Ingegneria del Software

1. Introduzione all'Ingegneria del Software

La locuzione **ingegneria del software** comprende i termini *ingegneria* e *software*: un ingegnere è colui che è in grado di costruire un prodotto di alta qualità usando delle componenti che vengono integrate tra loro, sfruttando un certo tempo e vincoli di budget.

I sistemi software sono creazioni complesse: eseguono numerose funzioni e sono costruiti perseguendo diversi obiettivi, spesso in conflitto tra loro. Sono costituiti da molte componenti, ognuna delle quali a sua volta complessa.
Il processo di sviluppo e il ciclo di vita di un software possono richiedere anche molti anni.
I sistemi complessi sono difficili da capire completamente da una singola persona: alcuni di essi durante le fasi di sviluppo sono così difficili da portare a termine da non venir mai conclusi: si parla così di *vaporware*.

Perchè è difficile creare software complessi:

1. **Problem Domain (o Application Domain)**: un software ha senso in un dominio applicativo, al variare di tale dominio possono insorgere problemi;
2. **Solution Domain**: oltre alla complessità del dominio, vi è la complessità della soluzione, e c'è un problema di matching tra le due.
3. **Development Process**: problema di processo di sviluppo; bisogna curare una serie di aspetti che vanno dalla costruzione del team, alla comunicazione al suo interno, alla condivisione dei piani di lavoro, assegnazione delle responsabilità, ma anche problemi più tecnici, come la gestione delle configurazioni (un software raramente esiste in un unica versione).
4. **Flexibility**: difficoltà del software di adattamento a cambiamenti interni o esterni.
5. **Discrete**: il software per sua natura è un sistema discreto, e i sistemi discreti sono per loro natura difficili da comprendere nelle loro conseguenze.

Maintenance del software

I sistemi software possono diventare obsoleti per 2 motivi: o perchè degradano o perchè varia il contesto in cui funziona il software.
I progetti di sviluppo software sono, quindi, in costante cambiamento: i requisiti sono complessi ed è necessario aggiornarli quando vengono scoperti degli errori, o quando gli sviluppatori acquisiscono una maggiore conoscenza sull'applicazione.

Se un progetto ha una durata di molti anni, è necessario avere un team di lavoro costantemente addestrato sui possibili cambiamenti (tecnologici, legislativi, organizzativi, etc.).
Il contesto intorno al quale avviene lo sviluppo di un software può cambiare anche molto velocemente e c'è il rischio che durante la modellazione del sistema, quest'ultimo non sia già più adatto al contesto in cui dovrà essere calato.

Che cos'è l'ingegneria del Software

Possiamo definire il processo di ingegneria del software come un'attività che comprende:

- **Modellazione**
- **Problem-solving**
- **Knowledge acquisition**
- **Rationale**

Modellazione

Gli ingegneri del software devono fare i conti con la complessità del sistema attraverso la costruzione di diversi modelli, in modo da concentrarsi, mano a mano, solo sui dettagli rilevanti ed ignorare il resto.

Un modello è una rappresentazione di un sistema che consente di rispondere a delle domande su di esso. Inoltre permettono di visualizzare e capire sistemi che non esistono più o che non esistono ancora (?).

Per la modellazione è necessario capire l'ambiente nel quale il sistema deve operare: l'ingegnere del software non deve diventare un completo conoscitore del contesto, ma soltanto imparare i concetti fondamentali e rilevanti del dominio applicativo per capire il sistema.

Inoltre bisogna capire le diverse soluzioni e relativi trade-off che possono essere messe in discussione per la costruzione del sistema. Molti sistemi sono troppo complessi per essere compresi da una singola persona e costosi da costruire: per risolvere questi problemi è necessario costruire dei modelli del **dominio della soluzione**.

I metodi object-oriented combinano il dominio dell'applicazione e il dominio della soluzione modellandoli in un'unica attività: il dominio applicativo è prima modellato come insieme di oggetti e relazioni; il dominio della soluzione è anch'esso modellato tramite oggetti, ma a partire da una trasformazione del modello precedente.

Problem-Solving

I modelli sono usati per cercare delle soluzioni accettabili, e questa ricerca è guidata dalla sperimentazione.
Gli ingegneri non hanno risorse infinite e sono vincolati da budget e scadenze: data l'assenza di una teoria fondamentale è spesso necessario fare affidamento su metodi empirici per valutare i benefici di diverse soluzioni alternative per la risoluzione dei problemi.

La ricerca della soluzione avviene, quindi, per prove ed errori, seguendo un processo empirico che richiede, nella sua forma più semplice, 5 step:

1. Formulare il problema
2. Analizzare il problema
3. Ricercare le soluzioni
4. Decidere la soluzione appropriata
5. Specificare la soluzione

Non si tratta di una attività algoritmica: richiede sperimentazione, il riuso di pattern di soluzioni, e l'evoluzione incrementale del sistema per ottenere una soluzione accettabile al cliente.
Lo sviluppo software object-oriented include tipicamente 6 attività di sviluppo:

- **Requirement elicitation**: definire i requisiti (funzionali e non funzionali) del sistema, ossia comprendere cosa deve fare il software e come lo deve fare, e renderlo formale.
- **Analysis**: queste prime due attività possono essere identificate nei primi 2 step del metodo ingegneristico.
- **System design** (o **high-level/architectural design**): si analizza il problema e lo si scompone in pezzi più piccoli per selezionare strategie generali per il design del sistema; stabilire quali sono le parti del sistema e come devono comunicare tra loro.
- **Object design** (o **low level design**): corrisponde agli step 3 e 4.
- **Implementazione**: corrisponde allo step 5.
- **Testing** (o **verification & validation**): attività aggiuntiva che permette di valutare l'appropriatezza del sistema rispetto ai modelli; prima vengono effettuate un'*analysis review* per comparare il

modello del dominio applicativo con la realtà del cliente, poi una *design review* per confrontare invece il modello della soluzione rispetto agli obiettivi di progetto. Infine vi è la fase di testing che permette di validare il sistema in relazione al modello della soluzione.

Knowledge acquisition

Durante la modellazione dell'applicazione e del dominio della soluzione, gli ingegneri del software collezionano dati e li organizzano in informazioni che vengono formalizzate in conoscenza.

Un errore comune è assumere che il processo di knowledge acquisition sia sequenziale: si tratta infatti di un processo **non lineare** perchè un singolo pezzo di una nuova informazione potrebbe invalidare tutta la conoscenza pregressa acquisita sul sistema.

Esistono diversi processi di sviluppo del software che cercano di risolvere i problemi che nascono con un modello di sviluppo **waterfall**, ossia a cascata (sequenziale).

Come queste attività sono collegate tra loro?
Queste attività non seguono un modello a cascata (waterfall), ossia non sono sequenziali.
Per molti anni l'ingegneria del software si è basata su un ciclo di vita del software strettamente sequenziale, in cui non si può passare allo step successivo se non si è completato lo step precedente.
Tali modelli *waterfall* storicamente sono risultati fallimentare per vari motivi:

- **Rigidità** del modello, che lo rende poco applicabile in casi reali.
- **Difficoltà nello spiegare i costi economici**: il modello a cascata è lineare, mentre i costi non rispecchiano tale andamento: l'80/90% dei costi può essere legato solo alla fase di manutenzione; pertanto, il modello lineare waterfall non è realistico.
- L'informatico e ricercatore IBM *Manny Lehman* affermava che un sistema che fa cose utili in un tempo reale deve cambiare nel tempo, altrimenti diventa obsoleto in quel contesto; il cambiamento non è quindi visto come un accidente, perchè il software deve cambiare in base ai cambiamenti del contesto. Man mano che cambiamo il sistema ne aumentiamo l'entropia, ne degradiamo la struttura e creiamo un fenomeno che altri autori come *Parnas* hanno etichettato come **software aging**, ossia invecchiamento del software; il cambiamento serve quindi anche per ripristinare la struttura funzionale del software.

Per superare il modello lineare sono quindi nate delle metodologie alternative che cercano di collegare le attività del processo ingegneristico del software in maniera diversa tra loro, in particolare modelli iterativi, ideati anche per massimizzare la possibilità futura di cambiamenti. Alcuni esempi sono:

- **Risk-based development**: si cerca di identificando le componenti ad alto rischio.
- **Issue-based development**: si tenta di rimuovere la linearità.

Metodologie agili: insieme di metodi di sviluppo del software fondati che premiano la risposta al cambiamento; si basano su un modello organizzativo, di comunicazione, di condivisione, continuamente pronto ad accettare cambiamenti del contesto.
Concetti fondamentali: formazione di team di sviluppo piccoli, poli-funzionali e auto-organizzati, sviluppo iterativo e incrementale, continuo e diretto coinvolgimento del cliente nel processo di sviluppo, etc.

Modello a Spirale di Boehm

Un modello a spirale iterativo dove lo sviluppo non finisce mai grazie al continuo processo di knowledge acquisition e discovery.
Alcuni cicli della spirale sono di prototipazione, altri di knowledge acquisition, etc.
Il numero di cicli della spirale è indefinito a priori.

I due paradigmi fondamentali: sistema di management rigido e un sistema molto più vicino alla realtà di sviluppo del software.

Supponiamo un processo di sviluppo a cascata e supponiamo che venga commesso un errore nella prima fase in cui si stila il documento dei requisiti: se l'errore non viene scoperto in questa fase, il costo per effettuare delle modifiche aumenta considerevolmente in funzione del tempo, quindi più tardi viene scoperto e maggiori saranno i costi.

I modelli a cascata hanno quindi intrinsicamente un'avversione al cambiamento perchè il cambiamento è pervasivo: qualunque modifica al sistema può avere conseguenze anche pesanti su tutto il resto.
Negli approcci moderni la curva dei costi non è più crescente col tempo: in un sistema orientato agli oggetti, o ben modularizzato, il costo del cambiamento è molto più basso.

Continuous integration: ogni volta che viene integrato un nuovo componente esso deve essere testato.

Rationale-driven

Quando si acquisisce conoscenza e si fanno decisioni sul sistema o sul dominio dell'applicazione, gli ingegneri devono catturare il contesto nel quale certe decisioni vengono prese e il razionale, ossia le motivazioni dietro queste decisioni.
Le informazioni sul razionale sono rappresentate come un insieme di modelli che permettono agli ingegneri del software di capire le implicazioni di un cambiamento proposto quando viene rivisitata una decisione.

Definizione di Progetto

Un **progetto** è un insieme di **attività**, organizzate in **task**, ognuno dei quali consuma risorse (tempo risorse umane e risorse macchina) per produrre artefatti (**work products)**.
Gli artefatti sono di varia natura e possono essere *sistemi*, *documenti* e *modelli*.
I progetti sviluppati con un approccio agile prevedono di stabilire come mettere insieme 3 cose: tempo, risorse e qualità.

Tutte le persone coinvolte in un progetto sono dette **partecipanti**, mentre un insieme di responsabilità costituiscono un **ruolo**.
Un ruolo è poi associato ad un insieme di task ed è assegnato a un partecipante al progetto.

Sistemi e modelli

Il termine **sistema** indica una collezione di parti interconnesse. La modellazione è fondamentale per fare i conti con la complessità di un sistema e per poterne ignorare i dettagli irrilevanti.

Artefatti - Work Product

Un work product è un artefatto prodotto durante il processo di sviluppo; distinguiamo:

- **Internal work product**: destinato a non essere consegnato al cliente (es. documenti di testing, status report, etc.)
- **Deliverable work product**: vengono consegnati al cliente, generalmente definiti a priori all'inizio del progetto.

Per noi il software è tutto l'insieme dei documenti necessari per la conoscenza: documenti, diagrammi UML, il codice sorgente, etc.
Possiamo considerare software anche tutto quel codice che non verrà destinato all'utente, come ad esempio il codice per la test automation.

Attività, task e risorse

Un'attività è un insieme di task, unità atomiche di lavoro.
Un manager assegna ad uno sviluppatore dei task e ne monitora progressi e completamento.
I task consumano risorse, producono work product e dipendono da work product prodotti da altri task.
Le risorse sono degli asset utilizzati per portare a termine un lavoro.

Dominio del problema: Requirement Stage

Requisiti funzionali e non funzionali

I requisiti specificano l'insieme delle funzionalità che un sistema deve avere e li distinguiamo in:

- **Requisiti funzionali**: cosa deve fare il sistema, ossia le funzioni che esso deve supportare
- **Requisiti non funzionali**: vincoli su come devono essere fatte operazioni tale operazioni; alcuni esempi di nonfunctional requirements riguardano specifiche piattaforme hardware, requisiti di sicurezza, gestione di failure e fault, gestione della backward compatibility con vecchi client, etc.

Il nostro approccio è basato su modelli: un modello è un'astrazione che dà una vista sul sistema o parte di esso e di alcune delle sue caratteristiche.
Tipicamente un modello si incarna in un grafo, che risponde a una notazione predefinita.
Una notazione è un insieme di regole grafiche o testuali per la rappresentazione di un modello.
La nostra notazione di riferimento è **UML** (**Unified Modelling Language**), molto comoda per sistemi object-oriented.

Un **metodo** è una tecnica ripetibile che specifica i passi per la risoluzione di uno specifico problema.
Una **metodologia** è un insieme di metodi per risolvere una classe di problemi e specifica come e quando usare ogni metodo.

Attività di sviluppo dell'Ingegneria del Software

Requirements Elicitation

Cliente e sviluppatori definiscono lo scopo del sistema. Il risultato di questa attività è una descrizione del sistema in termini di:

- **Attori**: entità esterne che interagiscono con il sistema; es. utenti, altre macchine, l'ambiente, etc.
- **Use case**: sequenze di eventi che descrivono tutte le possibili azioni tra un attore e il sistema per una data funzionalità.

Si produce quindi un modello funzionale, chiamato **use case model**, accompagnato da un insieme di requisiti non funzionali che il sistema dovrà rispettare.

Analysis

Durante questa fase gli sviluppatori mirano a produrre un modello del sistema che sia *corretto*, *completo*, *consistente* e *non ambiguo*.
Gli sviluppatori trasformano gli use case prodotti durante la fase di requirement analysis in un **object model** che descrive completamente il sistema.
Durante questa attività vengono scoperte ambiguità e inconsistenze dello use case model da risolvere con il cliente.
Il risu
Modelli UML utilizzati:

- **Dynamic model**: *state machine diagram* e *sequence diagram*.

- **Object model**: class diagram

System Design (High Level Design)

Gli sviluppatori definiscono i **design goal** del progetto e decompongono il sistema in sottosistemi più piccoli, realizzabili da piccoli team.
Vengono scelte strategie per la costruzione del sistema, tra cui piattaforme hardware/software, strategie di gestione dei dati persistenti, policy di controllo degli accessi, gestione delle condizioni dei boundary, etc.

Il risultato di questa attività è una descrizione di queste strategie, la decomposizione del sistema e un **deployment diagram** per il mapping hardware/software del sistema.

Sebbene anche nella fase di analysis si va a creare modelli del sistema, in questa fase vengono trattate entità più vicine al dominio della soluzione e, pertanto, sono modelli specifici per i team di sviluppo, non destinati al cliente.

Object Design (Low Level Design)

Si definiscono gli oggetti del dominio della soluzione come ponte tra il modello di analysis e system design.
Vengono descritti gli oggetti e le interfacce dei sottosistemi, si scelgono le componenti, si ristruttura l'object model in base ai design goal e si ottimizza l'object model rispetto alle prestazioni.
Il risultato di questa attività è un **object model** dettagliato e annotato con constraint e precise descrizioni per ogni elemento.

Implementazione

Gli sviluppatori traducono il modello del dominio della soluzione in codice sorgente. Questa fase include l'implementazione di metodi e attributi di ogni oggetto e l'integrazione tra essi per la costruzione di un singolo sistema.
Il risultato di questa fase è quindi un insieme di file sorgenti che possono essere compilati per costruire il sistema definitivo.

Testing

In questa fase si va a cercare le differenze tra il sistema effettivo e i modelli precedentemente costruiti con degli insieme di input d'esempio.
Diversi tipi di testing:

- Unit testing: confronto tra object design model e ogni oggetto/sottosistema.
- Integration testing: confronto del system design model con combinazioni e integrazioni di sottosistemi.
- System testing: confronto tra il requirement model e l'esecuzione di casi d'uso tipici o eccezionali sul sistema.

L'obiettivo di questa fase è quello di scoprire il maggior numero di fault possibili in modo da ripararli prima del delivery del sistema.
La fase di testing è spesso attuata in parallelo alle altre attività di sviluppo.

Attività di gestione dello sviluppo software

Le attività di gestione dello sviluppo riguardano la pianificazione del progetto, dello stato dei lavori, del coordinamento delle risorse in modo da consegnare un prodotto di alta qualità rispettando tempo e budget.

Le attività sono:

- **Comunicazione**: riguarda lo scambio di informazioni, modelli, documenti sul sistema e la comunicazione per le decisioni.
- **Rationale Management**: riguarda la giustificazione delle decisioni; gli sviluppatori devono catturare il razionale durante i meeting, le riunioni, e rappresentarlo sotto forma di modelli.
- **Software Configuration Management**: permette agli sviluppatori di tenere traccia dei cambiamenti sul sistema; il sistema è in evoluzione attraverso una serie di versioni, ed è possibile fare un roll back a versioni precedenti quando dei cambiamenti comportano dei problemi/errori; il configuration management permette anche di effettuare un check sui cambiamenti apportati dagli sviluppatori prima di essere effettivamente implementati nel sistema.
- **Project Management**: include le attività che permettono di avere un delivery del sistema nel rispetto di tempo e budget; include pianificazione e creazione del budget durante le negoziazioni con il cliente e l'organizzazione dei team di lavoro, monitoraggio dello stato del progetto e relativi interventi. Queste attività sono visibili solo agli sviluppatori.
- **Software Life Cycle**: è il modello generale di processo di sviluppo che viene scelto per il progetto.

Infine da citare l'attività di **Software Maintenance** che riguarda le attività di sviluppo post-delivery del sistema.

I 2 principi fondamentali di cui terremo conto: **information hiding** e **abstraction**.
Il concetto di abstraction è il cuore di ogni modello: è contemporaneamente un processo e una entità. L'astrazione è un processo perchè tolgo dettagli e ottengo una entità astratta, che serve per catturare le proprietà di una serie di entità concrete.

Chunking: mettere insieme elementi che interagiscono, es. classi di package diversi. Si collassano questi concetti. Meccanismo di presentazione.

Un altro meccanismo consiste nel definire via via i dettagli, descrivere le cose a livelli di astrazione diversi.

Articolo di Parnas - The Secret History of Information Hiding

Secondo Parnas il software dovrebbe essere progettato come un insieme di componenti modificabili e sostituibili in maniera indipendente → concetto di **modularità**.

Durante un corso in cui Parnas era insegnante, organizzò un progetto diviso in 5 moduli e lo assegnò agli studenti divisi in 5 gruppi: un gruppo per ogni modulo, ma ogni membro del gruppo doveva lavorare a una implementazione diversa del modulo → 4 implementazioni per ogni modulo.
Tali implementazioni dovevano essere intercambiabili, in modo da poter effettuare 25 delle 1024 possibili combinazioni di test di esecuzione del sistema.
Nella implementazione gli studenti avevano a disposizione una interfaccia astratta del modulo come descrizione delle specifiche e tutte le implementazioni dovevano soddisfare tale descrizione. I dettagli di implementazione dei moduli dovevano restare segreti.

La modularizzazione del sistema limita la quantità di informazione che deve essere condivisa da chi implementa i moduli.
Per avere un buon design le interfacce devono contenete solo informazioni "solide" o durature, mentre tutte le informazioni arbitrarie o modificabili devono restare nascoste.
Le interfacce sono un insieme di assunzioni che un utente può fare su un modulo.

Secondo Parnas non è corretto utilizzare i Flow-Chart per la progettazione di moduli perchè necessitano di un grande scambio di informazioni mediante strutture dati complesse e generano pessime interfacce. La decomposizione mediante flow chart vìola quindi i principi di information hiding.

Per sistemi complessi Parnas propone di utilizzare una struttura ad albero gerarchica per i moduli: ogni modulo ai livelli più alti è a sua volta diviso in moduli più piccoli.

Il concetto di **information hiding** è la base di tre concetti più recenti:

- **ADT - Tipi di dato astratto**: permettono di definire un tipo di dato di cui è possibile creare diverse implementazioni nascoste; gli ADT permettono quindi di usare un tipo di variabiles enza dover conoscere l'implementazione effettiva.
- **Linguaggi di programmazione Object Oriented**
- **Design orientato alle componenti**

Articolo - Modello a Spirale di Boehm

Boehm capì che il processo di sviluppo di un sistema software complesso non può essere lineare.
Il modello a cascata, pur con estensioni per permettere lo sviluppo incrementale, parallelo, l'evoluzione del sistema, l'analisi dei rischi, etc. ha storicamente incontrato numerose difficoltà di attuazione nel processo di sviluppo di sistemi complessi.

Boehm propose un modello di sviluppo **risk-driven** in cui il sistema evolve continuamente e iterativamente su 4 aree di una spirale. I passi di un ciclo della spirale sono:

- Specificazione formale di obiettivi e constraints (i vincoli) del prodotto
- Valutazione delle possibili alternative, identificazione e risoluzione di rischi (**risk analysis**) mediante prototipazione, simulazioni, modelli analitici, etc.
- Costruzione del prodotto in base agli obiettivi e alle decisioni prese e test
- Pianificazione della successiva iterazione

In sintesi, si tratta delle stesse fasi del modello waterfall ma in tempi più ristretti ma ripetute in maniera ciclica e incrementale.

L'articolo dimostra che tale modello aumenta considerevolmente la produttività e diminuisce i costi e tempi di risoluzione di problemi ed errori durante il processo di sviluppo, in particolare nella realizzazione di sistemi molto complessi e con elementi ad alto rischio.

Risk management plan: identificazione di una top 10 (o numero diverso) di rischi del progetto e sviluppo di una strategia per la loro risoluzione. Questa tecnica è applicabile anche nel caso non si riesca ad applicare totalmente l'approccio a spirale proposto da Boehm.

Conclusioni

1. Il modello a spirale, grazie alla sua natura risk-driven è particolarmente adatto a progetti di sistemi complessi e molto grandi
2. Possibilità di implementazione parziale utilizzando il Risk Management Plan, compatibile con molti degli altri modelli di processo di sviluppo.

Articolo: No Silver Bullet, Fred Brooks - Fattori di complessità dei progetti di sviluppo software

Brooks fa distinzione tra due tipi di complessità:

- **Complessità accidentale**: relativa ai problemi che gli ingegneri creano e possono risolvere.
- **Complessità essenziale**: causata dagli effettivi problemi che devono essere risolti e che non può essere rimossa.

La complessità accidentale ad oggi è ridotta moltissimo, e oggi gli sviluppatori passano la maggior parte del tempo a cercare di risolvere problemi di complessità essenziale.
Ridurre la complessità accidentale però non dà gli stessi miglioramenti di una pari riduzione della complessità essenziale.

Una delle tecnolgie che ha portato miglioramenti significativi della complessità accidentale è stata l'introduzione di linguaggi di programmazione ad alto livello, in particolare con l'introduzione del paradigma object oriented.

Brooks sostiene che il software debba subire una crescita in modo organico attraverso uno sviluppo incrementale e suggerisce di ideare e implementare sia il programma principale che sottoprogrammi fin dall'inizio.

...

Principio di decomposizione

L'idea del paradigma OO permette di fondere la vista behavior con la semantica dei dati, e relative relazioni, implementando i principi di astrazione ed information hiding.

Un **Abstract Data Type (ADT)** è un tipo che definisce un concetto in maniera astratta; permette di accoppiare dati e behavior in maniera assiomatica.

2. Modellazione con UML

Il processo di ingegneria del software è un processo in cui si esplicitano via via sempre più dettagli descrivendoli con uno strumento di modellazione: l'ultimo strumento che verrà utilizzato è il codice.

