti. Partendo dai requisiti del cliente si individuano delle componenti da realizzare, ma si analizzano anche le componenti pre-esistenti che potrebbero adattarsi al completamento del sistema software, di fatto una ulteriore nomenclatura del processo è COTS, ovvero Components Off The Shelf (componenti pronti all'uso).

Processo a Spirale

Come molti altri processi, si focalizza nella realizzazione di prodotti parziali per via via produrre un prodotto finale. La peculiarità è data da un processo a spirale, ovvero formato da cicli produttivi divisi in 4 settori, un ciclo che attraversa una sola volta ogni settore è denominato fase:

- Settore 1: si determinano gli obiettivi per la fase corrente, ad esempio la costruzione di una componente software o la raccolta di tutti i requisiti;
- Settore 2: si valutano i rischi della fase corrente;
    - Un rischio è un evento sconosciuto che può impedire il completamento degli obiettivi. Il rischio ha principalmente due attributi: probabilità d'occorrenza e impatto sul progetto.
- Settore 3: si sviluppano gli obbiettivi imposti per la fase corrente;
- Settore 4: si effettua una revisione del prodotto e si pianifica la fase successiva;

Ogni fase produce un codice testato ed integrato nel sistema complessivo. Il processo spirale permette di far passare poco tempo prima del rilascio di una prima versione del prodotto e dà l'opportunità di interagire più volte con il cliente. Tali proprietà lo rendono un processo agile.



Sviluppo Agile

Alcuni processi di sviluppo, in particolare il processo di sviluppo XP, fanno parte della categoria dei processi di sviluppo AGILE. Intorno alla fine degli anni '90 e all'inizio degli anni 2000, è stato creato un manifesto per lo sviluppo AGILE che mirava a privilegiare l'esigenza degli sviluppatori rispetto alla rigidità di alcune cose stabilite, come la tempistica.

Definizione di Agilità: capacità di reagire ai cambiamenti del cliente e dall'ambiente circostante, in modo da non distruggere la robustezza del sistema software.

Lo sviluppo AGILE promuove l'auto-organizzazione, la collaborazione, la comunicazione tra membri del team e l'adattabilità del prodotto rispetto all'ordine e alla coerenza delle attività del progetto. Il manifesto agile consiste in 12 principi che possono essere riassunti in 4 asserzioni:

- Focalizzare gli individui e le interazioni invece che processi e strumenti;
- Focalizzare la disponibilità di un software funzionante invece di una documentazione completa;
- Focalizzare la collaborazione con il cliente invece che la negoziazione del contratto;
- Rispondere prontamente ai cambiamenti invece che rispettare la pianificazione;

Extreme Programming

Uno dei più famosi processi software agile è il processo Extreme Programming (XP), sviluppato negli anni '90. L'approccio è basato sullo sviluppo e la consegna di piccoli incrementi di funzionalità. Una breve panoramica delle caratteristiche principali:

- Solo 2 settimane per lo sviluppo degli incrementi;
- Piccoli gruppi di sviluppatori (da 2 a 12 persone);
- Costante miglioramento del codice;
- Poca documentazione;
- Enfasi sulla comunicazione diretta tra persone;
- Iterazioni corte e di durata costante;
- Coinvolgimento di sviluppatori, clienti e manager;
- Testabilità dei prodotti e prodotti testati sin dall'inizio;

È adatto a progetti con requisiti instabili, poiché XP è fortemente adattivo, e particolarmente rischiosi, ad esempio con deadline corte.

Le 12 pratiche XP

Pair Programming

Quando si deve produrre il codice di una componente del sistema, è necessario lavorare in coppia. Nella coppia si individuano due ruoli:

- Il Driver utilizza il mouse e la tastiera e pensa al miglior modo per implementare la specifica;
- Il Navigator revisiona l'approccio, pensa ai test e controlla che il design sia il più semplice possibile;

I ruoli non sono fossi, bensì avvengono degli scambi periodici. Le coppie di programmatori si scambiano ad ogni specifica da implementare. Tale pratica aiuta la disciplina, sparge la conoscenza del sistema e promuove lo scambio di conoscenze tra i programmatori.