UML - Unified Modelling Language

UML è una notazione standard che fornisce un ricchissimo vocabolario di viste (oltre 20, più la possibilità di personalizzazione (*stereotypes*) per descrivere un modello di un sistema software complesso a vari livelli di astrazione.
UML si poggia su un meta-modello, ed è possibile arricchire il modello sottostante con gli stereotipi.

Ci concentriamo su 3 differenti modelli per il sistema:

- **Functional Model**: descrive le funzionalità del sistema, ossia quali sono i behavior (nel dominio applicativo e nel dominio della soluzione) che vogliamo realizzare → **use case diagram**
- **Object Model**: descrive la struttura del sistema in termini di oggetti, attributi, associazioni e operazioni; UML utilizzato: **class diagram**. Distinguiamo:
  - **Analysis object model**: descrive i concetti rilevanti del sistema
  - **System design object model**: include le descrizioni delle interface dei sottosistemi
- **Dynamic Model**: descrive il comportamento interno del sistema sulla base delle funzionalità previste nel *functional model*, mostrando le interazioni tra gli oggetti del sistema; alcuni esempi sono:
  - **State machine diagram**: descrive il comportamento in termini di transizioni di stato
  - **Interaction diagram**: descrive il comportamento come sequenza di messaggi scambiati tra gli oggetti
  - **Activity diagram**: descrive il comportamento in termini di flusso di controllo e di dati.

Use-Case Diagram

Sono utilizzati durante la fase di *requirement elicitation* e *analysis* per rappresentare le funzionalità che deve avere il sistema, ossia i behaviour nel proprio dominio applicativo.

Uno **use case** è una funzionalità fornita dal sistema che produce un risultato visibile ad un attore che interagisce con esso.
Un **attore** è una qualsiasi entità esterna che interagisce col sistema.
Per realizzare il modello funzionale è quindi fondamentale identificare attori e casi d'uso.

Class Diagram

Sono utilizzati per descrivere la struttura del sistema in termini di Classi.
Una **classe** è un'astrazione che specifica la struttura e il behavior comune di un insieme di oggetti. Gli oggetti sono quindi istanze delle classi e sono creati, modificati e distrutti durante l'esecuzione del sistema.
I class diagram descrivono il sistema in termini di oggetti, classi, attributi, operazioni e relazioni.

Ogni relazione ha una molteplicità, che descrive quanti link esistono tra gli oggetti di due classi in relazione tra loro.
Ogni classe è caratterizzata da informazioni, i cui valori determineranno lo **stato** degli oggetti di quella classe, e dei comportamenti, che sono il modo attraverso il quale è possibile interagire con quegli oggetti ed eventualmente cambiarne lo stato.

Interaction Diagrams → Sequence Diagram

Sono usati per formalizzare il comportamento dinamico del sistema e visualizzare la comunicazione tra gli oggetti.
Analizziamo un particolare tipo di interaction diagram → **Sequence diagram**.

In un sequence diagram andiamo a rappresentare la comunicazione tra gli oggetti coinvolti all'interno di uno use case.

Viene rappresentato uno scenario, non è un modello generale in cui si descrivono tutti i possibili comportamenti del sistema: servirebbero tanti diagrammi quanti sono utili a descrivere i possibili scenari di utilizzo del sistema e catturare gli andamenti tipici.

State-Machine Diagrams

Descrivono il comportamento dinamico di un oggetto in termini di stati e transizioni tra stati.
Uno **stato** è un particolare insieme di valori di un oggetto. Dato uno stato, una **transizione** è il passaggio dell'oggetto ad uno stato futuro al verificarsi di una data condizione.
A differenza del sequence diagram si concentra sulle transizioni tra stati come risultato di eventi esterni per un singolo oggetto individuale.

Activity Diagram

Descrive il comportamento del sistema in termini di activities, le quali rappresentano l'esecuzione di un insieme di operazioni.
L'esecuzione di una attività può essere attivata dal completamento di altre attività, dalla disponibilità di oggetti o da eventi esterni.
Sono simili a dei flow chart e sono molto utili per rappresentare control e data flow del sistema.

Concetti fondamentali: Sistema - Modello - Viste

Un **sistema** è un insieme organizzato di parti comunicanti; le parti di un sistema possono essere considerate come dei sistemi più semplici chiamati **sottosistemi**.
La decomposizione del sistema in sottosistemi può essere quindi applicata ricorsicamente anche ai sottosistemi stessi.
Gli oggetti rappresentano le entità finali di questa decomposizione.

I sistemi complessi necessitano di più di un modello per essere rappresentati. L'insieme di tutti i modelli costruiti durante lo sviluppo prende il nome di **system model**.
Una **vista** è una parte del modello che si focalizza su un aspetto particolare del sistema (funzionalità, struttura in parti, colloquio fra le parti, dinamica interna di ogni parte, etc.).

Una **notazione** è un insieme di regole grafiche o testuali per rappresentare delle viste.
Ad esempio un class diagram è una vista *grafica* di un object model.
UML è soltanto una delle possibili notazioni utilizzate nel campo dell'ingegneria del software.

Tipi di dato, ADT e istanze

Un **tipo di dato** è una astrazione nel contesto di un linguaggio di programmazione. Ogni tipo di dato ha un nome univoco che lo distingue dagli altri e un insieme di valori che i dati di quel tipo (le **istanze**) possono assumere.

Un **tipo di dato astratto** è un tipo di dato definito indipendentemente dalle specifiche di implementazione.
Un sistema può fornire diverse implementazioni di un insieme di ADT, ognuna delle quali ottimizza dei criteri differenti (consumo di memoria, tempo, etc.).
Uno sviluppatore che deve utilizzare un ADT ha bisogno di conoscere solo la sua semantica e non la rappresentazione interna, ossia l'implementazione.

Classi, classi astratte e oggetti

Una classe è un'astrazione dei linguaggi OO che incapsula sia la struttura che i behavior; a differenze degli ADT le classi possono essere definite a partire da altre classi attraverso l'**ereditarietà**: la classe generalizzata è chiamata **superclasse**, quella specializzata invece **sottoclasse**.

Una superclasse può essere una **classe astratta** se è definita soltanto per modellare attributi ed operazioni condivise, ma non deve essere istanziata. Le classi astratte spesso servono per generalizzare concetti del dominio applicativo.
Es. di classe astratta in Java: *Collection* che fornisce una generalizzazione per il concetto di collezione di oggetti di qualsiasi classe.
A volte è utile creare classi astratte per ridurre la complessità in un modello o per il riutilizzo di codice.

Un oggetto è un'istanza di una classe, che possiede uno stato definito dai valori dei suoi attributi.
L'ereditarietà permette di stabilire una **gerarchia di tipo**: un oggetto della sottoclasse **è** ("*is a*") anche un oggetto della sua superclasse, ma non viceversa.
Gli oggetti di una classe possono avere una o più referenze ma non un nome: solo la classe ha un nome.

Classi evento, eventi e messaggi

Le **classi evento** sono astrazioni che rappresentano eventi per i quali il sistema ha una risposta comune.
Un **evento** è un'istanza di una classe evento, ossia un'occorrenza rilevante per il sistema.
Un **messaggio** è il meccanismo tramite il quale è possibile inviare agli oggetti una richiesta di esecuzione di un'operazione.

Modellazione Object-Oriented

Il **dominio applicativo** rappresenta tutti gli aspetti del problema che si vuole risolvere ed è importante per gli sviluppatori comprenderlo per la creazione del sistema.
Il **dominio della soluzione** è lo spazio di modellazione di tutti i possibili sistemi: modellare il dominio della soluzione significa rappresentare le attività di system e object design del processo di sviluppo.

Il modello del dominio della soluzione è molto più volatile del modello del dominio applicativo, perchè molto più dettagliato e a causa del cambiamento delle tecnologie e delle conoscenze degli sviluppatori su di esse.

L'**object-oriented analysis** si preoccupa di modellare il dominio applicativo, l'**object-oriented design** invece del dominio della soluzione.
Entrambe le attività di modellazione sfruttano i concetti di classe e oggetto, ma nella seconda andiamo anche a rappresentare le informazioni associate agli oggetti del dominio e nuovi concetti, propri del dominio della soluzione.

Un modello è una semplificazione della realtà che ignora i dettagli non rilevanti.
La **falsificazione** è un processo che permette di dimostrare se i dettagli rilevanti sono stati correttamente rappresentati, o non sono stati rappresentati del tutto, ossia se il modello non corrisponde alla realtà che intende rappresentare.

La **prototipazione** è una tecnica per lo sviluppo di un sistema che permette di costruire un *prototipo* che simula l'interfaccia utente del sistema. Il prorotipo è quindi un modo per effettuare una *valutazione/falsificazione* del sistema.
Sebbene sia possibile stabilire quando un modello non rappresenta correttamente la realtà, non è possibile il contrario.

Principali diagrammi UML

Use case diagram

Rappresentano le funzionalità del sistema dal punto di vista dell'utente; definiscono i boundaries (limiti) del sistema.

Gli *attori* sono le entità esterne che interagiscono col sistema; hanno nomi unici e delle descrizioni; possono essere utenti, altri sistemi, etc.
Gli *use case* descrivono un comportamento del sistema dal punto di vista di un attore, i cosiddetti *external behavior*; descrivono una funzione fornita dal sistema come insieme di eventi che genera un risultato visibile agli attori.

Oltre alla rappresentazione grafica dello use case diagram è possibile creare delle descrizioni testuali in linguaggio naturale per ognuno degli use case usando un template in cui si descrive:

- Il **nome** univoco dello use case
- Gli **attori** partecipanti
- Il **flusso di eventi**, ossia la sequenza di interazioni tra gli attori e il sistema; per maggiore chiarezza i casi comuni ed eccezionali sono descritti in use case diversi;
- Le **condizioni di entrata**, ossia le condizioni che devono essere soddisfatte per dare inizio allo use case
- Le **condizioni di uscita**, condizioni da soddisfare per il completamento dello use case
- i **requisiti di qualità**, ossia vincoli sulle prestazioni del sistema, sull'implementazione, sulla piattaforma hardware, etc.

Gli use case diagram possono includere 4 tipi di relazioni:

- **Comunicazione**: per definire lo scambio di informazioni tra attori e use case.
- **Inclusione**: due use case sono correlati da una relazioni di inclusione se uno include l'altro nel flusso degli eventi; se lo use case incluso può essere incluso in un punto qualsiasi del flusso degli eventi si specifica la relazione nei quality requirements, se è invocato durante un evento allora lo si indica direttamente nel flusso degli eventi.

- **Estensione**: un altro modo per ridurre la complessità dello use case model; uno use case può estendere un altro aggiungendo eventi; un'istanza di uno use case esteso può includere sotto certe condizioni il comportamento di un altro use case che lo estende (es. ConnectionDown).

In generale: gli use case d'eccezione sono modellati utilizzando delle relazioni di estensione, gli use case che descrivono dei behavior comuni invece sono modellati con relazioni di inclusione.

- **Ereditarietà**: quando si vuole definire uno use case che specializza un altro aggiungendo maggiori dettagli (es. Autenticazione, AutenticazioneConPassword, AutenticazioneConCarta); gli use case specializzati ereditano l'attore iniziatore dello use case e le entry ed exit condition dallo use case generale.

Scenari

Uno **scenario** è una istanza di uno use case che descrive un insieme concredo di azioni. Gli scenari sono utili come esempi per illustrare di casi comuni. Il template per gli scenari è simile a quello degli use case e comprende:

- Nome dello scenario (sottolineato per indicare che è un'istanza)
- Istanze degli attori partecipanti (sottolineate)
- Flusso degli eventi

Non sono presenti entry ed exit condition perchè sono astrazioni che descrivono un range di condizioni di uno use case e non di una specifica situazione, quale è lo scenario.

Class diagram

Rappresentano la struttura statica del sistema in termini di oggetti e loro attributi, operazioni e relazioni. Una *classe* è un insieme di oggetti che condivide attributi e comportamenti. Un *oggetto* è un'istanza di una classe che si distingue dalle altre per il suo stato.

Classi e oggetti sono rappresentate attraverso dei box composti da tre compartimenti: uno per il **nome** della classe/oggetto, uno per gli **attributi** e uno per le **operazioni**. Per convenzione i nomi delle classi iniziano con una lettera maiuscola, mentre i nomi degli oggetti sono sottolineati per indicare che sono delle istanze.

Un **link** è una connessione tra due oggetti. Una **associazione** è una relazione tra classi e rappresenta un gruppo di link.
Le associazioni possono essere simmetriche (bidirezionali) o asimmetriche (unidirezionali).

Le **association class** ci danno informazioni su delle associazioni; sono simili a delle classi e possono essere trasformate in classi effettive, associate alle classi di partenza dell'associaizone.

Ogni classe collegata ad un'associazione può essere etichettata con un **ruolo** per chiarificare e distinguere meglio le classi partecipanti.

Le associazioni sono dotate di una **molteplicità** che indica il numero minimo e massimo di istanze di ogni classe che possono partecipare alla associazione.
Distinguiamo associazioni:

- **uno a uno**
- **uno a molti**
- **molti a molti**

La **qualificazione** è una tecnica utile per ridurre la molteplicità di un'associazione usando degli attributi come chiave → **qualificatori**. Il qualificatore è una particolare proprietà che mette in relazione gli oggetti delle classi associate.

Tipi particolari di associazione

- **Aggregazione**: implica che una classe è un contenitore di altre classi associate ad essa; si denota con un simbolo di diamante che termina nella classe container.
- **Composizione**: si distingue dall'aggregazione perchè indica un particolare tipo di aggregazione; gli oggetti delle classi composite hanno lo stesso periodo di vita degli oggetti delle loro componenti.
- **Ereditarietà**: relazione tra una classe generale e una o più classi specializzate; descrive quindi gli attributi e le operazioni in comune ad un insieme di classi; si denota con un triangolo che termina nella superclasse.

Il comportamento di un oggetto è specificato da delle operazioni; un oggetto richiede l'esecuzione di un'operazione da un altro oggetto inviando un messaggio; Il messaggio viene infine matchato con un **metodo** definito dalla classe dell'oggetto ricevente il messaggio o da una delle eventuali superclassi.
La distinzione tra operzioni e metodi ci permette di distinguere i concetti di *behavior* e *implementazione*.

Interaction diagram

Rappresentano i behavior del sistema in termini di interazioni tra un insieme di oggetti; utili per identificare oggetti nel dominio applicativo e nel dominio della soluzione.
Un oggetto interagisce con altri oggetti inviando messaggi, che comportano l'esecuzione di un metodo e che possono a loro volta inviare messaggi ad altri oggetti.

Tramite un messaggio è possibile passare degli **argomenti**, ossia i parametri del metodo dell'oggetto ricevente il messaggio.

I **sequence diagram** sono dei particolari interaction diagram che rappresentano le interazioni tra oggetti in orizzontale, in relazione al tempo rappresentato in verticale.
L'attore iniziatore dello use case è rappresentato all'estrema sinistra del diagramma; altri eventuali attori sono rappresentati a destra.

I sequence diagram sono utili per descrivere sequenze astratte o concrete di possibili interazioni tra gli oggetti.
Un **communication diagram** è simile ad un sequence, ma rende il tutto più compatto, di conseguenza anche più difficile da seguire.

State machine diagram

Descrivono la sequenza di transizioni di stato di un oggetto in seguito come risposta ad eventi esterni.
Uno **stato** è una condizione soddisfata dagli attributi di un oggetto; una **transizione** rappresenta il cambiamento di uno stato, attivato da eventi, condizioni o dal tempo.
Una **transizione interna** è una particolare transizione che non porta cambiamento di stato dell'oggetto.
Una **activity** è un insieme coordinato di azioni: uno stato può essere associato ad una activity che viene eseguita finchè l'oggetto risiede in quello stato.

Per ridurre la complessità è possibile utilizzare delle **macchine a stati innestate** al posto delle transizioni interne. Ogni stato infatti può essere modellato come macchina a stati innestata.

In sintesi: gli state machine diagram sono utili per rappresentare i comportamenti significativi di un oggetto/sottosistema, per identificare attributi di un oggetto e rifinire la descrizione del comportamento.

Activity diagram

Diagrammi di flusso usati per rappresentare il flusso di dati e di controllo attraverso il sistema.
Una **activity** è un'azione, oppure un grafo di altre *subactivity*, ognuna con un proprio flusso di oggetti associato.
Negli activity diagram distinguiamo:

- **Nodi di controllo**: per rappresentare decisioni, concorrenza, sincronizzazione
- **Decisioni**: diramazioni del flusso di controllo che denotano alternative basate su certe condizioni di uno stato di un oggetto o un insieme di oggetti.
- **Nodi fork e join**: rappresentano la concorrenza; i primi denotano la possibilità di splitting del flusso di controllo in più *thread*, i secondi la sincronizzazione e il merge del flusso di controllo di più thread in un unico.

Le activity possono essere raggruppate in **swimlane** (o activity partition) per denotare l'oggetto o il sottosistema che implementa le azioni.

Organizzazione e estensione dei diagrammi

Un **package** è un raggruppamento di elementi di modelli, come use case e classi, per migliorare la comprensione del sistema.
Una **nota** è un commento in cui è possibile registrare problemi di comprensione, chiarificazioni, reminder, etc.

Uno **stereotipo** è un meccanismo di estensione che permette agli sviluppatori di classificare gli elementi di modello. Attaccare uno stereotipo a un elemento di modello equivale a creare una nuova classe nel meta-modello UML e costruire quindi nuovi building block. (es. di stereotipi sono entity, boundary, control).
Un **constraint** (vincolo) è una regola attaccata ad un elemento di modello che restringe la sua semantica e permette di rappresentare fenomeni altrimente non esprimibili con UML.

Chunking e dettagliamento

Come abbiamo visto, nel caso dei package attuiamo il cosiddetto **chunking**, ossia mettiamo insieme elementi di modellazione.
Il meccanismo opposto, come visto ad esempio nelle state machine, consiste invece nel dettagliamento a più livelli di astrazione, definendo via via sempre più dettagli su un certo elemento.
Entrambe queste tecniche si possono applicare a ciascuna delle viste dei modelli.

4. Requirement Elicitation

L'attività di Requirement Elicitation riguarda la descrizione dello scopo del sistema: identificare l'area del problema e definire un sistema che lo risolva.
Tale definizione prende il nome di **requirement specification** (specifica dei requisiti) e funge da contratto tra cliente e sviluppatori. Essa si compone di requisiti funzionali e non funzionali.

Successivamente la requirement specification verrà formalizzata durante la fase di analysis per produrre un **analysis model**.
La specifica dei requisiti supporta la comunicazione tra cliente e sviluppatori, l'analysis model invece la comunicazione tra sviluppatori.
Entrambi sono modelli che tentano di rappresentare accuratamente gli aspetti esterni del sistema, dal punto di vista dell'utente.

La fase di requirement elicitation include le seguenti attività:

- *Identificazione degli attori*
- *Identificazione di scenari nei quali gli attori si collocano*

- *Identificazione degli use-case*
- *Rifinire gli use-case*
- *Elicitare le relazioni esistenti tra gli use-case*
- *Identificazione dei non-functional requirements*

Functional Requirements

Si parte dall'elicitazione di uno statement del problema da risolvere che descrive in linea generale il sistema che si vuole realizzare, a partire da interviste e incontri con il cliente.
L'obiettivo è descrivere le interazioni tra il sistema e l'ambiente, ossia utenti e altri sistemi esterni con cui il sistema dovrà interagire, senza alcun dettaglio di implementazione.

Nonfunctional requirements

Descrivono aspetti del sistema che non sono direttamente correlati ai comportamenti funzionali del sistema.
In letteratura esistono numerosi tentativi di standardizzazione della definizione dei requisiti non funzionali.

Tipicamente questi requisiti si realizzano sotto forma di alberi.

Approccio goal-question metrics: una sorta di linea guida per definire i requisiti di qualità; definire un proprio albero dei requisiti di qualità.

Modello FURPS+ - modello gerarchico di attributi di qualità (**quality requirements**):

- **Usability**: semplicità di utilizzo del sistema per un utente (es. user interface, livello della documentazione, etc.)
- **Reliability**: abilità del sistema o di un componente di eseguire correttamente le sue funzioni per uno specifico periodo di tempo; oggi questa categoria è spesso sostituita dalla **dependability**, che include:
    - reliability
    - **robustezza** (il grado con cui un sistema o componente funziona correttamente in condizioni difficoltose o con input non validi)
    - **safety** (misura dell'assenza di conseguenze catastrofiche per l'ambiente)
- **Performance**: che comprende attributi quantitativi come:
    - **tempo di risposta** (tempo di reazione agli input)
    - **throughput** (quanto lavoro il sistema riesce ad eseguire in uno specifico intervallo di tempo)
    - **availability** (il grado con cui un sistema o componente è operativo e accessibile quando richiesto per l'uso)
    - **accuracy**
- **Supportability**: facilità di effettuare cambiamenti al sistema dopo il deployment; comprende
    - **adattabilità**: possibilità di cambiare il sistema in seguito all'aggiunta di nuovi concetti del dominio applicativo.
    - **maintainability**: possibilità di cambiare il sistema con nuove tecnologie o di fix dei difetti; il modello ISO 9126 affianca alla maintainability la **portability**, ossia la facilità con cui il sistema o un componente può essere trasferito da un ambiente hardware/software all'altro.

Oltre ai quality requirements il modello FURPS+ fornisce categorie aggiuntive di nonfunctional requirement, chiamati **constraints** o **pseudo requirements**, come:

- **Implementation requirements**: vincoli sull'implementazione del sistema (es. uso di specifici tool, linguaggi, piattaforme hardware)
- **Interface requirements**: vincoli imposti da sistemi esterni (es. sistemi legacy)

- **Operations requirements**: vincoli sull'amministrazione e gestione del sistema nei setting operazionali
- **Packaging requirements**: vincoli sul delivery del sistema (es. installazione)
- **Legal requirements**: vincoli su licenze, regolamenti, certificazioni, etc.

Validazione delle specifiche - Proprietà - Quality Check Report

I requisiti del sistema sono in continua validazione con il cliente e gli utenti. La **validazione dei requisiti** è un passo critico del processo di sviluppo e richiede di effettuare un check su alcune caratteristiche:

- **Completezza**: se la specifica del sistema descrive tutti i possibili scenari, inclusi comportamenti eccezionali (cioè se tutti gli aspetti del sistema sono rappresentati nel requirements model)
- **Chiarezza**: se la specifica è non ambigua
- **Consistenza**: se la specifica non contraddice sé stessa
- **Correttezza**: se rappresenta accuratamente il sistema che il cliente desidera e che gli sviluppatori intendono costruire (cioè se tutto ciò che è presente nel requirement model rappresenta accuratamente un aspetto del sistema che soddisfi sia il cliente che gli sviluppatori)

Altre proprietà desiderate di una specifica:

- **Realismo**: se il sistema può essere realizzato nel rispetto dei constraints
- **Verifiability** (o falsificabilità): se una volta che il sistema viene costruito, è possibile dimostrare che il sistema rispetti le specifiche (es. testing per il codice)
- **Traceability**: se ogni requisito può essere tracciato attraverso lo sviluppo del software con delle corrispondenti funzioni del sistema, e se viceversa ogni funzione del sistema può essere associata al suo corrispondente insieme di requisiti.

Attività di Requirement Elicitation

Le seguenti attività sono necessarie per passare dallo statement del problema a una specifica dei requisiti formale in termini di attori, scenari e use case.

Identificazione degli attori

Gli attori sono delle astrazioni di ruoli di entità esterne che non necessariamente sono mappati direttamente con delle persone; una stessa persona può rappresentare diversi attori.
Durante l'inizio di questa fase può essere difficile distinguere attori da oggetti: ad esempio un database può essere sia un attore che parte del sistema.
Una volta che vengono definiti i boundary del sistema non vi è più ambiguità nel distinguere attori da oggetti e sottosistemi: gli attori sono al di fuori del system boundary, oggetti e sottosistemi sono interni.
Esempi di domande per identificare attori:

- Quale gruppo di utenti il sistema deve supportare per svolgere il suo lavoro?
- Quale gruppo di utenti esecue le funzioni principali del sistema?
- Quali gruppi di utenti eseguono funzioni secondarie (come manutenzione e amministrazione)?
- Con quali hardware o software esterni il sistema dovrà interagire?

Dopo l'identificazione degli attori bisognerà costruire scenari e use case, secondo 2 possibili approcci:

- **Top down**: dallo use-case agli scenari
- **Bottom up**: dagli scenari allo use-case; dal dettaglio all'astrazione.

Identificazione degli scenari

Uno scenario è una descrizione informale di una singola feature del sistema dal punto di vista di un singolo attore.
Gli scenari non sostituiscono gli use case perchè ne costituiscono una specifica istanza, ossia un evento concreto; tuttavia posono essere utili come strumento di comprensione per utenti e clienti.
Inoltre, quando il sistema è in un dominio di conoscenza lontano, è possibile partire da un insieme di scenari per derivare uno use case.
Tipi di scenario:

- **As-is scenario**: descrive una situazione corrente;
- **Visionary scenario**: descrive un sistema futuro, usati sia come mezzo di comunicazione per elicitare requisiti dagli utenti, sia per modellare idee per il sistema futuro.
- **Evaluation scenario**: descrive i task dell'utente contro i quali il sistema verrà valutato per migliorare le funzionalità.
- **Training scenario**: tutorial per introdurre nuovi utenti al sistema, costituiti da istruzioni passo-passo per l'esecuzione di task comuni.

Domande per identificare scenari:

- Quali task gli attori vogliono che il sistema esegua?
- A quali informazioni l'attore avrà accesso? Chi crea quei dati? Chi può modifcarli o rimuoverli?
- Di quali cambiamenti esterni l'attore dovrà informare il sistema? Quanto spesso? Quando?
- Di quali eventi il sistema l'attore dovrà informare il sistema? Con che latenza?

Identificazione degli use case

Uno use case specifica tutti i possibili scenari per un dato pezzo di funzionalit del sistema; è iniziato da un attore e può intergire con altri attori.
Il nome di uno use case dovrebbe essere un predicato verbale che denota ciò che l'attore che inizia lo use case intende eseguire.

Refinement degli use case

Una volta identificati gli use case è possibile aggiungere dettagli, tante volte fino al raggiungimento di un certo livello di qualità stabilito in precedenza.

Euristiche per sviluppare scenari e use case:

- Usare gli scenari per comunicare con utenti e validare delle funzionalità
- Prima rifinire un singolo scenario per capire le assunzioni dell'utente sul sistema, poi definire vari scenari non molto dettagliati per definire lo scope del sistema e validare con l'utente.
- Usare mock-up come supporto visuale ma solo dopo aver definito in maniera sufficientemente stabile le funzionalità del sistema
- Presentare all'utente diverse alternative
- Dettagliare verticalmente quando lo scope del sistema e le preferenze dell'utente sono state ben comprese e validare con l'utente

Identificazione delle relazioni tra attori e use-case

Per ridurre la complessità e migliorare la comprensione dello use case model è possibile identificare le relazioni tra gli use case e descrivere il sistema in layer di funzionalità.

Esempi:

- *Extend* per separare casi eccezionali e flussi comuni di eventi
- *Include* per ridurre la ridondanza tra gli use case

- *Communication* per rappresentare il flusso di informazioni durante gli use case; usare stereotipi come "*initiate*" e "*participate*"

Alcune euristiche: preferire *extend* per condizioni eccezionali e opzionali nel flusso degli eventi perchè permette una descrizione più modulare.
Se usassimo include, ogni passo richiederebbe un check per passare al caso d'uso d'eccezione.
Ricordare che la principale distinzione tra i due costrutti è che nell'include l'evento incluso è descritto nel flow degli eventi dello use case sorgente, mentre nell'extend come precondizione.
In altre parole per le relazioni di include ogni use case incluso deve specificare dove viene invocato, mentre nell'extend.

Identificazione dei primi oggetti di analysis

Identificare gli **oggetti partecipanti** per ogni use case e costruire un glossario come primo passo prima della fase di analysis. Il glossario è incluso nella specifica dei requisiti ed evolve nel tempo.
Euristiche per identificare gli iniziali oggetti di analysis:

- Termini che gli svilupattori o utenti devono chiarire per capire lo use case
- Sostantivi ricorrenti
- Entità e processi del mondo reale di cui il sistema deve tener traccia
- Casi d'uso
- Sorgenti di dati (input ed output)
- Artifact con cui l'utente interagisce

Infine effettuare un cross check per assicurare il completamento dell'identificazione degli oggetti rispetto agli use case.

Identificazione dei nonfunctional requirement

Identificare e descrivere tutti gli aspetti del sistema non direttamente correlati ai comportamenti funzionali: in particolare le due grandi categorie sono **quality requirements** e **constraints**.

Spesso non si è in grado di quantificare i requisiti di qualità di interesse, come ad esempio la manutenibilità; si possono usare però delle euristiche che ci fanno capire quando un sistema è manutenibile: un sistema ben documentato e ben scritto è sicuramente più manutenibile rispetto a un sistema non ben documentato.
Possibile creare anche delle metriche: ad esempio la densità di commento (quanti linee di commento rispetto al totale di linee di codice).

I requisiti di qualità sono sempre riferiti a un sistema gerarchico: la qualità è vista come un albero di varie sfaccettature, ognuna delle quali descritta attraverso molteplici parametri.

Un modo per andare a definire i requisiti di qualità è usare un approccio di tipo **Goal Question Metrics** (**GQM**): darsi un obiettivo di qualità, mappare l'obiettivo su una serie di domande e definire quali sono gli attributi che possono catturare il parametro di qualità.
Esistono molti esempi di metodi per elicitare i requisiti non funzionali, uno degli schemi che è possibile usare è proprio lo schema FURPS+ visto precedentemente.

RAD - Requirement Analysis Document

L'output che descrive in maniera esaustiva le fasi di requirement elicitation e analysis è un documento chiamato **Requirement Analysis Document** (**RAD**). Questo documento è destinato a clienti, utenti, project management, system analyst e system designer.

I problemi di qualità devono essere cercati nella documentazione (a partire dallo statement iniziale fino al codice, perchè tutto è documentazione) attraverso un processo di *inspection*: meccanismo di

controllo incrociato del contenuto del documento.

Questo può essere fatto in maniera sistematica o libera.
Le operazioni di inspection possono essere fatte anche sul codice: esistono approcci che premiano prima ancora di fare il controllo di qualità dinamico (il testing).

Il documento è essenzialmente composto da 4 sezioni:

- *Introduzione*: scopo del sistema, obiettivi del progetto, overview generale
- *Sistema corrente*: nel caso in cui il sistema dovrà sostituire un sistema già esistente in questa sezione si desrivono le funzionalità e i problemi di quest'ultimo;
- *Sistema proposto*: comprende requirement elicitation e analysis del nuovo sistema, divise in 4 sottosezioni:
  - *Overview*: vista funzionale
  - *Requisiti funzionali*
  - *Requisiti non funzionali*
  - *Modelli*: scenari, use case, object model, dynamic model ed eventuali mock-up dell'interfaccia utente e dei path navigazionali che il sistema dovrà avere.
- *Glossario*

Articolo - Software Quality Factors

Quando si analizzano i requisiti di qualità di un software non è possibile ignorare la loro natura **multidimensionale**, la grande quantità di attributi e caratteristiche che possono definire tale qualità. Possiamo raggruppare questi attributi in gruppi definiti come **software quality factors**.

Modello classico di McCall per i software quality factors

Questo modello classifica i requisiti di qualità del software in 11 quality factors, raggruppati in 3 categorie:

- **Product operation**: correttezza, affidabilità, efficienza, integrità, usabilità
- **Product revision**: manutenibilità, flessibilità, testabilità
- **Product transition**: portabilità, riusabilità, interoperabilità

Possiamo costruire un albero gerarchico dei quality factors grazie a questi tre principali rami.
Analizziamo più approfonditamente questi fattori:

Product Operation

Correttezza

Correlata agli output del sistema, definisce:

- L'accuracy desiderata dell'output
- La completezza richiesta delle informazioni di output
- L'aggiornamento richiesto delle informazioni (in accordo con la frequenza dell'aggiornamento dei dati)
- Il tempo di risposta desiderato per ottenere le informazioni richieste (tempo di reazione)

Reliability

Determina il massimo failure rate permesso dal sistema, la massima percentuale di downtime e il massimo tempo di recovery. Può riferirsi sia all'intero sistema che a una o più delle sue singole funzioni.

Efficienza

Ha a che fare con le risorse hardware necessarie per portare a termine le sue funzioni: capacità computazionali, capacità in termini di memoria e di disco, velocità di trasmissione delle informazioni e di comunicazione.

Integrità

Relativa alla sicurezza del sistema: capacità del software di prevenire accessi non autorizzati, permettere l'accesso alle informazioni solo a chi ne ha l'autorizzazione, etc.

Usabilità

I requisiti di usabilità sono legati alle operazioni, la produttività dell'utente, vista come numero medio di transazioni eseguibili per ora e come tempo medio speso per addestrare un nuovo employee all'utilizzo del sistema.

Product Revision

Riguardano le attività di manutenzione correttiva (correzione di fault e failure), adattiva (adattamento del sotfware a nuove circostanze) e perfettiva (miglioramenti del software esistente).

Manutenibilità

Determina lo sforzo necessario per utenti e personale di manutenzione per identificare i motivi dei failure del sistema e per correggere e verificare il successo della correzione. Per valutare la manutenibilità si osserva la struttura modulare del software, la documentazione interna, i manuali, etc.

Flessiblità

Capacità di supportare le attività di manutenzione adattiva: queste attività includono risorse necessarie per adattare un software a una varietà di clienti, prodotti, etc.
Questo requisito riguarda ance la manutenzione perfettiva, come cambiamenti e aggiunte al software per migliorare i suoi servizi e adattarlo ai cambiamenti tecnologici e commerciali dell'ambiente.

Testability

Ha a che fare col processo di testing del sistema e prende in esame la facilità con cui è possibile testare il software, possibilità di effettuare diagnostiche automatiche per il controllo del corretto funzionamento del sistema e rilevazione dei failure.

Product Transition

Riguarda l'adattamento del software ad altri ambienti e le interazioni con altri sistemi.

Portabilità

Adattamento del sistema ad altri ambienti caratterizzati da diversi hardware, sistemi operativi, etc.

Riusabilità

Requisito bidirezionale:

- Utilizzo di un modulo o dell'intera applicazione di un sistema esistene in un nuovo progetto
- Sviluppo di un modulo o più moduli (o intero progetto) in modo tale da essere riutilizzato in altri progetti futuri.

Interoperabilità

Riguarda la creazione di interfacce con altri sistemi.

Altri modelli

Esistono molti altri modelli che rappresentano i software quality factor come quello di Boehm, il modello FURPS, il modello ISO/IEC 25010, etc. che possono includere altri fattori non presenti nel modello di McCall, come:

- Evolvability
- Extensibility
- Manageability
- Modifiability
- Etc.

Per cercare di fare una valutazione più oggettiva vorremmo poter trovare metriche di valutazione di tutti questi attributi e sottoattributi di qualità, cercando di preferire quindi quelli quantitativi a quelli qualitativi.

5. Analysis

L'obiettivo della fase di Analysis è quello di produrre un modello che descrive il dominio applicativo in termini di oggetti, dei loro comportamenti, relazioni e la loro organizzazione a partire dalla formalizzazione dei requisiti.
L'**analysis model** deve essere corretto, completo, consistente, non ambiguo e verificabile.

Si compone di tre modelli individuali:

- **Modello funzionale**: modello di output della fase di requirement elicitation costituito da use case e scenari
- **Analysis object model**: class e object diagram; si concentra sui concetti individuali manipolati dal sistema, le loro proprietà e relazioni; è un dizionario visuale dei concetti principali visibili all'utente
- **Dynamic model**: sequence e state-machine diagram; si concentra sui behavior del sistema attraverso diagrammi che rappresentano le interazioni tra oggetti di uno use case e i comportamenti e le transizioni di stato di un singolo oggetto di interesse del sistema.

Per fare la transizione dal dominio del problema al dominio della soluzione usiamo un approccio che tende a strutturare lo spazio delle classi, dei primi oggetti in 3 grandi famiglie, 3 tipi di oggetti:

- **Entity**: rappresentano le informazioni persistenti del sistema; qualunque concetto del dominio applicativo di cui si sente la necessità di raccogliere informazioni
- **Boundary**: rappresentano le interazioni tra gli attori e il sistema; anche questi derivano dalla descrizione del problema
- **Control**: oggetti che realizzano gli use-case, i processi all'interno del sistema; attraverso il boundary interagiscono con l'utente e realizzano uno use-case; sostanzialmente vanno a modificare lo stato di un pezzo delle entità di dominio

Attività di Analysis - Come passare da use case agli oggetti

Identificazione degli entity object

Esistono delle euristiche per identificare oggetti che possono costituire delle entità di interesse:

- Nomi ricorrenti negli use case

- Entità e attività del mondo reale che il sistema deve tracciare
- Sorgenti di dati

Scegliere nomi univoci quanto più simili a quelli usati dagli utenti finali/specialisti del dominio applicativo. Scrivere delle descrizioni sugli oggetti per chiarire i concetti e evitare incomprensioni.

Identificazione dei boundary object

Rappresentano l'interfaccia tra il sistema e gli attori. In ogni use case ogni attore interagisce con almeno un boundary object; essi collezionano informazioni dall'attore e le traducono in un form che può essere usato sia da entity che da control object.

Euristiche:

- Identificare controlli della UI di cui l'utente ha bisogno per iniziare uno use case
- Form necessari per inserire dati
- Messaggi e notifiche con cui il sistema risponde all'utente

Identificazione dei control object

Responsabili della coordinazione di boundary e entity object. Non hanno una controparte concreta nel mondo reale ma spesso esiste una relazione stretta tra uno use case e un control object.
I control object sono spesso creati all'inizio degli use case e cessano la loro vita alla fine.
Collezionano informazioni dai boundary object e le inviano alle entity.

Euristiche:

- Un control object per use case (o un control object per attore in uno use case)
- Il tempo di vita di un control object deve coprire l'intero use case o la sessione di un utente; fare attenzione alle condizioni di entry ed exit degli use case

Sequence Diagram - Mapping tra use case e object

In un sequence diagram vogliamo modellare la sequenza di interazioni tra gli oggetti che realizzano uno use case, oltre che il tempo di vita degli oggetti che vi partecipano.
In orizzontale rappresentiamo con delle frecce i messaggi scambiati tra gli oggetti o tra attore e oggetti, in verticale si rappresenta il tempo.

In genere la seconda colonna del diagramma rappresenta un boundary con cui l'attore interagisce per iniziare lo use case, la terza è un control object per la gestione del resto delle operazioni dello use case.

Oltre a modellare le interazioni, tramite il sequence diagram facciamo una assegnazione delle responsabilità ad ogni oggetto nella forma di un insieme di operazioni.
Se l'operazione di un oggetto appare in più use case allora il suo comportamento deve essere sempre lo stesso.

Euristiche:

- Prima colonna: attore iniziatore
- Seconda colonna: boundary object
- Terza colonna: control object
- I control object sono creati dai boundary che iniziano lo use case
- I boundary sono creati dai control object
- Gli entity object vengono acceduti tramite control e boundary, ma non viceversa

Modellare interazioni tra oggetti con le CRC Cards

Una alternativa ai sequence diagram sono le **CRC Card**, molto utili per le sessioni di modellazione in team.
Gli sviluppatori e gli esperti di dominio analizzano uno scenario e identificano le classi coinvolte per la realizzazione di quest'ultimo.
Si costruisce una card per ogni classe e si definiscono **responsabilità** e **oggetti collaboratori**, ossia le dipendenze che quella card ha con altre card.

I sequence diagram sono molto utili per documentare una sequenza di interazioni in modo compatto e preciso, tuttavia le CRC card sono ideali per un gruppo di sviluppatori e per rifinire e iterare la struttura di un oggetto durante sessioni di brainstorming, in quanto sono più semplici da creare e da modificare.

Costruzione dei class diagram

Identificazione di associazioni

Una **associazione** è una relazione tra due o più classi. Le associazioni sono utili per chiarire l'analysis model.
Ogni associazione ha:

- Un **nome** (opzionale e non per forza univoco globalmente)
- Un **ruolo** ad ogni estremo che identifica la funzione di ogni classe partecipante
- Una **molteplicità**, che identifica il numero possibile di istanze di una classe che partecipa alla associazione

Euristiche per le associazioni:

- Esaminare predicati verbali
- Usare qualificatori per identificare attributi chiave
- Eliminare associazioni derivabili da altre
- Non esagerare con le associazioni, rischiano di rendere il modello troppo complesso

Identificazione di aggregati

Le **aggregazioni** sono particoari associazioni che definiscono una relazione di parte-tutto.
Distinguiamo:

- **Composizione**: l'esistenza della parte dipende dal tutto
- **Aggregazione condivisa**: parte e tutto possono esistere indipendentemente

Le aggregazioni consentono di organizzare i concetti del dominio applicativo in gerarchie o in grafi orientati.
Spesso si usano aggregazioni per le user interface, per aiutare l'utente a navigare attraverso molte istanze.

Identificazione di attributi

Sono proprietà individuali degli oggetti; è importante considerare solo le proprietà rilevanti rispetto al sistema che si vuole realizzare.
Alcune proprietà rappresentate da oggetti non sono attributi ma definiscono delle associazioni tra le classi.
Gli attributi hanno:

- Un **nome**
- Un **tipo** che descrive l'insieme dei valori che esso può assumere
- Una breve descrizione

Gli attributi sono la parte meno stabile dell'object model: spesso molti attributi vengono scoperti in fasi successive del processo di sviluppo.

Euristiche:

- Esaminare frasi possessive
- Non rappresentare un attributo come un oggetto, piuttosto usare associazioni
- Descrivere ogni attributo

State Machine Diagram - Modellare comportamenti di un singolo oggetto

Gli **state machine diagram** rappresentano il comportamento del sistema dalla prospettiva di un singolo oggetto e dei suoi stati.
Importante considerare solo oggetti con un tempo di vita piuttosto rilevante, come control object, raramente entity e quasi mai boundary.
Ogni stato può essere dettagliato e rifinito maggiormente.

Ereditarietà tra oggetti

La relazione di ereditarietà è utile per eliminare ridondanze dal modello di analysis: se due o più classi condividono attributi o behavior è utile consolidare le somiglianze in una superclasse.
La superclasse può rappresentare anche una classe astratta.

Review dell'analysis model

L'analysis model è costruito in maniera iterativa e incrementale: tali iterazioni sono necessarie per far convergere il modello verso una specifica quanto più corretta possibile, utile per il design e l'implementazione degli sviluppatori.

Una volta che il numero di cambiamenti al modello è minimale, il modello diviene stabile ed è possibile effettuare una review tra sviluppatori e clienti.
La review può essere fatta andando a realizzare una serie di domande e di questionari per scoprire eventuali errori, incomprensioni e fare opportune modifiche.

6. System Design: Decomposizione del sistema

La progettazione di sistema è la trasformazione del modello di analysis in un **modello di system design**. Durante questa fase gli sviluppatori definiscono gli obiettivi di design e decompongono il sistema in sottosistemi, selezionano strategie di costruzione (hardware/software), strategie di gestione dei dati persistenti, del flusso di controllo globale, policy di controllo degli accessi e gestione delle condizioni di boundary.

Il risultato del system design è un modello che include una decomposizione del sistema in sottosistemi, e una chiara descrizione di ognuna di queste strategie.

La progettazione del sistema è composta essenzialmente da tre principali attività:

- *Identificazione dei design goal*: le qualità che il sistema dovrà ottimizzare; sono derivati dai nonfunctional requirement e guidano le decisioni di design degli sviluppatori.
- *Design della iniziale decomposizione in sottosistemi*: decomporre il sistema in parti più piccole sulla base di use case e analysis model; usare **stili architetturali standard** come punto iniziale di questa attività.
- *Refinement della decomposizione in accordo ai design goal*: in genere la decomposizione iniziale non soddisfa tutti i design goal e quindi è necessario un refinement.

Overview del system design

L'analysis model precedentemente prodotto non contiene informazioni sulla struttura interna del sistema, sulla sua configurazione hardware e su come il sistema dovrà essere concretamente realizzato. Il system design produce:

- Design goal
- Architettura del software: descrive la decomposizione in sottosistemi in termini di responsabilità e dipendenze di questi ultimi e mapping con l'hardware, oltre che le decisioni sulle policy di controllo del flusso, degli accessi e data storage.
- Boundary use case: configurazione del sistema, startup, shutdown e gestione delle eccezioni.

Concetti fondamentali del system design

Un **sottosistema** è una *Parte* coerente del *Sistema* dotata di interfacce ben definite che incapsulano stato e comportamento in delle classi.
Un Sottosistema è quindi sia una parte del Sistema sia un aggregato di parti.
Nel mondo Java, ad esempio, un sottosistema può essere una Classe o un insieme/aggregato di classi organizzate in un **package**.

Le componenti di un sistema possono essere sia **logiche** che **fisiche**.
Una **componente logica** è un sottosistema che non ha un equivalente esplicito a run-time, mentre una **componente fisica** si, come ad esempio un database server.

Le parti del sistema comunicano tra loro attraverso la metafora del **servizio**, un insieme di operazioni che condividono uno scopo comune. Ogni parte del sistema offre servizi ad altre parti (o anche servizi pubblici per altri sistemi) e allo stesso modo richiede servizi dalle altre.

La metafora del servizio è sviluppata a livelli di astrazione diversi: il più semplice è quello dei servizi offerti e richiesti dalle classi. Il main crea le istanze delle classi, e queste si scambiano messaggi per richiedere e offrire servizi. Questo è il caso di un singolo processo.
Possiamo anche pensare però a programmi indipendenti che parlano tra di loro.
Salendo ancora possiamo avere anche sistemi che girano su macchine differenti in rete e che comunicano scambiando servizi, usando determinati protocolli, middleware, etc.
Concetto di **gerarchia dei servizi**.

L'insieme di operazioni che un sottosistema mette a disposizione di altri sottosistemi forma l'**interfaccia del sottosistema**, e include nomi delle operazioni, parametri e loro tipi, valori di ritorno.
Il system design si concentra sul definire i servizi forniti dai sottosistemi, mentre l'object design sulla creazione delle **Application Programming Interface** (**API**), che rifiniscono ed estendono le interfacce.

Il **coupling** (accoppiamento) è la misura delle dipendenze tra due sottosistemi, mentre la **cohesion** è la misura delle dipendenze tra le classi di un sottosistema.
Una decomposizione ideale deve mirare a **minimizzare il coupling** esterno e **massimizzare la cohesion** interna.
Minimizzare il coupling significa rendere le parti indipendenti tra loro e far sì che il cambiamento di un sottosistema abbia un impatto minimo sul resto del sistema.

Come nella modellazione concettuale esistono i principi di chunking e dettagliamento a vari livelli di astrazione, esistono due tecniche analoghe per l'organizzazione dei sottosistemi:

- **Layering**: sistema organizzato in gerarchie di sottosistemi, ognuna con dei servizi di alto livello per il sottosistema superiore, e dei servizi di basso livello per i sottosistemi inferiori.
- **Partitioning** organizza i sottosistemi come dei peer che forniscono mutuamente servizi gli uni con gli altri.

Un **layer** è un gruppo di sottosistemi che fornisce servizi realizzati usando servizi di un altro layer. Ogni layer dipende dai layer sottostanti, ad eccezione dell'ultimo livello (bottom layer), mentre il top layer è quello che non è usato da nessun altro.
In una **closed architecture** ogni layer può accedere solo ai livelli immediatamente sottostanti, in una **open architecture** un layer può accedere anche più in profondità.