Gioco di Pianificazione

Sviluppatori, manager e clienti si siedono ad un tavolo: gli utenti (clienti) scrivono le storie (i requisiti) su dei pezzettini di carta di dimensione limitata, per promuovere la scrittura di requisiti minimali, chiamati storycard. Gli sviluppatori stimano il tempo per lo sviluppo di ciascuna storia; se la storia è complessa, allora si torna la storycard al cliente per farla dividere in più storie. Gli utenti dividono, fondono e assegnano priorità alle storie.

Dopodiché, vengono riempite 3 settimane scegliendo delle storie e dandone una priorità di sviluppo: la prima settimana vengono sviluppate le storycard con priorità più alta, via via si scende di priorità; la pianificazione più accurata ricopre le prime due settimane di lavoro, le restanti storycard si traslano alla terza settimana. Le storycard vengono collocate in una Storyboard. La pianificazione è grossolana e si dà importanza al futuro prossimo.

Gli addetti al business prendono decisioni sulle date per le release, sul contesto e sulle priorità dei task. Per l'attuale release, gli sviluppatori dividono ciascuna storia in task e ne stimano la durata. Ciascuno si impegna a realizzare un task. Vengono svolti prima i task più rischiosi. Le storycard completate vengono posizionate nella parte bassa della storyboard; esse costituiranno la documentazione del progetto.

Cliente in Sede

Il cliente in sede è utile per scrivere dei test funzionali da applicare al codice, stabilire delle priorità, fornire informazioni extra e guidare le scelte dei programmatori. Un dubbio rispetto ad un aspetto funzionale del sistema software può essere subito disambiguato chiedendo al cliente in sede. Tuttavia, l'azienda potrebbe non essere d'accordo a lasciare una sua risorsa umana per un periodo di tempo all'interno della azienda software, quindi si permette al cliente sul sito di portare comunque avanti il suo lavoro.

Piccola release

L'obiettivo è quello di rilasciare una piccola release nel minor tempo possibile. Si stima che il tempo di rilascio sia di 2-3 settimane. Queste settimane dovrebbero garantire la produzione di software con un design semplice e sufficiente per la release corrente. I vantaggi delle piccole release sono tangibili:

- Si riduce il rischio;
- Si aumenta la possibilità di effettuare cambiamenti al variare dei requisiti;
- Si mantiene il cliente costantemente soddisfatto e se ne guadagna la fiducia;
- Feedback rapido sul lavoro svolto, quindi aumento della motivazione degli sviluppatori;

Metafora

È bene associare una metafora al sistema software in modo da far sentire il cliente a suo agio con essa. Molto spesso il cliente non è un esperto del settore, quindi può trovare difficile tenere il filo del discorso con un programmatore rispetto al software da produrre. Creare una metafora aiuta la collaborazione tra le due parti. Essa dovrà rappresentare l'architettura del software.

Possesso del codice collettivo

Chiunque può aggiungere qualunque codice su qualunque parte del sistema. Gli unit test ne prevengono la rottura e proteggono il codice già prodotto. Chiunque trova un problema, lo risolve. Ciascuno è responsabile per l'intero sistema.

Design Semplice

Il giusto design per il software si ha quando:

- Passa i test (utilizzare TDD);
- Non ha parti duplicate;
- È rilevante nel sistema;
- Ha il minor numero di classi e metodi

Non bisogna utilizzare anti-pattern ed è incoraggiato l'uso dei design pattern ove è possibile. È premiata la soluzione più semplice che funzioni.
Per documentare il design è possibile utilizzare delle CRC card (Class Responsibility Collaboration), esse permettono di ragionare meglio in termini di oggetti e contribuiscono a fornire una visione complessiva del sistema. Le card sono caratterizzate dal nome della classe, una piccola descrizione della sua responsabilità e un quadro delle collaborazioni con altre classi.