Un esempio di closed architecture è il modello OSI, costituito da 7 layer, ognuno dei quali è responsabile di funzioni ben definite a un certo livello di astrazione, e che fornisce servizi sulla base dei servizi dei layer sottostanti.
Un altro esempio è il sistema UNIX.

Un esempio di *open architecture* è il framework Java *Swing*: il layer più basso è fornito dal window manager del SO, AWT fornisce poi un'interfaccia astratta per applicazioni dotate di GUI.
Swing è una libreria di oggetti di UI che estende AWT e fornisce widget utili a sviluppare una GUI.
L'applicazione interagisce con Swing e, talvolta può bypassare il layer Swing e accedere ad AWT.
Un altro esempio di architettura open è Android.

Più in generale la decomposizione è il risultato combinato di partitioning e layering: prima si effettua una partizione del sistema in sottosistemi top-level, successivamente ogni sottosistema viene decomposto in layer di livello inferiore.
Ogni sottosistema aggiunge overhead a causa della sua interfaccia con altri sistemi.
Un eccessivo partitioning o layering può quindi incrementare considerevolmente la complessità del sistema.

In genere esiste un trade-off tra *cohesion* e *coupling*: possiamo incrementare la cohesion decomponendo il sistema in sottosistemi più piccoli, tuttavia questo incrementerà il coupling a causa dell'aumento del numero di interfacce.
Una buona euristica ci dice che gli sviluppatori possono fare i conti con un numero di circa $\mathbf{7 \pm 2}$ concetti per ogni livello di astrazione.
Se ci sono più di 9 sottosistemi o se un sottosistema offre più di 9 servizi potrebbe essere opportuno rivedere la decomposizione.
Allo stesso modo anche il numero di layer dovrebbe rispettare la stessa euristica.

Software Architecture - Stili/soluzioni architetturali

Il modo in cui organizzo le componenti e i servizi che si offrono prende il nome di **Software Architecture**, la forma generale del nostro sistema.
I dettagli implementativi delle componenti verranno specificati in una fase successiva.

In letteratura, in merito alle metodologie agili, l'architettura del sistema può essere sviluppata insieme al purpose del sistema, perchè anche il modo in cui realizzerò l'elicitazione del requisito e la definizione dei sequence e analysis può essere influenzata dalla forma che diamo al sistema.
Ad esempio quando già in fase di requisiti sappiamo che il sistema dovrà essere "web based": questo avrà un impatto realizzativo enorme.

Analizziamo diversi **stili architetturali** come base per la costruzione dell'architettura di un sistema.

Pipe and Filter

Meccanismo che permette la comunicazione tra componenti. I sottosistemi elaborano i dati ricevuti da un insieme di input e inviano i risultati agli altri sottosistemi come output.
Ogni sottosistema è chiamato **filter** mentre le associazioni tra essi sono dette **pipe**.

Ogni filter conosce solo il contenuto e il formato dei dati ricevuti dai pipe di input, ma non il filter che li ha prodotti.

Ogni filter è eseguito in maniera concorrente e la sincronizzazione viene realizzata grazie ai pipe.
Questo stile architetturale è modificabile: i filter possono essere sostituiti con altri o riconfigurati per altri scopi.

Il miglior esempio di applicazione di questo stile architetturale è la Unix shell, grazie alla possibilità di combinare i comandi (processi).
Questo stile è particolarmente adatto a sistemi che applicano trasformazioni a stream di dati senza l'intervento dell'utente, ma non per sistemi più complessi e interattivi.

Repository

In questo stile architetturale i sottosistemi accedono e modificano una singola struttura dati centrale chiamata **repository**.
I sottosistemi sono relativamente indipendenti e interagiscono solo attraverso il repository.
Il flusso di controllo può essere dettato o dal repository centrale o dai sottosistemi.

I repository sono tipicamente usati per i DBMS, sistemi bancari, etc.
Avere i dati centralizzati rende più semplice gestire problemi di concorrenza e integrità.
Anche i compilatori (per la tabella dei simboli) seguono questo stile architetturale.

Una volta definito un repository centrale è molto semplice aggiungere nuovi servizi con nuovi sottosistema.
Lo svantaggio principale dei sistemi a repository è il bottleneck dal punto di vista delle performance e modificabilità.
Il coupling tra ogni sottosistema e il repository è alto e ciò rende difficile cambiare il repository senza impattare fortemente su tutti i sottosistemi.

Model View Controller

I sottosistemi sono classificati in tre diversi tipi:

- **Model**: mantendono la conoscenza di dominio
- **View**: mostrano le informazioni all'utente
- **Controller**: gestiscono le interazioni con l'utente

Si tratta di uno speciale caso di modello a Repository dove il Model implementa la struttura dati centrale e i control object dettano il flusso di controllo.
Questo stile richiama molto la distinzione tra gli oggetti dell'analysis model, ossia entity, boundary e control.
In relazione a questo approccio ci sono i concetti di **inversione del controllo** e **observer design pattern**.

Questo stile architetturale è molto usato per le GUI, sistemi interattivi.
Anche il framework Spring utilizza questo approccio.
Come lo stile Repository, anch'esso è caratterizzato dagli stessi bottleneck relativi alle performance.

Client-Server

In questo stile architetturale distinguiamo una componente, **server** che fornisce servizi ad altre componenti, i **client**.
Si tratta quindi di un modello asimmetrico in cui c'è una distinzione di ruoli.

Il colloquio viene iniziato sempre e solo dal client, il quale richiede un servizio, mentre il server, che non ha alcuna consocenza dei client, risponde alle richieste offrendo (se possibile) tali servizi.

Quando l'architettura client server è a singolo server (come nel caso di un database) si tratta di una specializzazione del modello repository.
Oggi, in particolare sul web, un singolo client può accedere a dati su tantissimi differenti server, infatti questo stile architetturale è applicabile anche per sistemi detti **distribuiti**, che gestiscono grandissime quantità di dati.

La comunicazione tra le componenti può essere realizzata in vari modi: es. protocolli di comunicazione o usando un middleware.
Es. *Corba* è un middleware che consente di realizzare la comunicazione tra oggetti (componenti) distribuiti, ossia non sulla stessa macchina/processo, ma in processi distinti e possibilmente su macchine diverse, evitando di gestire direttamente socket etc.
Il middleware realizza quindi un meccanismo di astrazione, stabilisce un bus, un canale di comunicazione sul quale gli oggetti si registrano.
Corba è nato per usare quindi la stessa metafora dei servizi tra oggetti, ma su macchine diverse.
Oggi sempre più diffuso l'utilizzo di HTTP non solo come protocollo Web adatto a restituire un payload HTML ma anche altri tipi, come XML e JSON.

Peer-to-Peer

Deriva dallo stile client-server eliminando il concetto di asimmetria dei ruoli tra client e server: ogni sottosistema può essere sia client che server, ossia ogni sottosistema può richiedere e offrire servizi.
La comunicazione in questo caso, quindi non è asimmetrica.

Un esempio di applicazione dello stile architetturale peer-to-peer è un database che accetta richieste dall'applicazione e notifica ad essa quando certi dati sono stati cambiati.
Sono sistemi più complessi da progettare rispetto ai client-server perchè introducono la possibilità di deadlock e complicano il flusso di controllo.

Three-tier

Sottosistemi client-server organizzati in 3 layer:

- **Layer di Interfaccia**: include i boundary object con cui l'utente interagisce (finestre, form, pagine web, etc.)
- **Layer di logica applicativa**: include tutti i control ed entity object
- **Storage layer**: realizza la gestione, recupero e query degli oggetti persistenti.

Lo storage layer, analogamente come un Repository, può essere condiviso tra diverse applicationi che operano sugli stessi dati.
La separazione tra layer di interfaccia e di logica applicativa permette lo sviluppo di diverse user interface con la stessa logica applicativa.

Multi-tier (Four-tier)

Si tratta di un'architettura three-tier dove il layer di interfaccia è diviso in due layer: **Presentation Client layer** e **Presentation Server layer**.
Il primo si trova sulle macchine degli utenti, mentre il secondo su uno o più server.
Questo stile architetturale permette la creazione di una grande varietù di presentation client che possono riutilizzare alcuni oggetti.
Un esempio può essere un sistema informativo bancario, che include interfacce web per home banking, client per dipendenti, etc.
Tutto ciò che è condiviso ai diversi client è definito ed elaborato nel Presentation Server per rimuovere ridondanza.

Possiamo distinguere quindi anche i client:

- **Client pesante**: i client sono dei programmi che sono fisicamente installati su delle macchine e si connettono al server; in questa configurazione bisogna tenere aggiornati i programmi client; il server gestisce solo la parte dati, GUI e logica applicativa sono gestiti lato client.
- **Client leggero** (e universale): si tratta di web application che sfruttano come client un web browser e colloquiano con l'application server tramite esso; in questo caso non c'è bisogno di alcuna installazione del client, né aggiornamento; il client è semplicemente una GUI che prende dei dati e li trasmette a un server; logica applicativa e mondo dei dati sono nella componente server.

Concetto di Binding - Stile architetturale ad eventi (Event-driven)

Accoppiamento di due parti che devono parlare tra loro; il binding può essere statico o dinamico.
Con un meccanismo che consente di creare classi figlie a runtime il late binding diventa un late binding di tipo **open**.

Un esempio sono le architetture ad **eventi**, o **publish-subscribe**.
Immaginiamo un sistema con una componente runtime di dispatching (dispatcher) dove è possibile registrarsi per rispondere ad un certo tipo di evento.

Java utilizza per default il late binding grazie alla possibilità di usare variabili che sono istanze di classe diverse, ma è un late binding closed.

I sistemi publish-subscribe sono realizzabili in microscala (es. app Android) ma anche macro (es. alcuni sistemi della Pubblica Amministrazione - CART Toscana).

Articolo - Introduction to Software Architecture

Il progresso dei linguaggi di programmazione moderni ha permesso parallelamente anche un innalzamento del livello di astrazione nella progettazione del software.
Alla fine degli anni '60 i programmatori intuirono che è fondamentale la scelta delle giuste strutture dati per la semplificazione dello sviluppo di un programma.
Allo stesso modo un'altra delle intuizioni più importanti è stata l'introduzione del concetto di Abstract Data Type.

Stili architetturali più comuni

Uno stile architetturale definisce una famiglia di pattern di organizzazione della struttura di un sistema. Più nello spceifico determina il vocabolario di componenti e connettori usati in tale stile, insieme a una serie di constraint su come essi sono combinati.

Pipe and Filter

Ogni **filter** ha input ed output, legge flussi di dati in input e produce flussi di dati in output attraverso componenti dette **pipe**. I filter sono indipendenti e non condividono lo stato con altri filter né ne conoscono l'identità.

Specializzazioni comuni di questo stile sono: la **pipeline** che restringe la topologia a una sequenza lineare di filter, i **bounded pipe**, che restringono la quantità di dati che può transitare in un pipe e i **typed pipe** che richiedono un tipo definito di dati passati tra due filter.
Un caso degenerato di pipeline è quando ogni filter elabora i dati in input come singola entità → si parla di **batch sequential system**.

I sistemi pipe and filter permettono di definire l'I/O del sistema come composizione dei comportamenti dei singoli filter; inoltre supportano la sostituzione e aggiunta di nuovi filter, l'esecuzione concorrente.
I principali svantaggi sono legati alle performance e alla complessità, in relazione alla trasmissione e al gran numero di parse sui dati nel passaggio tra i filter.

Data Abstraction e Object-Oriented Organization

In questo stile i dati e le operazioni primitive sono incapsulate in **abstract data type** o oggetti. Le componenti sono oggetti o istanze di ADT.
Gli oggetti interagiscono tra loro scambiandosi messaggi che provocano l'invocazione di funzioni e sono responsabili di preservare l'integrità della rappresentazione interna, nascosta agli altri oggetti.

Grazie al concetto di *information hiding* è possibile cambiare un'implementazione senza avere un impatto sui programmi client che sfruttano tali oggetti.
Svantaggio principale dei sistemi OO: per interagire tra loro gli oggetti devono conoscere l'identità degli altri oggetti (a differenze dei pipe and filter) .

Event-based, invocazione implicita

L'invocazione implicita è una tecnica alternativa rispetto a quella esplicita che consente a una componente di annunciare uno o più eventi e altre componenti possono registrarsi rispetto all'evento di interesse associando una procedura.
Quando l'evento viene annunciato il sistema invoca tutte le procedure registrate per quell'evento specifico.

Le componenti di questo stile architetturale sono moduli le cui interfacce forniscono funzioni e insiemi di eventi.
Un componente può registrare alcune delle sue procedure con degli eventi del sistema, in modo tale che esse possano essere invocate a runtime.
Le componenti che annunciano gli eventi non sanno quali componenti saranno effettivamente affetti, ossia quali procedure saranno effettivamente invocate e verranno eseguite.

Un importante vantaggio di questo stile è che ogni componente può essere introdotta nel sistema semplicemente registrandola per gli eventi, inoltre le componenti possono essere facilmente sostituite da altre senza alcun impatto alle interfacce di altre componenti.

Lo svantaggio principale dell'invocazione implicita è che la componente che annuncia gli eventi oltre a non conoscere chi verrà invocato, non conosce nemmeno l'ordine di invocazione. Un altro problema è legato allo scambio di dati: alcun dati possono essere passati con l'evento ma in altre situazioni è necessario usare un repository condiviso per le nterazioni, causando problemi di performance e gestione delle risorse.

Sistemi a layer

Sistema organizzato gerarchicamente, ogni layer fornisce servizi ai layer superiori e funge da client per i layer inferiori.
I connettori tra i layer sono definiti da protocolli che determinano come i layer dovranno interagire.

Permettono di implementare sistemi come sequenze di passi incrementali, a crescenti livelli di astrazione. Difficile standardizzare le implementazioni dei bassi livelli di astrazione (come nel caso del modello OSI).

Repository

Due componenti principali: una struttura dati centrale e una collezione di componenti indipendenti che operano su di essa.

Se lo stato del repository è il principale attivatore dei processi da eseguire parliamo di una **blackboard**, un modello che si rifà alla metafora di una sessione di brainstorming di un gruppo di specialisti.
Il modello blackboard ha 3 parti principali:

- *Sorgenti di conoscenza*: moduli che forniscono conoscenza specifica necessaria per l'applicazone che interagiscono tramite la blackboard.
- *Struttura dati blackboard*: un repository condiviso di problemi, informazioni, soluzioni parziali, una sorta di libreria dinamica di contributi al problema; i continui cambiamenti da parte delle sorgenti di conoscenza portano in maniera incrementale alla soluzione del problema.
- *Control*: controllano il flusso dell'attività di problem solving del sistema.

Table Driven interpreters

Un interprete include generalmente ha 4 componenti:

- Un engine che fa da interprete
- Una memoria che contiene lo pseudo-codice da interpretare
- Una rappresentazione dello stato dell'engine
- Una rapresentazione dello stato corrente del programma

Gli interpreti sono spesso usati per costruire macchine virtuali.

Altre architetture

- **Processi distribuiti**: alcune sono caratterizzate dalle loro caratteristiche topologiche (anello, stella, etc.), altre in termini di protocolli inter-processo usati per la comunicazione; una forma comune di architettura distribuita è quella client-server: un processo server fornisce servizi ad altri processi client; il server non conosce l'identità nè il numero dei client, mentre i client conoscono l'identità del server e vi accedono tramite delle procedure remote.
- **Organizzazione main program/subroutine**: rispecchia l'organizzazione del linguaggio in cui il sistema è scritto; per linguaggi procedurali si tratta di un sistema organizzato da un main e un insieme di subroutine dove il main funge da programma driver.
- **Architetture software domain-specific**: applicazioni con organizzazione specifica rispetto al dominio applicativo di interesse
- **State transition systems**: sistemi definiti in termini di stati e transizioni
- **Process control systems**: sistemi usati spesso per controlli dinamici di un ambiente fisico; sono caratterizzati da un feedback loop nel quale degli input provenienti da sensori sono usati per determinare degli output e produrre un nuovo stato dell'ambiente.

Architetture eterogenee

Fino ad ora abbiamo parlato di stili architetturali puri, ma in realtà molti sistemi possono includere molti degli stili appena citati in maniera combinata.

Casi di studio

Caso 1

Nel 1972 Parnas propose un problema di shift circolare di parole all'interno di un insieme di linee.
Egli voleva mettere in contrasto diversi criteri per la decomposizione di un sistema in moduli e descrisse varie soluzioni:

1. *Main/subroutine*: i dati venivano comunicati tra le componenti attraverso uno storage condiviso e la comunicazione tra le componenti era un protocollo lettura-scrittura sena vincoli con un programma che garantisce l'accesso sequenziale ai dati. Questa soluzione aveva però numerosi

problemi: cambiare un formato di dati o degli algoritmi avrebbe impattato su tutti i moduli; soluzione che supporta poco il riuso di codice.
2. *Abstract Data Type*: sistema decomposto in 5 moduli, ognuno fornisce un'interfaccia che permette alle altre componenti di accedere ai dati solo tramite procedure permesse dall'interfaccia. Questa soluzione ha dei vantaggi rispetto alla prima perchè sia i dati che gli algoritmi possono essere cambiati nei singoli moduli senza avere impatto sugli altri; inoltre è supportato anche il riutilizzo di codice.
3. *Invocazione implicita*: come la prima si basa su una componente con dati condivisi ma con due importanti differenze: la prima è che l'interfaccia ai dati è più astratta e la seconda è che le procedure sono invocate implicitamente quando i dati vengono modificati; questa soluzione supporta facilmente miglioramenti funzionali al sistema, aggiunta di moduli e registrazione per l'invocazione dettata da eventi sul cambiamento dei dati; supporta il riutilizzo di codice visto che l'invocazione implicita dei moduli si basa solo sull'esistenza di certi eventi esterni che vengono triggerati; tuttavia questa soluzione presenta problemi nel controllare l'ordine di invocazione di questi moduli, inoltre a causa delle invocazioni data-driven, questa implementazione tende a usare più spazio delle altre.
4. *Pipe and filter*: soluzione a pipeline con 4 filter, ognuno dei quali processa i dati e li invia al filter successivo; tra i vantaggi vi sono il flusso di elaborazione intuitvo, il supporto al riutilizzo di codice (ogni filter funziona in isolamento) e la possibilità di aggiungere e sostituire facilmente nuovi filter al sistema; tra i difetti vi è l'impossibilità di modifcare il design per supportare un sistema interattivo, e l'inefficienza dal punto di vista dello spazio utilizzato (ogni filter deve copiare tutti i dati in output).

Queste soluzioni possono essere confrontate analizzando su una tabella i loro pregi e i loro difetti nel venire incontro agli obiettivi di design del problema.

Parnas affermò che le soluzioni a dati condivisi erano deboli in merito al supporto ai cambiamenti di dati e algoritmi, ma avevano buone performance e permettevano la facile aggiunta di nuove componenti.
La soluzione con ADT permetteva di cambiare le rappresentazioni dei dati e il riutilizzo di codice senza compromettere le performance, ma le interazioni tra componenti erano nei moduli stessi, quindi aggiungere nuove funzioni o cambiare l'algoritmo principale poteva portare a molti cambiamenti nel sistema.
La soluzione ad invocazione implicita, pur supportando l'aggiunta di nuove funzionalità, soffriva di problemi legati alla condivisione dei dati (scarso supporto ai cambiamenti e riuso) e introduceva overhead.
La soluzione pipe and filter permetteva l'aggiunta di nuovi filter al flusso ma presentava notevole overhead dovuto al parsing dei dati nei pipe tra i vari filter.

Caso 2

Progetto di sviluppo di un framework architetturale per oscilloscopi della Tektronix. Il primo tentativo di soluzione fu un *modello object-oriented* basato sugli oggetti di dominio.

Il secondo tentativo fu un *modello layered*: il core layer rappresentava le funzioni di manipolazione dei segnali, il successivo l'acquisizione delle forme d'onda, il terzo la manipolazione delle forme d'onda attraverso misure, trasformazioni, etc. , il quarto layer responsabile della digitalizzazione delle forme d'onda e relativa rappresentazione visuale, oltre che dell'interazione con l'utente.

Il terzo tentativo fu un modello *pipe and filter* che migliorava il modello layered perchè non isolava le funzioni in partizioni separate.

La quarta soluzione fu modificare il modello pipe and filter associando ad ogni filter un'interfaccia di controllo che permetteva il setting dei parametri dell'operazione di quel filter.

Caso 3

Analisi dell'architettura dei compilatori: inizialmente il processo di compilazione era visto come un processo sequenziale modellato come un una pipeline, ma con una componente dati condivisa che creava una simbol table per l'analisi lessicale.
Col passare del tempo i compilatori sono diventati molto più complessi, ma la struttura è rimasta sostanzialmente simile: parliamo di una struttura a repository simile a una blackboard per certi aspetti, dove l'informazione è centralizzata e le componenti indipendenti interagiscono solo attraverso i dati condivisi.

Caso 4-5-6

Troppo specifici, difficili e noiosi

7. System Design - Addressing Design Goals

Dopo aver definito i design goal e la decomposizione del sistema sono necessarie delle attività di design per rifinire la decomposizione in accordo ai design goal:

- *Mapping tra sottosistemi e hardware*
- Design dell'*infrastruttura di gestione dei dati persistenti*
- Specifica delle *policy di controllo degli accessi*
- Design del *flusso di controllo globale*
- Gestione delle *condizioni di boundary*

Molte delle decisioni di design sono guidate dai design goal e nonfunctional requirements. Alla fine del system design, tutti i design goal devono essere stati risolti e rispettati.

Mapping hardware/software

Consiste nell'andare a definire la configurazione hardware del sistema, le responsabilità funzionali dei nodi che lo costituiscono e la loro comunicazione.

In questa fase è possibile pensare a sottosistemi aggiuntivi dedicati al trasporto dei dati tra i nodi, gestione della concorrenza e problemi di affidabilità.
Le componenti off-the-shelf consentono di realizzare sistemi e servizi complessi in maniera più semplice ed economica: degli esempi sono componenti per le user interface e DBMS.
Tali componenti, tuttavia, devono essere incapsulate per minimizzare le dipendenze e avere la possibilità di sostituirle in futuro.

UML mette a disposizione un diagramma chiamato **component diagram** per rappresentare le componenti software le loro dipendenze mentre il **deployment diagram** per il mapping delle componenti sulle risorse hardware a disposizione.
I nodi sono rappresentati come *box* contenenti delle componenti e la comunicazione tra i nodi è rappresentata da linee stereotipate in base al tipo di comunicazione.

Selezionare la configurazione hardware include anche la possibilità di selezionare una macchina virtuale nel quale costruire il sistema, ossia anche il sistema operativo e le componenti software necessarie.

Una volta definita la configurazione hardware e le eventuali macchine virtuali, è necessario assegnare oggetti e sottosistemi ai vari nodi.
In generale allocare sottosistemi ai nodi hardware permette di distribuire le funzionalità e la potenza computazionale in base a dove è maggiormente necessaria.
Tuttavia questo introduce problemi relativi alla memorizzazione, trasferimento e sincronizzazione dei dati.

Identificazione e memorizzazione dei dati persistenti

I **dati persistenti** vivono anche al di fuori dell'esecuzione del sistema.
Come e dove memorizzare i dati influenza anche la decomposizione del sistema: in uno stile architetturale a repository, infatti, un sottosistema potrebbe essere interamente dedicato allo storage dei dati.

Prima di tutto bisogna identificare le entità persistenti da memorizzare: i primi candidati sono gli entity object identificati nella fase di analysis.
Non tutti, però, sono entità persistenti, nè tutti gli oggetti persistenti sono entity object, perchè potrebbe essere necessario memorizzare in maniera persistente anche informazioni relative a boundary object.

In generale identifichiamo gli oggetti persistenti esaminando tutte le classi che devono sopravvivere anche quando il sistema è spento o in caso di crash. Quando il sistema torna attivo questi oggetti verranno recuperati dal database.

Strategie di gestione dei dati persistenti

Una volta identificati gli oggetti è necessario capire come memorizzarli. La decisione di storage management è molto complessa e dettata dai requisiti non funcionali.
Esistono in generale tre opzioni di storage management:

- **Flat file**: usare dei semplici file, l'astrazione più semplice usata dai sistemi operativi per immagazzinare dati sotto forma di sequenze di byte. Tuttavia i file richiedono all'applicazione di tener conto di molte problematiche come l'accesso concorrente e la possibile perdita di dati in caso di system crash.
- **Database relazionale**: forniscono astrazioni dei dati ad un livello più alto dei flat file; i dati sono memorizzati in tuple di tabelle con uno **schema** predefinito (ossia tipi e attributi definiti); ogni tupla è un'istanza di un oggetto, ogni colonna è un attributo di quella classe di oggetti. I database relazionali forniscono servizi per la gestione concorrente, controllo degli accessi e crash recovery; essi però non sono particolarmente adatti per dati non strutturati.
- **Database object-oriented**: simili ai database relazionali, ma memorizzano i dati come oggetti e relative associazioni; riducono sensibilmente il tempo di sviluppo del sottosistema di storage ma sono più lenti dei database relazionali e hanno query tipicamente più difficili.

Esistono infine soluzioni di database **NO-R**, ossia non relazionali (o not only), tipicamente caratterizzati da modelli di natura gerarchica dove i dati vengono rappresentati da coppie chiave-valore.

Una volta definito il modello dei dati, esso dovrà interfacciarsi con il sistema mediante delle tecnologie che permettono lo scambio di dati tra il layer dei dati e quello applicativo.
Nel caso di DB di tipo SQL, ad esempio, il mondo Java si connette con il mondo SQL grazie a JDBC, una particolare API basata su ODBC, che consente di effettuare query in Java e ottenere come risultato un oggetto chiamato *cursore* che consente di navigare le tabelle del mondo SQL.
Viceversa attraverso JDBC è anche possibile serializzare oggetti Java e renderli persistenti all'interno di un DB SQL.
Anche per DB non relazionali come MongoDB esistono driver che permettono tale connessione.

Controllo degli accessi

Nei sistemi multi-utente, attori differenti hanno accesso a diversi dati e funzionalità. A seconda dei requisiti di sicurezza del sistema dobbiamo definire come definire il controllo degli accessi, ossia come gli attori verranno autenticati e come criptare i dati selezionati nel sistema.

In generale dobbiamo definire per ogni attore quali sono le operazioni a cui egli ha accesso e su quali oggetti.
Distinguiamo i concetti di authentication e authorization:

- **Autenticazione**: il riconoscimento di un utente che dimostra al sistema di essere colui che dice di essere. Es. Login mediante email e password. La fase di autenticazione termina, quando 'utente fornisce al sistema delle informazioni che consentono a quest'ultimo di riconoscerlo.
- **Autorizzazione**: una volta avvenuto il riconoscimento il sistema sa cosa può o non può fare quel determinato utente.

Authentication

La fase di autenthication può essere realizzata in molti modi: il modo più banale è attraverso il submit di una coppia username-password. Se entrambi i campi sono correttamente associati allora avviene l'autenticazione.

Questa modalità è oggi ritenuta debole, poiché la sola password non è più affidabile.
Esistono metodi di autenticazione più "forti" che vanno al di là dell'assunto "dimostrami di sapere qualcosa che solo tu potresti sapere".

Un esempio è l'*autenticazione a due fattori*: l'utente deve fornire anche un codice temporaneo per completare l'autenticazione. Ciò permette di dimostrare che l'utente *ha qualcosa che è suo*.
La password è un segreto; la 2FA è un segreto + un oggetto (il telefono, un codice temporaneo, etc.).
Ma è possibile andare oltre: metodi di autenticazione che usano biometrie (impronta digitale, riconoscimento dell'iride, riconoscimento facciale, etc.).

Al concetto di autenticazione va però associato il concetto di **encryption**: gli algoritmi di encryption permettono di tradurre messaggi di testo in messaggi criptati in modo che, anche se intercettati, non possano essere decriptati.

Authorization

La seconda parte del processo che serve per definire, per ogni utente del sistema, cosa può fare e su quali risorse.

Il controllo degli accessi può essere modellato attraverso una *matrice degli accessi*: le righe rappresentano tutti gli attori del sistema, le colonne le risorse; all'incrocio fra righe e colonne abbiamo gli **access right**, una lista di operazioni che quel dato utente può eseguite sulle risorse specifiche.
Questa matrice prende il nome di **global access table**.

Una alternativa consiste nell'utilizzare delle liste chiamate **access control list**: associamo ad ogni risorsa una lista di coppie attore-operazione. Si tratta quindi di leggere la matrice per colonne e non per righe: per ogni risorsa ogni utente può fare solo determinate operazioni.

Ci si può però anche focalizzare sull'utente: ad ogni utente associo la lista di coppie risorsa-operazione, che prende il nome di **capability**: ogni utente ha la sua lista di capability.

Queste due alternative sono quindi due modi diversi per leggere la stessa matrice.
L'utilizzo di una matrice degli accessi di questo tipo è un metodo di **static access control**.

Negli approcci più moderni si usa un altro tipo di rappresentazione perchè queste richiedono di conoscere a priori utenti e risorse, i quali possono essere anche molto numerosi.
Parliamo quindi di un altro approccio chiamato **dynamic access control**: i diritti di accesso alle risorse sono modellati dinamicamente.

L'idea è di mettere fra l'utente e la risorsa il concetto di **ruolo**: attribuisco dei permessi ai ruoli, e successivamente i ruoli agli utenti.
In questo modo se c'è una variazione di ruolo per un utente non c'è bisogno di cambiare i permessi ma soltanto il ruolo.

Questo prende il nome di **Role-Based Control Access**: definisco i ruoli e definisco in un certo momento per ogni utente il suo ruolo.

C'è un caso particolare in cui si torna molto spesso all'utilizzo della capability: è il caso dei permesso esclusivi su risorse private.
Es. il voto di un esame nel libretto di un singolo studente: ogni studente può vedere solo le proprie informazioni, non quelle di altri; parliamo quindi del concetto di *Owner*.

Articolo - Authentication and Access Control

Autenticazione

La user authentication è la base per molti tipi di controllo degli accessi per il riconoscimento degli utenti. Nell'RFC 4949 la user authentication è definita come un processo di due step:

- **Identificazione**: presentare un identificatore al sistema di sicureza per il riconoscimento dell'identità; in sostanza significa fornire al sistema una identità conosciuta.
- **Verifica**: presentare o generare informazioni di autenticazione che permettono di confermare l'accoppiamento tra entità ed identificatore.

Possiamo definire 4 generali metodi di autenticazione, non in mutua esclusione:

- **Fornire un'informazione che solo l'utente conosce**: es. password, PIN, risposte segrete, etc.
- **Fornire qualcosa che solo l'utente possiede**: smart card, chiavi fisiche → **token**
- **Fornire qualcosa dell'utente**: riconoscimenti biometrici (impronta digitale, retina, viso, etc.)
- **Riconoscimento biometrico dinamico**: riconoscimento vocale, calligrafia, ritmo di digitazione, etc.

Autorizzazione / Controllo degli accessi

Esistono due principali definizioni di access control:

1. processo per garantire o negare delle specifiche richieste per:
    1. ottenere informazioni e servizi
    2. entrare in specifiche strutture fisiche
2. processo con cui un sistema regola l'utilizzo delle risorse rispetto a una policy di sicurezza e permette l'accesso solo ad entità autorizzate

Il principale obiettivo della computer security è prevenire accessi non autorizzati di utenti e prevenire l'accesso a risorse a cui un dato utente non dovrebbe poter accedere.

Access Control Context

L'**audit** è un modo per valutare e testare in maniera indipendente il controllo degli accessi di un sistema rispetto alle policy stabilite, per rilevare eventuali **breach** e raccomandare specifici cambiamenti nel controllo, nelle policy o nelle procedure.

Tipicamente una funzione di autenticazione determina se un utente ha il permesso di accedere al sistema; in seguito la funzione di controllo degli accessi va a determinare se l'accesso dell'utente specifico è permesso.
Un security admin mantiene un database di autorizzazione che specifia il tipo di accesso e a quali risorse può accedere quell'utente.

La funzione di control access accede al database e determina quali accessi garantire.
La funzione di auditing monìtora e mantiene un record degli accessi degli utenti alle risorse del sistema.

Discretionary Access Control

In un approccio generale si costruisce una **access matrix** in cui si identificano, per ogni utente, le risorse a cui egli può avere accesso e con quali operazioni. Una matrice degli accessi è di solito *sparsa*.
Se si naviga la matrice per colonne si ottengono delle **access control list**, delle liste che stabiliscono per ogni risorsa quali utenti hanno i diritti di accesso.
Se si naviga la matrice per righe si ottengono dei cosiddetti **capability ticket**, ossia le specifiche operazioni e risorse a cui l'utente può accedere.

I due approcci hanno aspetti positivi e negativi opposti tra loro: le ACL permettono di determinate facilmente la lista di utenti con specifici diritti di accesso per una specifica risorsa, viceversa i capability ticket permettono di determinate l'insieme dei diritti di accesso di un singolo utente.

Role Based Access Control

Modello che definisce il concetto di **ruolo** che un utente può assumere all'interno del sistema al posto della sua identità individuale.
Gli utenti possono essere associati con diversi ruoli staticamente o dinamicamente, a seconda delle loro responsabilità.

La relazione tra utenti e ruoli è quindi *molti a molti*, così come la relazione ruoli-risorse.
Questo tipo di approccio è molto comodo in sistemi in cui l'ambiente cambia frequentemente (l'insieme degli utenti) e rende possibile l'assegnazione dinamica dei ruoli agli utenti.

In genere invece l'insieme di risorse dell'ambiente è pressoché statico con occasionali aggiunte e rimozioni.
In questo approccio si costruiscono due matrici:

- Una matrice che ha sulle righe gli utenti e sulle colonne i ruoli: ogni utente può avere più ruoli.
- Una seconda matrice che ha sulle righe i ruoli e sulle colonne le risorse: ogni entry stabilisce i diritti di accesso, ossia le operazioni di un ruolo per una data risorsa.

13. Configuration Management

Il **configuration management** è un processo che permette di controllare, monitorare e gestire i cambiamenti nell'evoluzione dei sistemi software.
I sistemi di configuration management automatizzano l'identificazione delle versioni, il loro storage e recupero.
Il configuration management include:

- **Identificazione degli oggetti di configurazione**: le componenti del sistema e gli artefatti prodotti, così come le loro versioni, sono identificati ed etichettati univocamente; gli sviluppatori creano versioni e nuovi elementi di configurazone man mano che il sistema evolve nel tempo.
- **Controllo dei cambiamenti**: i cambiamenti al sistema e i rilasci agli utenti sono controllati per assicurare la consistenza con i goal del progetto; il controllo dei cambuamenti può essere fatto dagli sviluppatori, dal management, da un control board, a seconda del livello di qualità richiesto.
- **Status accounting**: lo stato delle componenti individuali, dei work product e delle richieste di cambiamento viene registrato e ciò permette agli sviluppatori di distinguere le versioni e tracciare i cambiamenti e lo stato del progetto.
- **Auditing**: le versioni sono selezionate per la release e validate per assicurare completezza, consistenza e qualità; l'auditing è effettuato dal team di controllo qualità.

Altre attività spesso considerate parte del configuration management sono:

- **Build management**: molti sistemi di CM permettono il building automatico del sistema man mano che gli sviluppatori creano nuove versioni delle componenti.
- **Process management**: i progetti possono avere delle policy sulla creazione e documentazione delle versioni; una di queste potrebbe essere che solo codice sintatticamente corretto possa essere parte di una effettiva versione; infine questa attività include policy per notificare quando vengono create nuove versioni o quando una build fallisce.

Concetti fondamentali e definizioni di Configuration Management

Un **configuration item** è un artefatto o pezzo di software trattato come singola entità per gli scopi del configuration management. Una composizione di questi item è un **coniguration management aggregate**.

Una **versione** è lo stato di un configuration item (o aggregato) in un certo istante di tempo. Dato un aggregato, un insieme di versioni dei suoi configuration item è definito **configurazione**, ossia una versione di un aggregato.
Le versioni che coesistono sono dette **varianti**.

Una **promotion** è una versione resa disponibile ad altri sviluppatori del progetto.
Una **release** è una versione resa disponibile a clienti o utenti.

Una **baseline** è un versione di un configuration item che è stata formalmente revisionata e approvata dal management o dal cliente, e che può essere modificata solo attraverso una change request.
Una **change request** è un report formale di un utente/sviluppatore che richiede una modifica di un configuration item; un esempio di change request informale può essere, invece, una mail.

Una **libreria software** memorizza le versioni e fornisce dei servizi per tracciare lo stato dei cambiamenti.
Distinguiamo tre tipi di librerie:

- il **workspace** di uno sviluppatore, anche detto **dynamic library**: i cambiamenti sono controllati esclusivamente dal singolo sviluppatore.
- Una **master directory**, anche detta **controlled library**, è una libreria di *promotion*: i cambiamenti sono controllati e devono essere approvati.
- Un **repository**, conosciuto anche come **static library** è una libreria di *release*: le promotion devono rispettare certi criteri di qualità prima di diventare delle release.

Nel gergo comune anche il workspace locale di uno sviluppatore è chiamato repository.

Identificatori delle versioni

```
<version> :: = <configuration item name>.<major>.<minor>.<revision>
```

Le versioni sono identificate univocamente da un **version identifier** o **version number**.
In generale si usa uno schema a tre cifre per distinguere i vari tipi di cambiamenti: funzionali, piccoli miglioramenti e bug fix.
La cifra più a sinistra denota una **major version**, la seconda una **minor version**, la terza una revisione/correzione.
Per convenzione le versioni che precedono la 1.0.0 sono versioni per **alpha** o **beta** testing, di solito non destinate agli utenti.

Un **branch** identifica un path di sviluppo concorrente che richiede un configuration management indipendente.

I branch permettono lo sviluppo di diverse feature in maniera concorrente che possono essere successivamente unite in una singola versione attraverso un **merge**.
Possibile inserire una cifra per identificare il branch di sviluppo che precede il numero di revision.

Modelli di evoluzione di un configuration item

Due possibili rappresentazioni dell'evoluzione di un configuration item:

- **State-based view**: vede l'artefatto come un insieme di stati identificati dal numero di versione; si parla di **rappresentazione a snapshot**.
- **Changed-based view**: si considera una serie di cambiamenti tra due versioni successive, detti **delta**; i cambiamenti possono essere in termini di linee o paragrafi aggiunti o rimossi dal configuration item. Tutti i cambiamenti associati a una singola revisione sono raggruppati in un *change set*.

Strumenti di configuration management

Esistono numerosi strumenti e tecnologie per il CM: RCS, Subversion etc.
Nel nostro caso usiamo Git, un software di controllo versione basato su snapshot che consente di creare repository locali e interfacciarle con repository remote fornite, ad esempio dal servizio GitHub mediante un sistema client-server.

Un item in un repository può essere in 3 possibili stati:

- Nel workspace
- In Staging area: il file è stato modificato ed è preparato per il commit
- Commited: la modifica del file è stata confermata e commentata.

Gli item possono poi essere recuperati da remoto con operazioni di **checkout** come **fetch** e **pull** e inviati in remoto con operazioni di **push**.

GitHub gestisce le sequenze di commit come una lista: quando faccio il commit viene connesso con un puntatore al commit precedente.
Per lavorare in parallelo è possibile creare un branch: il master è un branch come tutti gli altri, il primo che viene creato.
Le **pull request** permettono di fare un **merge** tra due branch.

Project Planning

In riferimento alla parte di Project Plan del System Definition Document

Project Plan: definire lo sviluppo temporale del progetto; due possibili notazioni:

- **Analisi Pert/CPM**: più operativa, legata ad alcuni algoritmi
- **Diagrammi di Gantt**: diagrammi a barre che rappresentano graficamente le attivitàe i task di progetto in relazione al tempo

Diagrammi di Gantt

Per tracciare un diagramma di Gantt si parte da una **Work Breakdown Structure**: si definisce una vista ad albero delle attività che entreranno a far parte del progetto. Tipicamente l'albero avrà un'altezza di un paio di livelli.
Il progetto comprende una serie di attività che possono essere scomposte in dei task (o ulteriori attività e così via).

Non confondere la Work Breakdown Structure con le attività del ciclo di vita di un software: le attività sono proprio le attività concrete del progetto (es. scelta di determinate tecnologie, etc.).

Sulla griglia del tempo dello schema è possibile inserire una milestone, un momento di verifica, di check del progetto.

Gestione delle dipendenze

Sappiamo che in generale è possibile includere librerie in un progetto con delle referenze statiche.
Per automatizzare la gestione delle dipendenze esistono 2 tecnologie molto simili: Maven e Gradle.
Maven ci consente di risolvere 3 problemi: risoluzione delle dipendenze, avere un ciclo di build del sistema soddisfacente, e consente di interfacciarci con Github per vedere un primo esempio di pipeline di continuous integration.

Maven sfrutta un file chiamato *pom.xml* (Project object model) che contiene informazioni relative al progetto, alle dipendenze e a come fare il build.
Le dipendenze ci dicono che il progetto dipende da certe librerie (es. JUnit), e se una dipendenza non è presente viene scaricata, linkata al progetto e poi il progetto potrà essere buildato.

Articolo - Continuous Integration

Nella pratica comune tutto il codice di un software deve essere mantenuto in un repository.
In un ciclo di **continuous integration**, quando qualcuno fa un check in di codice in un repository, un sistema automatizzato esegue un insieme di comandi per verificare che i cambiamenti non abbiano portato dei problemi nel sistema.

Per i linguaggi compilati (come Java, C++, etc.) un **build** è uno step di compilazione seguito da un run di test suite. Per linguaggi dinamici come Python, JavaScript, si fa riferimento solo al run delle test suite.

L'ultimo step di un sistema di build automatico è la possibilità di effettuare un deploy automatizzato.

Strumenti per continuos integration: Github Action, CircleCI, Jenkins, etc.

11. Testing

Attività fondamentale del processo di software engineering e anche una delle più critiche e costose dell'intero ciclo di vita di un software.
Il testing è il processo che consente di trovare le differenze tra i comportamenti attesi specificati dai modelli del sistema e il comportamento effettivo del sistema implementato.

Concetti fondamentali del testing

Una **test component** è una parte del sistema che può essere isolata per il testing (un oggetto, un gruppo di oggetti, o uno o più sottosistemi).
Un **fault** (anche *bug* o *difetto*) è un errore di design o di programmazione che causa un comportamento anormale di una componente del sistema. L'obiettivo del testing è massimizzare la scoperta dei fault per permetterne la correzione e incrementare la reliability del sistema.
Un **failure** è una difformità tra la specifica e il comportamento effettivo.
Uno **stato di errore** è una manifestazione di un fault durante l'esecuzione; uno stato di errore è causato da uno o più fault e può portare il sistema ad una failure.
Una **correzione** è un cambiamento di una componente con lo scopo di riparare un fault; una correzione può però introdurre nuovi fault.

Possiamo definire il **testing** come il tentativo sistematico di trovare i fault in maniera pianificata nel software. Un'altra possibile definizione afferma che il testing consiste nel dimostrare che i fault non siano presenti.
Sebbene entrambe le visioni siano valide, nella maggior parte dei casi è impossibile dimostrare l'assenza di fault in sistemi software di una certa complessità.
L'approccio ideale consiste nell'invertire l'ottica: si prova a far emergere gli errori assumendo che, più errori trovo e correggo, maggiore è la probabilità che il programma si comporti in maniera coerente rispetto alle attese.

A differenza delle altre attività del processo di sviluppo il testing non è quindi un'attività costruttiva, bensì di **falsificazione**: possiamo dire che un test ha successo se ha portato alla luce degli errori.
Un sistema può essere rilasciato quando i test mostrano un ragionevole livello di confidenza tale da poter dire che il sistema si comporti in maniera corretta.

Diverse tecniche per incrementare la reliability del sistema:

- **Fault avoidance**: tecniche per rilevare gli errori staticamente senza esecuzione di codice; questo approccio consente di prevenire l'inserimento di errori prima ancora che il sistema venga rilasciato.
- **Fault detection**: tecniche come debugging e testing consentono di identificare e mettere in evidenza errori, sia prima che dopo il rilascio del sistema;
- **Fault tolerance**: tecniche che assumono che un sistema possa essere rilasciato anche in presenza di errori e failure risolvibili a runtime; in questo corso non vedremo questa area.

Il termine testing è utilizzato spesso con due significati: **testing dinamico**, che riguarda gli aspetti dinamici e di comportamento del sistema mediante l'esecuzione, oppure **testing statico**, una qualunque forma di verifica della qualità e correttezza di un sistema, o di una sua componente, che non preveda l'esecuzione. Si parla anche di **review** del sistema. Due possibili modi di fare review:

- Informale → **walkthrough**
- Formale → **inspection**: review formale in cui si effettua un check di interfacce e codice delle componenti del sistema e confronto con i requisiti; es. compilazione di un questionario con serie di domande

Il testing statico si può definire più correttamente come un insieme di metodologie statiche di V&V (**verification & validation**), per distinguerlo dall'effettivo testing, ossia quello dinamico e funzionale, relativo ai requisiti funzionali del sistema.

Un'attività spesso affiancata al testing è il **debugging**, la quale assume che i fault possano essere trovati a partire da un failure: una volta identificato uno stato di errore del sistema si va a determinare la causa. Distinguiamo due tipi di debugging:

- *Debugging di correzione* per trovare le deviazioni del sistema osservato dai requisiti funzionali
- *Performance debugging* per i requisiti non funzionali.

Se in un programma un errore non si manifesta in forma di failure allora esso non è rilevabile dal processo di testing.
Il testing ha quindi il compito di trasformare un bug in una failure, mentre il debugging serve per passare dalla failure al bug e capire dov'è l'errore per poterlo risolvere.

Un buon modello di test deve contenere casi di test che possano identificare dei fault. I test devono avere dei valori di input diversi, inclusi input invalidi e casi boundary, in modo da massimizzare la probabilità di rilevare fault.

Attività di testing

Test planning

Allocazione delle risorse e scheduling del testing; questa attività deve essere fatta nelle fasi iniziali di sviluppo per poter aver sufficiente tempo da dedicare al testing; un esempio è progettare i test case appena i modelli sono stati validati come stabili.

Usability testing

Ricerca dei fault nella UI

Unit testing

Ricerca di fault nelle componenti del sistema in relazione agli use case. Ogni componente è testata in isolamento rispetto alle altre.

Integration testing

Ricerca di fault testando più componenti combinate tra loro. Lo **structural testing** è invece l'integration testing che coinvolge tutte le componenti del sistema
Si verifica molto frequentemente che moduli differenti che singolarmente si comportano correttamente poi non abbiano comportamenti corretti insieme.
Anche se avrebbe poco senso fare integration testing prima dello unit testing, non è detto che vada fatto solo dopo aver completato lo unit testing.

System testing

Dopo aver verificato le singole parti e l'integrazione tra di esse, devo verificare se il sistema nel suo complesso si comporta in maniera conforme alle attese.
Testing di tutte le componenti combinate per identificare fault in relazione agli scenari, ai requirement e ai design goal; varie sottoattività:

- **Functional testing**: test dei requirement dal RAD e dallo user manual
- **Performance testing**: check dei nonfunctional requirement e design goal
- **Acceptance testing** e **Installation testing**: check del sistema rispetto agli accordi col cliente
- **Alpha e Beta testing**: rilascio di una versione prototipo del software a un gruppo ristretto di utenti interni all'organizzazione (alpha) o di utenti esterni (early access e beta) per ottenere feedback.