Testing

Il testing è fondamentale nel processo. Si testa tutto ciò che potenzialmente può andar male, per tutto il tempo. Si eseguono test più volte al giorno o non appena è stato prodotto del nuovo codice. I test fanno parte della documentazione totale del software poiché descrivono come esso dovrebbe funzionare. Si distinguono:

- I test funzionali, scritti dall'utente rispetto al suo punto di vista ed effettuati dagli utenti, dagli sviluppatori e dal team di testing. Sono una parte della specifica dei requisiti, quindi ne compongono una documentazione;
- Gli unit test, scritti dagli sviluppatori per testare il codice da produrre (TDD), ma anche dopo la codifica;

Integrazione continua

Nuovo codice viene prodotto e testato giornalmente. In media ogni coppia di sviluppatori completa un task al giorno e, una volta testato ed aver passato tutti gli unit test, il codice viene integrato alla parte di software esistente. Questa pratica viene detta integrazione continua.
Se un test fallisce, prima di integrare il codice va riparato per poter passare tutti i test, se, invece, si riconoscono scelte di design o algoritmiche inadatte, si butta il codice e si ricomincia da capo lo sviluppo del task.

40 ore a settimana

Se lo sviluppatore non riesce a completare il lavoro in meno di 40 ore a settimana, allora ha troppo lavoro da fare. Mantenere un totale di ore lavorative sotto le 40 settimanali permette di mantenere la lucidità del programmatore, evitando il burnout, e di produrre abbastanza codice. Mantenere la lucidità degli sviluppatori previene l'inserimento di potenziali errori difficili da trovare. Pianificazioni frequenti evitano a ciascuno di avere troppo lavoro a carico. Le ore lavorative extra sono sintomo di una cattiva pianificazione.

Refactoring
Effettuare refactoring significa migliorare la struttura del codice senza influenzarne il comportamento. Dovrebbe essere fatto in piccoli passi e testato tramite gli unit test. Questi ultimi sono necessari a non introdurre errori durante la modica del codice esistente.

Standard di codifica
Stabilire delle convenzioni sul codice da produrre ne aumenta la leggibilità e la capacità di comprensione. Costruzioni complicate per il design non sono permesse, si incoraggia di intraprendere soluzioni semplici.

Progettare per preservare
Alcune euristiche che guidano lo sviluppatore a progettare il software per preservarlo nel tempo:

- Creare un oggetto specificandone la classe esplicitamente orienta ad una particolare implementazione invece che a una interfaccia, è preferibile creare oggetti indirettamente (Factory method, prototype);
- Le dipendenze da operazioni specifiche vincolano le modalità di esecuzione di una richiesta, è preferibile evitare che le richieste siano indicate nel codice degli oggetti (Chain of responsibility, Command);
- Le dipendenze da piattaforme hardware e software rendono più difficile fare il porting del sistema software verso altre piattaforme, è consigliato progettare il sistema limitando le dipendenze dalla piattaforma (Factory, Bridge);
- Le dipendenze implementazioni rendono le classi client soggette a cambiamenti quando questa cambia, è preferibile nascondere le informazioni alle classi client, evitando così cambiamenti a cascata (Factory, Bridge, Memento, Proxy);
- Le dipendenze di oggetti da algoritmi che sono spesso estesi, ottimizzati e rimpiazzati durante lo sviluppo ed il riuso, possono richiedere cambiamenti agli oggetti al variare degli algoritmi. Si devono isolare gli algoritmi soggetti a cambiamenti (Builder, Iterator, Strategy/State, Template Method, Visitor);
- L'accoppiamento forte fra varie classi porta a sistemi monolitici, difficili da modificare e riusare. Si deve promuovere l'accoppiamento lasco per favorire la modificabilità, la riusabilità e l'estensione delle classi (Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator, Observer);
- L'estensione di funzionalità tramite ereditarietà è detto "riuso a scatola bianca" e richiede una profonda comprensione della superclasse e complessi ovveride, può causare l'anti-pattern Class Explosion. È quindi consigliabile affidarsi alla composizione, detta "riuso a scatola nera", anziché l'ereditarietà (Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy);
- Quando è impossibile modificare classi, per es. poiché il codice sorgente non è disponibile, o quando una modifica può richiedere cambiamenti a tante altre classi, possono essere utilizzati pattern come Adapter, Decorator e Visitor;

Metriche

Una metrica definisce un set di misure per un sistema software. Le metriche permettono di capire se il software ci soddisfa. Ogni metrica è calcolata in base ad un obiettivo:

- Monitorare il prodotto mentre si costruisce;
- Identificare i livelli per ciascuna metrica;
- Rimediare in caso i livelli non siano soddisfacenti;

Metriche tradizionali

Code coverage

La code coverage misura la copertura del codice, ovvero la percentuale di codice coperta (testata) dalla test suite fornita. Si può smettere di produrre test quando si raggiunge una percentuale di code coverage desiderata.
In generale, questa metrica è definita dal rapporto tra il numero di unità eseguite dai test e quelle rimanenti. Avere una copertura del 100% non significa non avere difetti.

Complessità ciclomatica [CC]

La complessità ciclomatica (CC) indica il numero di percorsi indipendenti tra una operazione e l'altra e valuta la complessità di un algoritmo. CC è il numero di test necessari per valutare esaurientemente l'algoritmo. Se l'algoritmo consiste in una sequenza lineare, basta un solo test. Se è presente una condizione, sono necessari due test: uno per condizione. Se si ha davanti un oggetto, la CC deve essere applicata ai singoli metodi.

Lines of codes [LOC]

La metrica lines of codes (LOC) misura la dimensione del progetto calcolando il numero di linee di codice. Spesso vengono esclusi i commenti e le linee vuote di codice (NCNB, non-comment non-blank). Una buona percentuale di commenti rispetto al numero di righe di codice è del 30%. Per capire la dimensione del codice è possibile basarsi sul numero di ';'.

Metriche Object Oriented

Le metriche object oriented si focalizzano sullo studio di progetti sviluppati con la OOP.

Weighted methods per class [WMC]

La WMC indica la somma delle complessità dei metodi per una classe. Il parametro con cui si misura la complessità è variabile, potrebbe essere:

- Le linee di codice (LOC);
- La complessità ciclomatica (CC);
- Il numero di metodi che non sono getter e setter;
- Potrebbe anche indicare semplicemente la quantità di metodi presenti in una classe.

Un alto valore di WMC indica grande lavoro di manutenzione, una grande specificità della classe e quindi poca possibilità di riuso.

Depth of inheritance tree [DIT]

La DIT indica il livello di profondità della classe rispetto ad un albero di ereditarietà. La radice dell'albero ha valore convenzionale 0. Per una classe, il valore di DIT si conta proseguendo verso l'alto. Le classi più in basso hanno un DIT maggiore, il che implica maggiore complessità (eredita il comportamento delle classi superiori).

Un alto DIT indica maggiore complessità dell'intero sistema, ma anche maggiore riuso.

Number of children of a class [NOC]

La NOC indica il numero di sottoclassi di una certa classe. Una classe con un alto NOC ha una importante influenza sul sistema.
Maggiore è il NOC, maggiore è il riuso. Se A è superclasse di B e C, NOC(A) = 2.

Coupling between object classes [CBO]

Il CBO indica il numero di classi con cui la classe interagisce. Se A ha una dipendenza verso B e C, allora CBO(A)=2. Non importa sapere quanti metodi utilizza la classe.
Maggiore è CBO maggiore è la dipendenza di una classe da altre, quindi minore possibilità di riuso, maggiore difficoltà a comprendere, modificare e correggere la classe. (Poco significativa per una classe Facade).

Response for a class [RFC]

La RFC indica il numero di metodi eseguiti al ricevimento di un messaggio, cioè [il numero di metodi di una classe + numero di metodi invocati da ciascuno di essi].
Un alto RFC indica una grande complessità e difficoltà di comprensione e di testing. Più è alto l'RFC e più è alta l'interazione della classe con le altre classi del sistema, ovvero indica un accoppiamento forte fra le classi.

Lack of cohesion of methods [LCOM]

Nella LCOM, per ogni attributo della classe, si calcola la percentuale di metodi che usano tale campo; si mediano le percentuali e si sottrae dal 100%.
Se LCOM = 0, allora la classe gode del 100% della coesione, e viceversa con LCOM = 1 si ha una classe scarsamente coesa, ovvero è accoppiata fortemente con le altre classi. Un valore ottimo di LCOM è 0.6 (60%).