Un'ultima categoria di testing trasversale è il **regression testing** (o test di non regressione).
Ogni volta che il sistema cambia devo assicurarmi che il resto del sistema continui a comportarsi come prima, ossia che rimanga "complessivamente corretto".
Il test di regressione mira quindi a trovare le regressioni (o viceversa a dimostrare che non ci siano regressioni).

Nell'ultimo periodo si è sviluppato un nuovo approccio, detto di **retest all**: ogni notte o più volte al giorno, viene effettuato un testing del sistema.
Si parla di tecnologie di **test automation**.
Il **behaviour driven development** si basa molto su questi concetti.

Il processo di testing NON inizia quando ho scritto l'intero codice.
L'approccio *waterfall* prevedeva il testing come attività finale, al termine di tutte le altre attività di sviluppo.
Esistono molteplici casi però dove il testing viene anticipato: numerosi approcci di ingegneria del software, per esempio **test driven development** vedono il testing come guida dell'intera fase di sviluppo.

L'idea di fondo è di progettare i test case per i vari behavior e poi cominciare ad implementarli, in modo che man mano che lo implemento posso avere una sorta di stato di avanzamento.

Test case

Un test case è definito attraverso:

- *Nome*: seguendo un'euristica si usano nomi che suggeriscono la componente testata
- *Location*: attributo che descrive dove il test case può essere trovato (es. path o URL dell'eseguibile o del programma di test e input)
- Un insieme di *input*
- *Oracolo*: risultati attesi
- *Log*: un report dei comportamenti osservati in relazione a quelli attesi al termine dell'esecuzione di vari test

Test stub e test driver

Stub e driver sono utili per sostituire parti mancanti del sistema e rendere possibile il testing di componenti che dipendono da altre componenti in maniera isolata:

- Un **test stub** è una implementazione parziale di componenti dai quali le componenti testate dipendono. Esso simula una componente chiamata dalla componente testata; deve fornire la stessa API e restituire valori degli stessi tipi della componente simulata.
- Un **test driver** è una implementazione parziale di una componente che dipende da un test component. Esso simula la parte del sistema che chiama la componente testata. Un test driver passa gli input di test identificati nell'analisi del test case alla componente e mostra i risultati.

Stub e driver permettono quindi di isolare le componenti da testare dal resto del sistema.

Articolo Sommerville - Software Testing

Il processo di testing ha due obiettivi distinti:

1. Dimostrare che il software rispetta i requisiti
2. Scoprire situazioni nelle quali i comportamenti del software sono scorretti o non conformi alle specifiche

Il primo dei due obiettivi è il validation testing, che permette di verificare che il sistema si comporti correttamente usando dei test case che riflettono il tipico utilizzo del sistema.
Il secondo obiettivo è il defect testing, progettare test case in maniera specifica per la ricerca dei difetti.

Il testing non può mai dimostrare che il software sia esente da difetti o che si comporti sempre come da specifica in ogni circostanza.

L'obiettivo definitivo delle attività di V&V è di stabilire con una certa confidenza che il sistema è adatto al suo scopo. Tale livello di confidenza dipende dallo scopo del sistema, dalle aspettative degli utenti e dall'ambiente di mercato.

Le tecniche statiche V&V differiscono dal testing dinamico perchè non si basano sull'esecuzione del software per verificarlo. L'inspection si concentra principalmente sul codice sorgente, ma anche su altri modelli.
Tre vantaggi dell'inspection rispetto al testing:

1. Durante il testing gli errori possono mascherare altri errori; l'inspection è un processo statico e, pertanto, non è necessario preoccuparsi delle interazioni tra gli errori; una singola inspection può permettere di scoprire molti errori.

2. Possibile inspection anche su versioni incomplete del sistema senza necessità di sviluppare parti necessarie per le componenti da testare
3. L'inspection permette di cercare sia i bug, ma anche altri difetti come le inefficienze, algoritmi inappropriati, un pessimo stile di programmazione, etc.

Tuttavia la software inspection non può rimpiazzare il testing perchè non può scoprire tutti quei difetti che sorgono a causa di interazioni tra diverse parti di un programma, problemi temporali e problemi di performance.

Tipicamente un sistema software commerciale ha 3 stage di testing:

1. *Development testing*: system designer e programmatori testano il software alla ricerca di bug e difetti
2. *Release testing*: un testing team separato testa una versione completa del sistema prima del rilascio per controllare che i requisiti siano rispettati
3. *User testing*: gli utenti o potenziali tali testano il sistema nel proprio ambiente; acceptance testing è un tipo di user testing dove il cliente testa il sistema e decide se è accettabile per la consegna.

Il testing include un mix di attività di testing manuale e automatizzato.

Development testing

Avviene durante lo sviluppo, a tre livelli di granularità:

- **Unit testing**: testing di singole unità di programma
- **Component testing**: (integration testing) testing di integrazione tra varie componenti
- **System testing**

Il development testing è prima di tutto un processo di defect testing che mira alla scoperta dei *bug*.
Molto importante il **debugging**, il processo che permette di localizzare i problemi di codice ed effettuare dei fix.

Unit testing

Testing di singole componenti del sistema (metodi, classi, etc.).
Quando si testano classi è necessario progettare dei test che forniscano un coverage di tutte le feature:

- Testare tutte le operazioni associate agli oggetti della classe
- Check dei valori degli attributi
- Portare gli oggetti di quella classe in tutti i possibili stati (simulare tutti gli eventi che causano cambiamenti di stato)

Unit testing può essere automatizzato con dei framework come JUnit che forniscono classi di test generiche che è possibile estendere per creare specifici test case.
Possibile eseguire tutti i test e ottenere dei report sul successo o fallimento dei test.

Un test automatizzato ha tre parti:

- Una parte di setup: si inizializza il sistema specificando input e output attesi
- Una chiamata, in cui si chiama l'oggetto o metodo da testare
- Un asserzione, in cui si confronta il risultato della chiamata con il risultato atteso; se l'asserzione è valutata true allora il test ha successo, altrimenti l test fallisce

Importante progettare test case efficaci che permettano di scoprire difetti.
Due possibili strategie:

- *Partition testing*: identificare gruppi di input con caratteristiche comuni per raggruppare i test

- *Guideline-based testing*: usare delle linee guida che riflettono esperienze pregresse su errori che i programmatori commettono spesso nello sviluppo

Il partition testing permette quindi di identificare le classi di equivalenza di input e output per evitare di progettare più volte casi di test equivalenti.
Una volta identificate le classi di equivalenza una buona regola è selezionare test case sui boundary di queste classi, oltre che test case posti nel mezzo; in questo modo è possibile sia testare valori tipici che valori atipici.

Possibili linee guida per il secondo approccio:

- Scegliere input che forzano il sistema a generare tutti i messaggi di errore
- Progettare input che causano input buffer e overflow
- Ripetere lo stesso input o serie di input più volte
- Forzare la generazione di output non validi
- Forzare risultati molto grandi o piccoli

Component testing

Quando si testa l'integrazione tra varie componenti è possibile incontrare degli *interface error* di vari tipi:

- *Interface misuse*: una componente chiama altre componenti e commette errori nell'uso dell'interfaccia (parametri con tipi sbagliati o nell'ordine sbagliato, numero di parametri scorretto)
- *Interface misunderstanding*: una componente chiamante non rispetta le specifiche di una interfaccia facendo assunzioni sul suo funzionamento; es. un metodo di Binary Search che riceve un array non ordinato → la ricerca fallisce
- *Timing error*: nei sistemi real-time con memoria condivisa è possibile che accadano errori di temporizzazione tra produttori e consumatori

System testing

Si crea una versione del sistema e si testano le componenti integrate. Due differenze principali dal component testing:

- Le componenti sviluppate separatamente dalle altre possono essere integrate con componenti sviluppate in seguito
- Possibilità di integrare anche componenti sviluppate da membri/team differenti; in alcune aziende può esserci un team di testing separato dai designer e programmatori

Il system testing deve concentrarsi sull'interazione tra le componenti e gli oggetti che creano l'intero sistema; questo test di interazione serve per scoprire quei bug che possono rivelarsi solo una volta che le componenti interagiscono tra loro.
Un approccio ideale per il system testing è il testing basato sugli use-case (e sequence diagram).

Per molti sistemi complessi è difficile capire quando concludere il testing: in generale si hanno delle policy aziendali per scegliere un certo sottoinsieme di use case da testare; in alternativa si testano tutti gli use case relativi all'esperienza d'uso generale del sistema.

Il system testing automatizzato è di solito molto più difficile rispetto allo unit testing (e component testing) perchè il sistema può generare output che non sono facilmente predicibili.

Test-driven development

Si tratta di un approccio di sviluppo in cui si fondono le attività di testing e di programmazione. Lo sviluppo del codice è incrementale, in parallelo al testing.

Tale approccio è stato introdotto come parte dei metodi agili (come l'Extreme Programming).

Passi del processo TDD:

1. Identificazione di una funzionalità da implementare
2. Progettazione e implementazione di un test automatizzato per tale funzionalità
3. Esecuzione del test e degli altri test implementati prima di implementare la funzionalità: il test fallirà
4. Implementazione della funzionalità e riesecuzione del test
5. Una volta che tutti i test vengono eseguiti con successo si passa alla funzionalità successiva

Oltre a migliorare la comprensione di un problema, altri benefici del TDD:

- *Code coverage*: il codice viene testato fin da subito e ciò permette di scoprire molto prima eventuali difetti del processo di sviluppo
- *Regression testing*: la test suite è sviluppata incrementalmente ed è sempre possibile eseguire test di regressione per verificare che dei cambiamenti non abbiano introdotto nuovi bug
- *Debugging semplificato*: quando un test fallisce è semplice capire dov'è localizzato il problema, anche senza appositi strumenti di debugging
- *System documentation*: i test sono anche una forma di documentazione che descrive cosa il codice dovrebbe fare

L'approccio TDD ha riscontrato un buon successo soprattutto per progetti di medio-piccole dimensioni.

Release testing

Processo di testing di una release del sistema non destinata per il rilascio al cliente.
Si differenzia dal system testing per 2 importanti motivi:

- il release testing è effettuato da un team separato rispetto a quello di sviluppo
- Mentre il system testing è focalizzato sulla scoperta di bug e difetti, il release testing serve per verificare che il sistema rispetti i requisiti e sia ben utilizzabile nell'ambiente in cui è destinato

Il release testing è principalmente un'attività di testing black-box, cioè si tiene conto solo degli input e dei relativi output del sistema, non del codice.

Requirements-based testing

Un principio generale nella scelta dei requisiti ingegneristici è che essi siano testabili.
Il requirement-based testing è quindi un approccio sistematico nel design degli use-case in cui, per ogni caso d'uso, si deriva un insieme di test.

Scenario testing

Approccio di release testing in cui si prendono in considerazione alcuni scenari per lo sviluppo di casi di test del sistema. Tali scenari devono essere realistici.
Possibile il riutilizzo di eventuali scenari elicitati durante il processo di *requirement elicitation* anche per il testing.

Performance Testing

Una volta che il sistema è completamente integrato è possibile testare anche requisiti non funzionali, come performance e reliability.
I performance test devono essere progettati per assicurare che il sistema possa processare il carico ipotizzato.

Si esegue una serie di test in cui si incrementa progressivamente il carico fino a che le prestazioni del sistema non diventano inaccettabili.
Si tratta quindi di una sorta di *stress testing*.

Tale tipo di testing è particolarmente rilevante per sistemi distribuiti, i quali mostrano spesso una severa degradazione delle prestazioni quando sono sottoposti a un grande carico.

User testing

Coinvolge utenti o clienti; si tratta di una tipologia di testing molto importante perchè per gli sviluppatori è praticamente impossibile replicare l'ambiente di utilizzo del sistema.
Distinguiamo tre tipi diversi di user testing:

- **Alpha testing**: utenti collaborano col team di sviluppo per testare il software in ambiente di sviluppo.
- **Beta testing**: una release del software viene resa disponibile ad utenti per trovare possibili problemi.
- **Acceptance testing**: i clienti testano il sistema per decidere se accettarlo ed eventualmente permettere il deployment.

Il beta testing è particolarmente utilizzato per software che sono destinari a diversi ambienti di utilizzo, in quanto è impossibile per gli sviluppatori conoscere e replicarli tutti.
Può essere usato anche come forma di marketing e per addestrare i clienti all'utilizzo del sistema.

L'acceptance testing si compone di 6 fasi principali:

1. Definizione dei criteri di accettazione: questa fase avviene di solito prima della firma del contratto;
2. Pianificazione dell'acceptance testing: decisione di risorse, tempo e budget per l'acceptance testing
3. Derivazione dei test: progettazione dei test, che mirino a testare sia i requisiti funzionali che quelli non funzionali
4. Esecuzione dei test: i test vengono eseguiti nell'ambiente reale di utilizzo del sistema
5. Negoziazione: difficilmente tutti gli acceptance test avranno successo, per questo svilupatori e clienti negoziano per decidere se il sistema può essere utilizzato o necessita di ulteriore lavoro
6. Rifiuto o accettazione del sistema: in un meeting tra sviluppatori e clienti si decide l'eventuale accettazione definitiva del sistema, con conseguente ripetizione dell'acceptance testing

I metodi agili (in particolare l'Extreme Programming) prevedono il coinvolgimento degli utenti nel processo di sviluppo, i quali hanno un ruolo chiave per il design dei test del sistema.

Capitolo 5 Robillard - Unit Testing

Unit testing è un approccio di testing che consente di testare una piccola parte di codice in isolamento.

Uno *unit test* consiste in una o più esecuzioni di una **unit under test** (UUT), con dati di input e un confronto tra il risultato dell'esecuzione e un **oracolo**.
Una UUT è un qualsiasi pezzo di codice che si desidera isolare.
L'oracolo è il risultato atteso dell'esecuzione di una UUT.

Regression testing

Testing di codice già testato per verificare che sia ancora corretto e rilevare eventuali nuovi bug causati da dei cambiamenti.

Importante: lo unit testing non può verificare la correttezza del codice. Quando un test passa possiamo sapere soltanto che quell'esecuzione rispetta il comportamento atteso.

JUnit - Unit Testing Framework

JUnit è un framework che permette di automatizzare lo unit testing. Tali framework includono tool per collezionare, fare il run e il display dei test, ma anche costrutti per scriverli in maniera strutturata ed efficiente.

L'annotazione **@Test** permette di indicare che un metodo deve essere eseguito come unit test.
Gli **assert method** sono metodi statici che verificano un predicato e, nel caso sia falso, riportano un *test failure*.
Il framework JUnit include una componente UI chiamata *test runner* che rileva ed esegue automaticamente i tet ed effettua un report sui test passati e falliti.

Una collezione di test di un progetto è chiamata **test suite**.
JUnit fornisce l'annotazione **@Suite** per eseguire insieme più classi di test.

Esistono differenti approcci per costruire suite di unit test, ma l'approccio di base in Java è di avere una test class per classe di progetto e all'interno di essa metodi che testano scenari che coinvolgono una data classe.

Metaprogramming - Reflection

Il *metaprogramming* è una tecnica che consente di scrivere codice che non opera su dati che rappresentano entità del mondo reale, ma su altri pezzi di codice.
In Java il metaprogramming è chiamato **reflection**.

Introspection

Il task più semplice del metaprogramming consiste nell'ottenimento di un riferimento ad un oggetto che rappresenta un pezzo di codice e ottenere informazioni da esso.

Un esempio è l'introspection di un oggetto di una qualsiasi classe e ottenere informazioni su metodi, variabili d'istanza, eventuali interfacce implementate e/o classi ereditate, eccezioni lanciate dai metodi, etc.

Ognuna di queste informazioni è modellata come un oggetto di una classe Java specifica (Method, Constructor, Field, etc.).

Program Manipulation

Permette di cambiare l'accessibilità di membri di una classe, settare valori dei campi, creare nuove istanze di oggetti e invocare metodi.

Program Metadata

Il metaprogramming non consente soltanto di operare su dati che sono elementi di codice ma anche sui metadati di questi elementi.

In Java è possibile usare le annotazioni per definire delle informazioni addizionali (meta-informazioni) su elementi di codice, leggibili dal compilatore.
In Java è possibile creare un *annotation type* in modo simile ad una interface.

Come strutturare i test

Esistono dei principi basilari sulla progettazione di unit test:

- **Velocità**: gli unit test devono poter essere eseguiti frequentemente e, pertanto, devono avere esecuzioni molto veloci; evitare lunghe operazioni di I/O e accessi in rete (lasciare il testing di queste funzionalità ad altri tipi di test come *acceptance test* e *integration test*).
- **Indipendenza**: ogni unit test deve essere eseguito in isolamento, non deve dipendre da altri test; molti framework infatti non garantiscono che l'ordine di esecuzione dei test sia predicibile
- **Ripetibilità**: l'esecuzione ripetuta degli unit test deve produrre gli stessi risultati in diversi ambienti
- **Focus**: i test devono verificare solo una porzione di codice che esegue un comportamento ragionevolmente piccolo; i test devono avere quindi un focus su un solo aspetto di un UUT
- **Leggibilità**: la struttura e il coding style del test deve rendere semplice l'identificazione dei componenti del test (UUT, input e oracolo).

Test e condizioni d'eccezione

Quando si usa il *design by contract* non ha senso testare codice con input che non rispettano le precondizioni, perchè porterebbero ad un comportamento non specificato.

JUnit permette di specificare una possibile eccezione nell'annotazione *Test* usando la proprietà *expected*.
In questo modo il test fallirà automaticamente se l'esecuzione del metodo viene completata senza sollevare l'eccezione del tipo specificato.

Incapsulamento e Unit Testing

Come testare metodi privati? Due possibili approcci:

- Testare i metodi privati indirettamente attraverso l'esecuzione dei metodi che li invocano
- Non considerare nei test tutto ciò che ha private come modificatore d'accesso

Nei casi in cui sia preferibile testare un metodo privato è possibile bypassare la restrizione d'accesso utilizzando il metaprogramming.

Testing con Stub

In alcuni casi può risultare difficile testare pezzi di codice in isolamento, ad esempio quando alcune componenti attivano o dipendono dall'esecuzione di altre componenti.

Uno **stub** è una versione semplificata di un oggetto (componente) che simula un comportamento, a supporto di una UUT da testare.

Strategie di testing

Una strategia di testing è un approccio metodologico sistematico con il quale scegliamo gli input da utilizzare per fare i nostri casi di test.
Un insieme di casi di test è una **test suite** o **piano del testing**.

Come si progetta una test suite?
Mentalità di tipo **adversarial approach**: mettersi nelle condizioni di trovare gli errori.
Strategie di testing orizzontale (top-down, bottom-up, sandwich, etc.) sono approcci e diciture obsolete, non consone nel nostro caso.

Test Coverage

Possiamo distinguere i test case in due grandi categorie.

- **Blackbox testing - Testing funzionale**: ci si concentra sui comportamenti di I/O della componente testata, senza tener conto della struttura interna (il codice), ma soltanto delle

specifiche (comportamenti attesi); Si preparano i casi di test leggendo il comportamento atteso dalla componente che si vuole testare (nel caso di unit testing).
- **Whitebox test - Testing strutturale**: il focus è sulla struttura interna della componente testata; un whitebox test permette di testare ogni stato del dynamic model di un oggetto e le interazioni tra gli oggetti, a partire da particolari condizioni iniziali; questa strategia tiene quindi conto dell'implementazione dell'UUT.

Lo **unit testing** è una metodologia che combina entrambe le strategie: blackbox testing per le funzionalità di una componente, whitebox testing per il testing di struttura e aspetti dinamici.
Il black box è, però, applicabile a qualsiasi livello di testing, non solo unit testing.

Un classico metodo per determinare cosa testare è basato sul concetto di **coverage**, ossia copertura del codice. Possibile usare delle metriche di test coverage come metriche informali utili per determinare la quantità di codice che viene verificata durante l'esecuzione dei test.
Distinguiamo diverse strategie di coverage:

- **Statement Coverage**: usare come metrica la percentuale di statement (istruzioni) testate da un test o da una test suite rispetto al totale codice di interesse; tale strategia è poco robusta perchè non assicura di aver eseguito tutte le possibili combinazioni di istruzioni del codice (eventuale presenza di cicli, ramificazioni if-else, etc.); il codice viene rappresentato come un grafo e l'obiettivo è cercare di attraversare tutti i nodi del grafo.
- **Branch coverage**: si cerca di coprire tutti i branch del codice almeno una volta; si utilizza come metrica il numero di diramazioni (branch) attraversati dai test rispetto al totale; tale strategia è più robusta rispetto allo statement coverage perchè a parità di copertura permette di testare un maggior numero di diverse esecuzioni del codice testato; il branch coverage è la strategia più utilizzata anche perchè è ben supportata dagli strumenti di testing ed è semplice da interpretare.
- **Path coverage**: copertura di tutti i cammini possibili del codice; in questo caso il numero di test case aumenta. Strategia molto robusta ma non facilmente applicabile, poichè in molti casi (es. in presenza di cicli) il numero di path di un pezzo di codice potrebbe non essere finito; la metrica di path coverage è pertanto una metrica teorica, non è possibile stabilire una vera metrica di copertura dei cammini.

Fare path coverage è spesso molto complesso, soprattutto in presenza di cicli. Quando non riusciamo a risolvere questo problema possiamo semplificarlo utilizzando le classi di equivalenza: ad esempio in presenza di un ciclo, è possibile considerare un caso che comprenda l'esecuzione del ciclo 1 o più volte e un caso in cui non viene eseguito.
Si definiscono quindi le classi di equivalenza ei sceglie un solo caso tra tutti quelli equivalenti.

Mutation Testing - Testing di mutazione

Tecnica white box che consiste nell'inserire degli errori di proposito nell'UUT per verificare l'esistenza di test case che possano rilevare tali errori. In questo modo è possibile capire l'eventuale necessità di progettare nuovi casi di test o valutare la qualità e correttezza di quelli esistenti.

Articolo - Misura di complessità di McCabe

La misura di complessità proposta si basa sulla misura e il controllo del numero di path di un programma. Questo approccio richiede la costruzione di un **grafo del flusso di controllo** del codice, ossia un grafo con un unico nodo di entrata e un unico nodo di uscita; ogni nodo è un pezzo di codice di programma con istruzioni sequenziali, mentre un arco è un branch.
Definizione: In un programma strutturato la **complessità ciclomatica** *v(G)* è definita come:

$$v(G) = e - n + 2p$$

Dove e è il numero di archi, n il numero di nodi e p è il numero di componenti connesse.
Alcune proprietà della complessità ciclomatica:

- $v(G) \geq 1$
- In un grafo fortemente connesso $v(G)$ è uguale al massimo numero di cammini linearmente indipendenti, è la dimensione di una base.
- Inserire o eliminare statement funzionali al grafo *G* non ha alcun effetto su v(G)
- $v(G) = 1$ se e solo se il grafo G ha un solo path (il codice non contiene if-else o cicli For)
- Inserire un nuovo arco in *G* incrementa $v(G)$ di un'unità
- $v(G)$ dipende solo dalla struttura decisionale di *G*

Secondo McCabe è buona norma per gli sviluppatori usare la complessità ciclomatica come guida nello sviluppo dei moduli software: qualora si riscontri una complessità maggiore o uguale a 10 per i moduli, sarebbe opportuno considerare una rimodularizzazione del codice per diminuire tale complessità.

Un'applicazione della metrica è nella determinazione del numero di casi di test necessari per il coverage di un particolare modulo, grazie a due proprietà:

- *v(G)* è un limite superiore al numero di test necessari per raggiungere uno statement coverage
- Limite inferiore per il numero di cammini all'interno del grafo di controllo del flusso; assumendo che ogni test sia un cammino, il numero di casi necessari per raggiungere un path coverage totale è uguale al numero di cammini che possono essere effettivamente presi. Tuttavia, talvolta, alcuni cammini sono impossibili (o infiniti), quindi questo numero può essere inferiore a *v(G)*

Ne emerge la seguente relazione:

$$coverage \leq complessita' \; ciclomatica \leq numero \; di \; cammini$$

1. Introduzione al Software Design

Il processo di design di un software è sicuramente un'attività di decision making: consiste nel prendere decisioni nella scelta di soluzioni all'interno di uno spazio di soluzioni.

Immaginiamo di muoverci in uno spazio n-dimensionale in cui gli obiettivi di design costituiscono le n dimensioni. All'interno di questo spazio sono presenti le varie soluzioni, ossia le diverse possibili implementazioni del sistema.
Possiamo dividere tutte le soluzion in 2 grandi aree:

- L'area delle soluzioni ammissibili
- L'area delle soluzioni desiderabili

L'attività di software design ha il compito di individuare quelle soluzioni che appartengono ad entrambe le aree.

Cattura delle decisioni e conoscenze di design (Design Know-How)

Un design è quindi un insieme di decisioni, ognuna delle quali è il risultato di un processo di ricerca attraverso uno spazio di design per un particolare problema.
La conoscenza di design, però, può essere catturata ed esplicitata in vari modi:

- **Codice sorgente**: molte decisioni di design possono essere codificate direttamente nel codice sorgente; un codice ben scritto, documentato, commentato e strutturato, è molto significativo e rende possibile esplicitare le decisioni di design; il codice da solo non è però un buon substrato per il razionale delle decisioni di design, per il quale è possibile usare i commenti.
- **Documenti di design** (inclusi diagrammi general-purpose): documenti di vari tipi, standardizzati, post su blog, etc. ; tali documenti possono includere diagrammi.

- **Piattaforme di discussione e sistemi di controllo-versione**: le informazioni di design possono essere registrate nei commenti e mailing list degli strumenti di sviluppo software, come sistemi di issue management e sistemi di controllo-versione.
- **Modelli specializzati**: non sono nostro oggetto di studio; modelli formali che possono essere automaticamente convertiti in codice scritto in un linguaggio di programmazione; tale approccio prende il nome di *model-driven development* (o generative programming).

Design Pattern

L'idea di design pattern è stata introdotta dalla cosiddetta *Gang of Four* nel libro *Design Patterns: Elements of Reusable Object-Oriented Software*. Il libro descrive 23 pattern per la soluzione di problemi di software design comuni, ma da allora ne sono stati documentati molti altri.

I design pattern sono infatti un modo ricorrente di risolvere uno specifico problema che si è dimostrato solido e che può essere catturato in uno schema e riapplicato nel tempo: possiamo vederli come *template di soluzione*.

Così come le soluzioni sono ripetibili in diverse applicazioni, anche gli errori possono ricorrere frequentemente: concetto di **design antipattern**.
Essi prendono anche il nome di *code smell* o *bad smell*.

2. Incapsulamento e Information Hiding

Nel software design incapsuliamo sia dati che elementi di computazione per limitare i punti di contatto tra diverse parti di codice.
L'incapsulamento ha diversi benefici: rende più semplice la comprensione di un pezzo di codice in isolamento e, isolando una parte dal resto, rende più semplice cambiare pezzi di codice senza avere impatto sulle altre componenti.

Il concetto di incapsulamento è particolarmente collegato al concetto di **information hiding**: l'incapsulamento di strutture dati è utile a rivelare il minimo numero di informazioni possibili sulla logica interna di tali strutture.
Un tipico esempio di applicazione di information hiding sono gli **Abstract Data Type** (**ADT**), che forniscono interfacce minimali ai programmi client che non devono conoscere l'implementazione interna per poterli utilizzare.

Codifica delle astrazioni come tipi

Immaginiamo di voler modellare un mazzo di 52 carte come una semplice collezione ordinata di carte da gioco.
Per farlo dobbiamo capire come codificare il concetto di carta, la quale è caratterizzata da numeri e semi. Abbiamo varie soluzioni:

- Rappresentare le carte come numeri interi compresi tra 0 e 51 (usando una convenzione per i semi)
- Rappresentare le carte come coppie di valori, il primo che rappresenta il valore e il secondo per il seme (o viceversa);
- Un'altra possibilità è usare 6 valori booleani: si tratta di una decisione possibile, ma decisamente non accettabile

Ognuna di queste soluzioni ha numerosi problemi:

- Prima di tutto la rappresentazione di una carta non mappa un corrispondente concetto di dominio; per facilitare la comprensione ed evitare errori di programmazione la rappresentazione dei valori

dovrebbe idealmente essere simile al concetto che deve rappresentare; usare un generico tipo *int* non è la scelta giusta per una carta da gioco.
- In secondo luogo la rappresentazione della carta è in queste soluzioni accoppiata alla sua implementazione: se decisiamo di usare degli interi ogni pezzo di codice che deve memorizzare un valore di una carta dovrà far riferimento ad un intero; cambiare questa codifica significherebbe cambiare ogni singola locazione in cui è presente una variabile intera usata per memorizzare una carta e tutto il codice che lavora con le carte come interi.
- Terzo, è molto semplice corrompere una variabile che memorizza un valore che rappresenta una carta: il tipo *int* ha un dominio di valori molto più grande rispetto al concetto di Carta da gioco e pertanto esistono innumerevoli valori che rappresenterebbero informazioni non valide.

In generale si dimostra che i designer di software tendono a mostrare spesso un antipattern chiamato **primitive type obsession**: non ragionano sulle informazioni salienti ma danno per scontato i livelli più bassi e li mappano automaticamente su tipi primitivi.

Per applicare il principio di information hiding organizziamo il codice per nascondere le decisioni di design su come rappresentare una carta mediante una **interface**.

Un modo molto semplice e più corretto di rappresentare una carta da gioco consiste nel considerare valore e seme delle carte come dei tipi **enumeration**, che consentono di restringere il dominio dei valori in un dominio discreto di valori costanti.

```
class Card
{
        Suit aSuit;
        Rank aRank;
}
```

Dove Suit e Rank sono due tipi *enum*.

A questo punto, definita la classe Card possiamo progettare il concetto di Deck come collezione di carte, ad esempio come List di Card.
Tale possibilità presenta degli svantaggi:

- Una lista di carte non è particolarmente vicina al concetto di mazzo: di solito le carte sono considerate in "pila".
- Usare una lista renderebbe la rappresentazione del mazzo troppo legata alla sua implementazione: cambiare la lista, ad esempio, con un array, necessiterebbe di cambiare buona parte del codice
- Struttura facile da corrompere: un mazzo di carte può contenere al più 52 carte, senza possibilità di duplicati, mentre la lista rende possibile aggiungere un numero qualsiasi di carte, inclusi duplicati.

Visibilità e scope delle informazioni

Una volta decisi i tipi delle informazioni, in molti linguaggi, tra cui Java, vi è la possibilità di specificare la visibilità e lo scope usando degli **specificatori di accesso**.

Per applicare l'idea di incapsulamento ed information hiding è possibile restringere l'accesso ai campi di una classe in maniera simile alla visibilità delle variabili locali di una funzione.
In molti linguaggi si parla di *scope* per indicare la regione in cui esiste una variabile.

In Java è possibile controllare la visibilità di classi e membri delle classi (campi e metodi) usando tali modificatori di accesso: i membri **public** sono visibili e accessibili ovunque nel codice, anche all'esterno della classe in cui sono dichiarati; i membri **private** sono visibili solo all'interno dello scope della classe.

Un principio generale per avere un buon incapsulamento è avere variabili di istanza di una classe private e metodi pubblici che rivelano il meno possibile sulla propria implementazione interna.

Reference escaping

Usare campi di tipo *private* non basta come misura di protezione della struttura interna di una classe.

Immaginiamo di avere un'istanza di Deck e mettere a disposizione un metodo get che permetta di ottenere la lista delle sue carte.
Questa soluzione permette, però, di accedere a una referenza alla lista di carte interna, permettendo quindi un **escape** al di fuori dello scope della classe e garantendo l'accesso agli elementi interni anche al di fuori di essa.

Pertanto dichiarare dei campi private non basta per garantire l'incapsulamento ma è necessario prevenire l'escape delle referenze alle strutture interne al di fuori dello scope della classe.
Esistono tre modi principali in cui è possibile che un riferimento a una struttura privata possa sfuggire allo scope della classe:

- **Restituzione di una referenza ad un oggetto interno**: tipicamente attraverso metodi **get** che restituiscono membri privati
- **Memorizzazione di una referenza esterna internamente**: usare una referenza ad un oggetto esterno che inizializza lo stato interno di un altro oggetto; ad esempio se forniamo metodi **set** è possibile "corrompere" lo stato interno della classe inserendo valori non consentiti; lo stesso può essere fatto con i metodi costruttori
- **Leaking delle referenze attraverso strutture condivise**, che rende ancora più difficile l'identificazione dell'escaping delle referenze.

Immutabilità

Una feature fondamentale utile per l'incapsulamento è rendere impossibile la modifica dello stato interno di un oggetto se non attraverso i suoi metodi, ma abbiamo visto che anche questo non è esente dal problema di escaping delle referenze.

La soluzione potrebbe essere l'utilizzo di tipi **immutabili**, come il tipo String.
In Java le stringhe sono costanti, ossia il loro valore non può essere modificato dopo la creazione, pertanto l'escaping delle referenze sulle stringhe non ha alcun impatto poiché non è possibile cambiare il valore usando una referenza.
Lo stesso vale per qualsiasi oggetto immutabile.

Gli oggetti possono essere resi immutabili se la propria classe non fornisce alcun modo per cambiare lo stato interno in seguito all'inizializzazione.
Sfortunatamente non esiste alcun modo per garantirlo, ed è necessario progettare la classe attentamente in modo da assicurare l'immutabilità e prevenire ogni modifica.

L'ideale sarebbe mettere a disposizione un'interfaccia che includa metodi che restituiscano soltanto riferimenti ad oggetti immutabili; in alternativa, se ciò non è possibile, la soluzione sarebbe restituire delle **copie** dei riferimenti ai campi della classe.

Una tecnica comune consiste nel progettare dei **costruttori copia**, costruttori che prendono come argomento un oggetto della classe stessa e ne copiano i valori dei campi.

In Java esistono ulteriori meccanismi per la copia di oggetti, come **cloning**, **metaprogramming** e **serialization**.

In sintesi:

- Scegliere bene i tipi evitando di usare tipi molto più ampi dell'insieme di valori che si vuole rappresentare (spesso si possono usare le enum)
- Non basta usare in maniera cieca le feature di un linguaggio per garantire l'esempio concettuale; ad esempio fare attenzione a come si implementa l'information hiding: creare variabili private non basta.
- Non esiste una soluzione data.

Design by Contract

Uno dei benefici dell'incapsulamento consiste nell'impedire ad un client di corrompere il valore di una variabile.
Il problema sorge però in presenza di valori **null**.

Il **design by contract** è un'idea che consiste nel seguire un approccio per la specifica di interfacce: non sempre le signature dei metodi specificano abbastanza informazioni senza lasciare ambiguità, ma è necessario eliminare tali ambiguità documentando il range dei valori possibili ed accettabili.

L'idea di base è di fornire una sorta di *contratto* tra client (programma chiamante) e *server* (metodo chiamato) mediante un insieme di **precondizioni** e **postcondizioni**.

Una **precondizione** è un predicato che deve essere vero all'inizio dell'esecuzione di un metodo; una **postcondizione** è invece un predicato che deve essere vero al termine dell'esecuzione del metodo stesso.
Tali predicati coinvolgono tipicamente i valori di argomenti, lo stato degli oggetti, etc.

Date precondizioni e postcondizioni, il contratto sta nel fatto che il chiamante deve rispettare le precondizioni: se ciò non accade, il comportamento del metodo non è definito.

Java fornisce i tag *@pre* e @post per stabilire pre e postcondizioni.
Un'alternativa è l'utilizzo delle **asserzioni** con lo statement **assert**: si tratta di predicati che possono sollevare degli *AssertionError* qualora vengano valutati falsi.

Le asserzioni sono disabilitate di default ma è necessario aggiungere -ea come parametro per la Virtual Machine per specificare di controllare l'assertion altrimenti resta solo un commento.

Se correttamente implementato il design by contract permette di evitare il cosiddetto **defensive programming**, in cui si va a effettuare numerosi check all'interno del codice (es. le null reference).
Inoltre questa tecnica rende facile capire le responsabilità di errore durante il debugging: se fallisce una precondizione, il problema è nel metodo chiamante, viceversa se fallisce una postcondizione è "colpa" del metodo chiamato.

Possibili quindi numerose strategie quando si implementa un metodo:

- Strategia defensive: controllo tutti i possibili parametri, catturo le eventuali problematiche e sollevo un'eccezione (checked); questo costringe tutti i chiamanti a dover gestire tali eccezioni.
- L'altra possibilità è assumere che il client debba rispettare il design by contract

Strategie non lecite:

- Utilizzare dei print nel corpo del chiamato nel momento in cui viene vìolata una certa condizione; il chiamante non è a conoscenza di ciò che è successo!
- Utilizzare uno stesso risultato di restituzione con semantiche diverse; es. restituire null quando il metodo non è stato eseguito correttamente: il chiamante potrebbe non effettuare il controllo su null!

3. Tipi e interfacce

Disaccoppiare i comportamenti dall'implementazione

Un'interfaccia di una classe è caratterizzata dai metodi che essa rende accessibili (o visibili) alle altre classi.
Ci sono molte situazioni in cui vogliamo disaccoppiare l'interfaccia dall'implementazione della classe: sono casi in cui vorremmo progettare il sistema in modo che una parte del codice possa dipendere dalla disponibilità di alcuni servizi senza essere legata ai dettagli di come essi sono implementati.

In Java le *interface* forniscono una specifica di metodi che è possibile invocare sugli oggetti di una classe che la implementano.
Un'interfaccia dichiara dei metodi astratti, pertanto è utile includere dei commenti che descrivono il behavior di ognuno, poiché le interface sono una specifica ed è importante fornire i dettagli su cosa il metodo deve fare.

Per legare una classe ad una interface si usa la keyword *implements*, la quale garantisce che le istanze di quella classe abbiano implementazioni concrete dei metodi dell'interface (forzato dal compilatore) e crea una relazione di **sottotipo**, poiché è sempre possibile definire un'istanza della classe usando il tipo dell'interface.

Questa relazione di sottotipo permette anche l'uso del **polimorfismo**, ossia la possibilità di referenziare con il tipo dell'interface, oggetti di classi diverse (a patto che implementino la stessa interface).
Il polimorfismo fornisce due principali benefici al software design:

- Diminuzione del coupling: si riduce il legame tra i metodi e le implementazioni
- Estensibilità: possibile aggiungere facilmente nuove implementazioni di una interface (nuove "forme polimorfiche")

Specificare comportamenti con le interface

L'interfaccia è un modo per definire i behaviour di una classe. Dire quindi che l'interfaccia è l'insieme dei metodi pubblici è corretto ma non completo, perchè una classe potrebbe voler fornire diverse interfacce per diversi client.

Uno dei principi di base del software design è il **principio di segregazione delle interfacce** il quale afferma che un client non dovrebbe dipnederei dai metodi che non utilizza, pertanto è preferibile che una classe implementi più interfacce, anche piccole, piuttosto che poche e grandi.
Inoltre questo rende possibile anche la riduzione delle dipendenze tra metodi ed effettive implementazioni.

Un esempio di applicazione di questi principi è nell'utilizzo di strutture dati come *ArrayList*, la quale implementa l'interface *Collection* che fornisce un metodo di ordinamento *sort*.
L'ordinamento non ha bisogno di conoscere le informazioni interne della classe di oggetti da ordinare, ma deve soltanto effettuare dei confronti.
Ci sono 2 alternative per implementare l'algoritmo di ordinamento:

- Il metodo sort deve avere a disposizione oggetti di tipo *Comparable*, una interface che dichiara soltanto un metodo, *compare*To che rende possibile la comparazione tra due oggetti; in questa soluzione la responsabilità di sapere come comparare gli oggetti è una proprietà degli oggetti stessi, poiché sono questi ultimi a dover implementare un metodo che ne permetta il confronto.
- Un'alternativa, che permetterebbe anche l'implementazione di diversi tipi di ordinamento è l'utilizzo di una interface *Comparator*, che dichiari un metodo *compare*, una funzione di comparazione tra oggetti; si crea una classe che implementa tale interface e si passa un oggetto di tale classe al metodo di ordinamento, che potrà quindi utilizzarlo per effettuare i confronti; inoltre non è neanche necessario creare una classe esplicita poiché è possibile creare delle *anonymous class* all'atto del passaggio dell'argomento.

La seconda soluzione è la migliore, perchè permette di ordinare anche oggetti che non sono di tipo *Comparable*, ed è inoltre caratterizzata da un'altra importante funzionalità di Java, i **generics**, un tipo *parametrizzato* che consente di creare comparatori di qualsiasi natura.

L'importante è evitare di ricadere nel cosiddetto **switch statement** *antipattern*, in cui si va a utilizzare switch e if-else per effettuare controlli e scegliere diversi tipi di implementazione di ordinamento.

Design Pattern: Iterator

Quando si progettano strutture dati un requisito comune è capire come accedere alla collezione senza vìolare incapsulamento ed information hiding.
Nascono quindi i concetti di **Iterator** e **Iterable**.

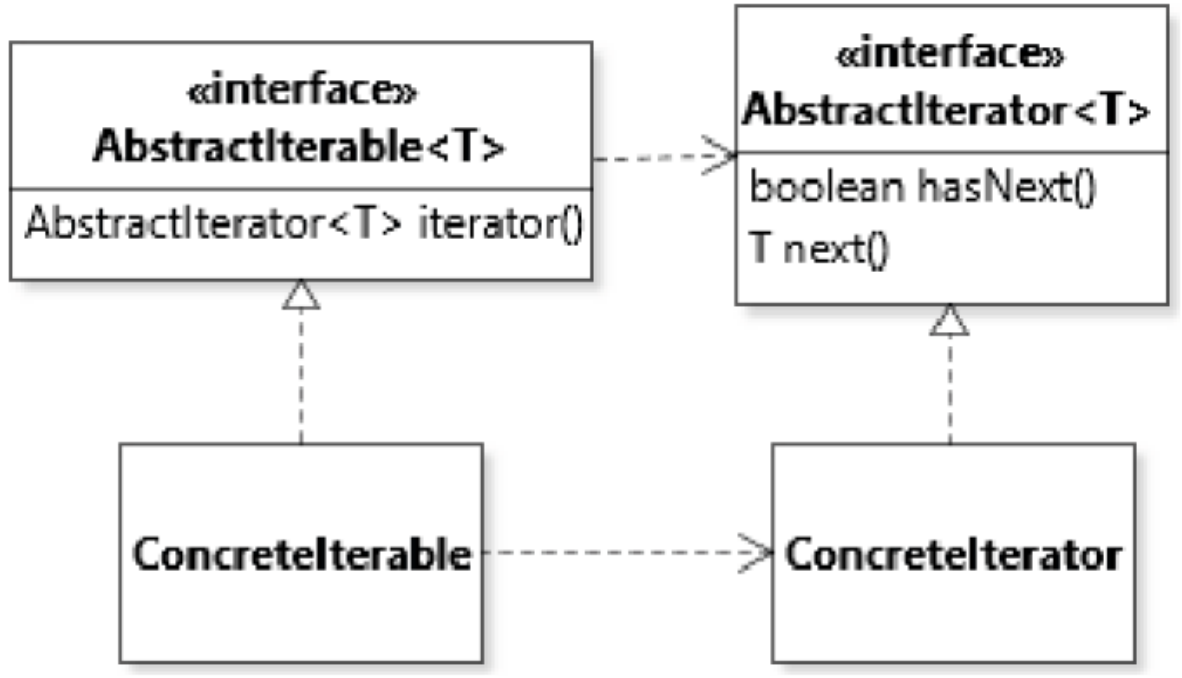
Un oggetto *iterable* è un oggetto che si candida a fornire una strategia di navigazione dei suoi dati. L'interfaccia *Iterable* ha un metodo fondamentale, *iterator()*, che restituisce un oggetto di tipo Iterator, che realizza la struttura.

L'interfaccia *Iterator<T>* dichiara invece due metodi astratti: *hasNext()* e *next()*, e permette ad un client di guadagnare l'accesso a riferimenti di oggetti di un sottotipo di *Iterator* per "iterare" su una struttura dati.

Una struttura dati implementerà quindi l'interface *Iterable*, implementando il metodo *iterator* che restituisce una istanza di un sottotipo di *Iterator*. Questo oggetto è un oggetto di una classe che implementa l'interface *Iterator* e permetterà di attraversare la struttura dati senza conoscerne i dettagli interni.

Un uso tipico degli iterator avviene nei cicli **for each**, che istanziano in maniera implicita un iterator per attraversare gli elementi di una collezione.

L'iterator è in realtà una soluzione comune che rientra in un design pattern omonimo: **Iterator**.



Questo template di soluzione prevede i concetti di:

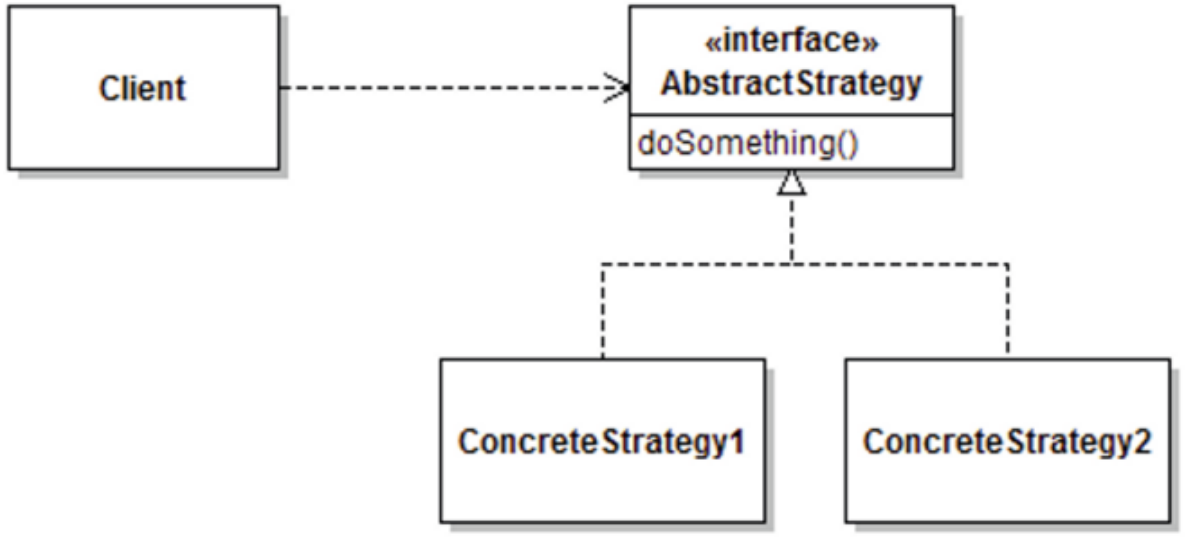
- <<interface>> AbstractIterable<T>, una interface che dichiara il metodo iterator()
- <<interface>> AbstractIterator<T>, interface che dichiara, tra gli altri, i metodi hasNext() e next()
- Una classe ConcreteIterable
- Una classe ConcreteIterator

In questo class diagram notiamo una relazione tra le due interfacce, una relazione tra le due classi, e infine una relazione di sottotipo "is a" tra le classi concrete e le rispettive interfacce.

Design Pattern: Strategy

Sia nel caso dell'iterator, sia nel caso del comparator, potrei avere la necessità di implementare strategie diverse di iterazione o comparazione. Ad esempio per gli iterator di un Binary Tree, potrei voler scorrere gli elementi in maniera pre-order, post-order o in-order.

In questi casi entra in gioco il design pattern **Strategy**, il quale permette di definire una famiglia di algoritmi intercambiabili dai client che li usano.



Struttura:

- <<interface>> AbstractStrategy che dichiara un metodo doSomething()
- Una o più classi ConcreteStrategy che implementano tale strategia implementando il metodo doSomething()

Un esempio di applicazione di questo design pattern è, come già visto, nell'utilizzo dei *Comparator*: l'interface *Comparator<T>* rappresenta l'interface di AbstractStrategy, mentre le varie classi che la implementano sono le ConcreteStrategy.

Principio di segregazione delle interface

Abbiamo analizzato i benefici della definizione di tante piccole interfacce specializzate per fornire ai client l'accesso soltanto ai behavior di cui hanno bisogno.
Questo permette anche la minimizzazione del coupling tra le classi, ma anche alcuni svantaggi, perchè talvolta i client potrebbero essere interessati a più behavior.

In certi casi può tornare utile il concetto di *estensione* di un'interfaccia, ossia all'atto della definizione di una interface dichiarare la keyword **extends** che permette di estenderne un'altra.
In questo modo l'interface appena creata dichiarerà implicitamente anche i metodi dell'interface estesa, e tutte le classi che la implementano dovranno transitivamente implementare i metodi di entrambe.

4. Design degli oggetti

Definire lo stato degli oggetti

Un concetto importante da progettare è lo stato degli oggetti, un particolare insieme di informazioni che l'oggetto rappresenta in un dato momento.

In generale possiamo definire lo **stato** come uno spazio ad **multidimensionale** (una dimensione per ogni variabile), dove ogni punto rappresenta l'insieme dei valori assunti da tutte le sue variabili di stato. Distinguiamo però due tipi di stato:

- **Stato concreto**: la collezione dei valori memorizzati nei suoi campi
- **Stato astratto**: sottoinsieme arbitrariamente definito dello spazio degli stati concreti

In particolare gli stati astratti sono utili per catturare quelle partizioni dello spazio degli stati maggiormente significative.
Ad esempio per un Deck di Card, lo stato astratto *Empty* è significativo perchè comporta l'eliminazione di uno scenario d'uso per l'oggetto (impossibilità di chiamare il metodo *draw*).

Esistono inoltre casi speciali di oggetti che non hanno alcuno stato, come le classi che implementano una singola funzione (*Comparator*), in tal caso parliamo di **stateless object**, mentre gli oggetti che lo posseggono prendono il nome di **stateful object**.

Nel caso di oggetti immutabili tale distinzione non ha senso, in quanto essi possono avere un singolo stato.

Un altro esempio è uno Stack di oggetti: esso ha virtualmente infiniti stati, ma posso caratterizzarli considerando solo quelli fondamentali:

- Stack vuoto: posso fare solo operazioni *push*
- Stack non vuoto: posso fare operazioni di *push*, *top* e *pop*;
    - Posso distinguere uno stato in cui lo stack contiene un solo elemento: in questo stato, se eseguo *pop* si torna nello stato *vuoto*

Ho definito quindi degli stati astratti, delle vere e proprie **classi di equivalenza** tra i vari stati.
Ad esempio due Stack non vuoti, anche se con oggetti diversi, sono considerati in due stati equivalenti perchè sono entrambi non vuoti.

Gli State Diagram sono molto utili per rappresentare le transizioni di stato di un oggetto di una classe. Tali transizioni sono costituite dai metodi della classe. L'assenza di una transizione implica che essa non è possibile (invalida) per un certo stato.
Il modello a stati può essere considerato anche come un **ciclo di vita** di un oggetto, perchè descrive la vita a partire dall'inizializzazione fino all'eventuale distruzione da parte del garbage collector.

Ragionare sugli stati astratti e quindi sulle classi di equivalenza è molto importante anche perchè ci danno indicazioni sulla strategia di testing da mettere in campo per valutare il sistema.

In generale, non c'è una regola unica per definire lo stato o gli stati astratti, ma è possibile considerare il cosiddetto criterio della **meaningfulness**: gli stati devono essere significativi, devono avere senso. Bisogna sempre immaginare di essere chiamati a fare il testing, ossia la convalida del codice.

Concetto di Nullability

Un aspetto molto importante di molti linguaggi di programmazione, tra cui Java, è la possibilità di assegnare valore **null** a una variabile. Questo valore indica l'assenza di valore.

Tale condizione è la causa di moltissime situazioni di *NullPointerException*, eccezione che viene lanciata quando si prova a eseguire operazioni su di una variabile che ha valore *null*.

Tony Hoare, l'inventore della **null reference**, ha affermato che questa invenzione è stata il suo *"billion dollar mistake"*.
Questo perchè in termini di low-level design le null reference complicano il flusso di un programma introducendo nuovi path che possono portare a delle dereferenziazioni a null e quindi a stati di

eccezione.
Più in generale le null reference presentano anche delle ambiguità, perchè possono essere interpretate in molti modi:

- Una variabile temporaneamente non inizializzata ma che sta per essere inizializzata
- Una variabile inizializzata non correttamente
- Valore di un flag che rappresenta l'assenza di un valore utile nel normale ciclo di vita di un oggetto
- Valore da interpretare in un modo speciale

Pertanto una best practice sarebbe progettare classi che non usino le null reference, e quindi nel progettare classi, evitare, se possibile, stati astratti di assenza di valore.
Per fare questo si può utilizzare come sempre gli approcci di input validation o design by contract.

Tipo Optional<T>

Un modo per rendere esplicito il problemi delle null reference è l'utilizzo del tipo Optional<T>, un tipo *Generic* che funge da **wrapper** per istanze del tipo T che possono essere prive di valore.

```
private Optional<Book> aBook;
```

Per rappresentare l'assenza di valore della variabile possiamo usare il valore di ritorno di *Optional.empty()*; per assegnare un valore reale ad un'istanza Optional usiamo invece *Optional.of(value)* se non ci si aspetta che value sia null, *Optional.ofNullable(value)* altrimenti.

Optional mette a disposizione anche altri metodi, ad esempio *empty()*, *get()*, *isPresent()*.
Ad esempio è possibile ottenere l'oggetto solo se presente nel seguente modo:

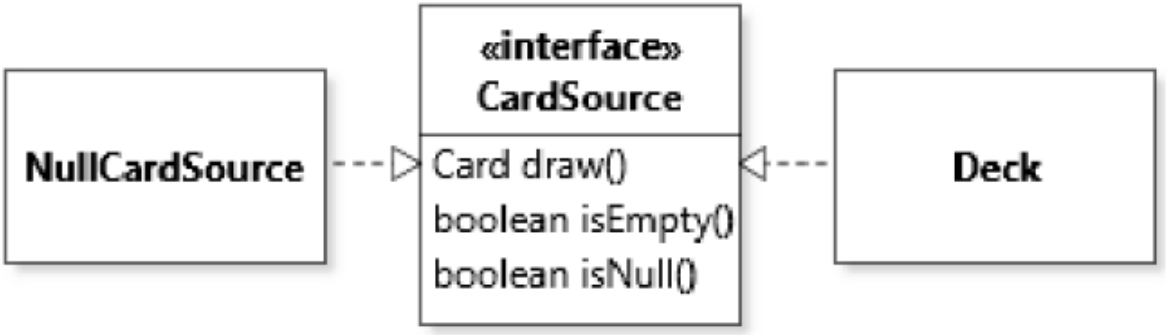
```
if x.isPresent()
        x.get()
```

Design Pattern: Null Object

Esiste una soluzione per evitare l'utilizzo delle null reference per la rappresentazione di valori assenti, che evita l'utilizzo di wrapper e rappresenta un vero e proprio design pattern chiamato **Null Object**.

L'idea di base è creare un oggetto speciale che rappresenti un valore assente e testare l'assenza usando una chiamata ad un metodo polimorfico.

Esempio di applicazione del pattern:



In questo design si aggiunge un metodo *isNull()* ad un'interfaccia per determinare la presenza del valore null e si aggiunge una classe per rappresentare tale valore (*NullCardSource*).
Il metodo *isNull()* restituisce false per qualsiasi oggetto di tipo *CardSource* ad eccezione di *NullCardSource*, true per un'istanza di *NullCardSource*.

A partire da Java 8 è possibile implementare questo pattern semplicemente modificando l'interfaccia posta alla "root" della gerarchia di tipo, e in particolare è possibile usare un metodo **default** per evitare che tutte le classi che implementino l'interfaccia debbano implementare il metodo restituendo true.

Un'alternativa per risparmiare linee di codice è usare una classe anonima come costante dell'interface che implementa il metodo *isNull()* restituendo true e poi usare un metodo default per restituire false, in questo modo si evita di costruire una classe separata per la gestione del null.
Esempio:

```
public interface CardSource {
        public static CardSource NULL = new CardSource {
                        public boolean isEmpty() { return true; }
                        public Card draw() { assert !isEmpty(); return null; }
                        public boolean isNull() { return true; }
        }

        Card draw();
        boolean isEmpty();
        default boolean isNull() { return false; }
}
```

Campi costanti e variabili

In alcuni casi potremmo voler dichiarare dei campi come **final**, i cui valori possono essere assegnati soltanto una volta, o in fase di dichiarazione, o nel costruttore. Tentare di riassegnare il valore di un campo *final* genererà un errore di compilazione.

Importante ricordare che per tipi non primitivi il valore memorizzato è sempre una referenza ad un oggetto: dunque, sebbene non sia possibile riassegnare un valore a un campo final, è certamente possibile cambiare lo stato dell'oggetto referenziato (se l'oggetto è mutabile).

I campi final sono utili per restringere lo spazio degli stati di un oggetto e rendere più semplice la comprensione dei behavior di un oggetto a run-time, anche se non rendono gli oggetti referenziati immutabili.

Object Identity, Equality e Uniqueness

Tre concetti importanti da tenere in mente nel design del ciclo di vita degli oggetti sono **identità**, **uguaglianza** e **unicità**.

L'**identità** fa riferimento ad un oggetto, anche se esso non è referenziato da una variabile: l'identità di un oggetto si riferisce quindi alla sua locazione di memoria, o al suo riferimento. Negli IDE, come Eclipse, questa identità è rappresentata da un object id.
In Java l'operatore == restituisce true se due operandi hanno lo stesso valore; nel caso di tipi reference per "stesso valore" si intende stesso object id; pertanto due variabili che referenziano oggetti uguali nello stato, ma distinti nell'identità, restituiranno false in un predicato di uguaglianza con tale operatore.

L'**uguaglianza** è un concetto più complesso perchè dipende dal design della classe. In generale potremmo dire che due oggetti sono uguali se tutti i campi assumono gli stessi valori; talvolta, in classi molto complesse, potremmo voler considerare uguali anche oggetti che presentano soltanto soltanto alcuni campi uguali, o ancora due Set possono essere considerati uguali se contengono gli stessi elementi, anche se memorizzati in ordine diverso.

Java fornisce un meccanismo per specificare l'uguaglianza mediante l'overriding del metodo **equals** dalla classe *Object*. L'implementazione di default definisce l'uguaglianza come l'identità (uguaglianza tra

gli id), quindi se non ridefiniamo tale metodo, due variabili che referenziano istanze di una classe saranno ritenute uguali solo se referenziano lo stesso oggetto in memoria.

Ogni classe che sovrascrive il metodo *equals* deve inoltre sovrascrivere il metodo *hashCode()* in modo da mantenere il vincolo secondo cui due oggetti sono considerati uguali se l'invocazione del metodo hashCode su entrambi produce lo stesso risultato intero.

Proprietà di **unicità**: gli oggetti di una classe sono unici se non è possibile avere due oggetti uguali. Se possiamo garantire l'unicità allora non ha senso definire l'uguaglianza, perchè essa sarebbe uguale all'identità e ci consente di usare direttamente l'operatore == .

Anche se non è possibile garantire l'unicità stretta degli oggetti a causa del metaprogramming e della *serialization*, esistono dei design pattern che forniscono una certa garanzia di unicità.

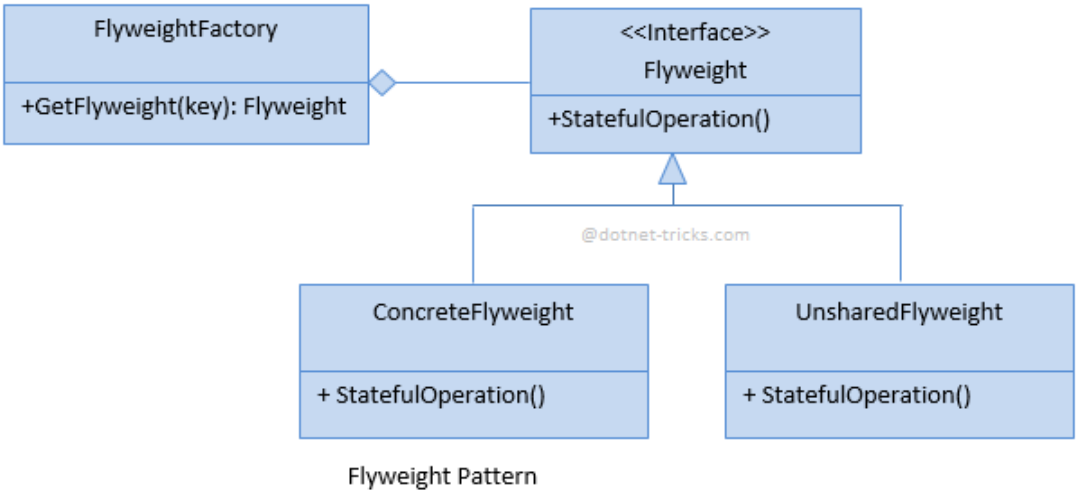
Design Pattern: Flyweight

Si tratta di un pattern struttuale che fornisce un modo per gestire collezioni di oggetti immutabili e consente di assicurare l'unicità degli oggetti di una classe.
Il contesto di utilizzo è la presenza di classi fortemente condivise in un sistema software.

L'idea di base è di controllare la creazione di oggetti di una certa classe chiamata *flyweight class* attraverso un metodo di accesso che assicuri l'impossibilità di creare oggetti duplicati (distinti ma uguali). Per realizzare questo pattern c'è bisogno di 3 principali componenti:

- Un **costruttore privato** per la flyweight class, per impedire che i client possano creare oggetti
- Uno static **flyweight store** che mantiene una collezione di istanze della classe flyweight
- Un **metodo static di accesso** che restituisce un oggetto flyweight unico; il metodo tipicamente effettua un check per capire se l'oggetto richiesto esiste già nello *store* e, in alternativa, crea e restituisce l'oggetto

Schema generale del pattern:



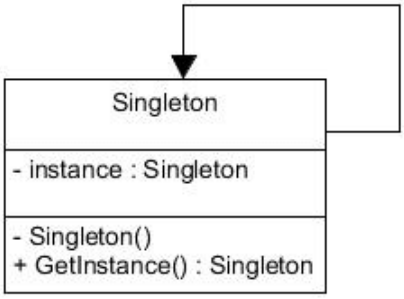
Flyweight Pattern

Questo pattern viene utilizzato in particolare per ottimizzare l'utilizzo delle risorse, infatti il suo utilizzo può migliorare le prestazioni di una applicazione.
In particolare in Java, ogni volta che viene utilizzato l'operatore new, la JVM alloca nell'heap uno spazio in memoria di 32 bit, pertanto un'eccessiva generazione di oggetti può portare alla saturazione delle risorse del sistema.
Molto spesso la generazione di nuovi oggetti non è motivata da reali esigenze, ma dovuta a superficialità o errata analisi degli impatti prestazionali. Da qui l'esigenza di riutilizzare gli oggetti precedentemente creati ai fini del loro riutilizzo.

Nota importante: gli oggetti flyweight dovrebbero essere immutabili per assicurare la loro unicità.

Design Pattern: Singleton

Questo pattern permette di assicurare l'esistenza di un'**unica** istanza di una data classe in qualsiasi punto del codice del programma.
Il contesto di utilizzo sta nella necessità di gestire un'istanza di una classe che mantenga una grande quantità di informazioni importanti di cui diverse parti del codice potrebbero essere interessate.



Il template della soluzione del pattern **Singleton** è costituito da 3 elementi:

- Un **costruttore privato**
- Una **variabile statica e privata** che mantiene una referenza ad una singola istanza dell'oggetto singleton
- Un **metodo di accesso** (di solito chiamato *instance* o *getInstance*) che restituisce l'istanza se già istanziata, altrimenti la crea invocando il costruttore e la restituisce

A differenza del pattern *Flyweight* gli oggetti *Singleton* sono tipicamente *stateful* e mutabili, mentre gli oggetti *Flyweight* dovrebbero essere immutabili.

Nota: attenzione ai contesti multi-threading, pensare ad una implementazione del pattern che sia **thread-safe** e gestisca al meglio gli accessi concorrenti.

Oggetti di classi innestate

Le classi innestate possono essere divise in due grandi categorie:

- **Inner class**: classi dichiarate all'interno di altre classi che di solito forniscono behavior addizionali che coinvolgono un'istanza della classe esterna ma che per qualche motivo non vogliamo integrare in essa.
- **Static nested class**: si distinguono dalle inner class perchè non sono collegate ad istanze della classe in cui sono innestate. Sono usate principalmente per incapsulamento e organizzazione del codice.

Tra le *inner class* distinguiamo altre due categorie speciali:

- *Anonymous class*: spesso utilizzate per implementare function object (classi che implementano semplicemente una singola funzione).
- *Local class*: simili, ma più raramente utilizzate

Sintesi finale di capitolo

- Per classi stateful considerare l'utilizzo di state diagram per ragionare sugli stati astratti e sul ciclo di vita degli oggetti
- Cercare di minimizzare il numero di stati astratti significativi e impedire stati invalidi o inutili
- Evitare l'utilizzo di null reference per rappresentare informazioni legali nelle variabili degli oggetti; considerare l'utilizzo del tipo *Optional* e/o del pattern **Null Object** se necessari

- Considerare la dichiarazione di campi *final* se possibile
- Rendere esplicito se gli oggetti di una classe debbano essere unici o meno
    - Se si, considerare l'utilizzo del pattern **Flyweight** per forzare tale unicità, o **Singleton** per classi di cui si vuole assicurare la presenza di una singola istanza.
    - In caso contrario, sovrascrivere i metodi *equals* e *hashCode
- Ricordare che è possibile aggiungere dati addizionali ad istanze di inner class sia in forma di referenza ad un'istanza di una classe esterna, sia come copia delle variabili locali in un blocco

6. Composizione

Composizione ed aggregazione

Una strategia generale per gestire la complessità nel software design è definire astrazioni grandi in termini di astrazioni più piccole, secondo un principio di problem solvingdi tipo "*divide et impera*".

La **composizione** è un modo per definire un oggetto composto da altri oggetti, ossia che memorizza riferimenti ad uno o più altri oggetti.
Si tratta di una astrazione di una rappresentazione di una collezione di altre astrazioni.

Es. un Deck è un'istanza di una collezione di Card, mentre una String è una collezione di caratteri di tipo char.

Una variante della relazione di composizione è l'**aggregazione**, che rappresenta una astrazione in cui gli elementi aggregati delegano alcun i servizi ad oggetti che servono un ruolo di servizi specializzati al proprio aggregato (?).
La composizione è, in realtà, una forma più forte dell'aggregazione.

Una importante proprietà della composizione è la **transitività**: un oggetto composto da altri oggetti può a sua volta essere componente o delegato di altri oggetti.

Nei class diagram la composizione è rappresentata da archi che terminano con un simbolo di diamante nel lato della classe che mantiene le referenze delle istanze della altre classi, e un diamante colorato per la relazione di aggregazione.

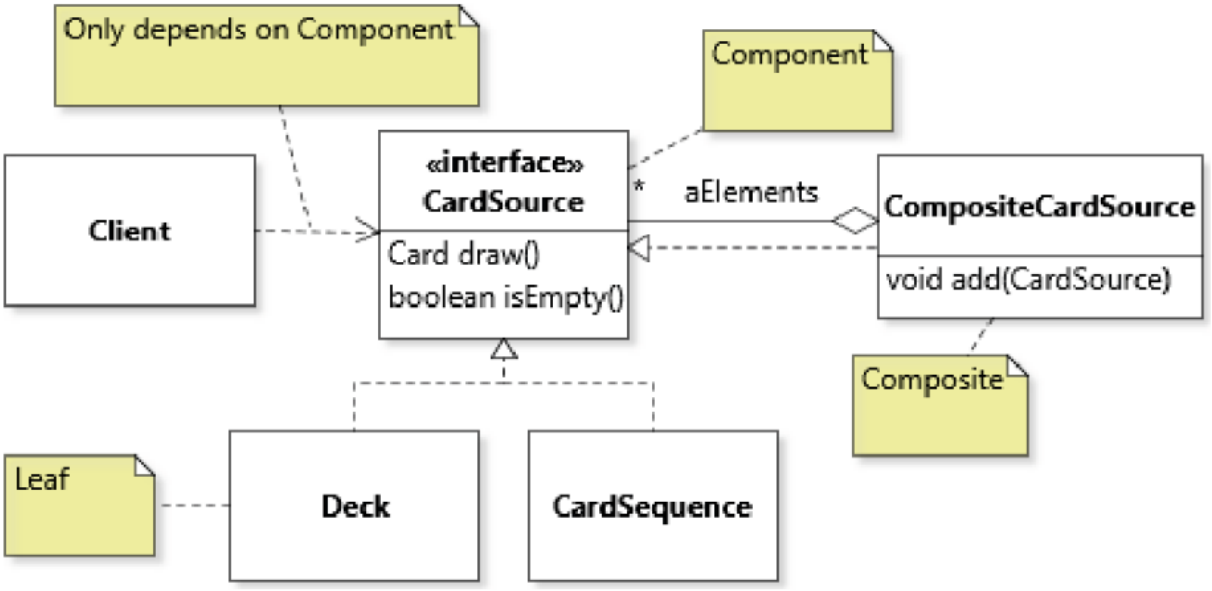
Design Pattern: Composite

Consideriamo una situazione in cui vogliamo raggruppare oggetti che si comportano come un singolo oggetto.
In generale potremmo creare una interface che disaccoppia il comportamento di una composizione generica di oggetti dalla sua implementazione e scrivere diverse classi che la implementano per sfruttare il polimorfismo.

Questa decisione di design fa sì che l'insieme delle possibili implementazioni sia specificato staticamente, nel codice sorgente e non dinamicamente (a run-time). 3 principali limitazioni delle strutture statiche sono:

- Il numero delle possibili strutture di interesse può essere molto elevato (molte definizioni di classi)
- Ogni opzione richiede la definizione di una classe, anche se usata raramente
- Difficile cambiare scelte a runtime

Entra così in gioco il design pattern chiamato **Composite**.
Esempio di applicazione del pattern:



In questo template distinguiamo 3 ruoli principali:

- **Component**: interfaccia o classe astratta che dichiara i metodi resi disponibili al client, il quale deve sapere soltanto quali metodi può utilizzare senza preoccuparsi della gerarchia di classi incluse nel component; permette di definire diversi tipi di componenti per i composite
- **Composite**: classe che aggrega/si compone di component; può essere a sua volta un component (implementare l'interface) creando così una gerarchia
- **Leaf**: la foglia, l'oggetto semplice che implementa l'interface *Component*

Quando si applica questo pattern un problema importante di implementazione è il considerare coem aggiungere al composite delle istanze del component che lo compone, ossia un modo per specificare quali sottotipi dell'interface compongono il composite. Due modi:

- Fornire un metodo per aggiungere elementi nel composite, permettendo in questo modo la modifica a run-time del composite
- Inizializzare gli oggetti composite attraverso i loro costruttori, usando ad esempio costruttori copia per evitare il leaking delle referenze a una struttura dati privata

Se la modifica a run-time del composite non è necessaria la seconda è l'opzione da preferire, sempre perchè legata al concetto di immutabilità e ai suoi vantaggi.

Design Pattern: Decorator

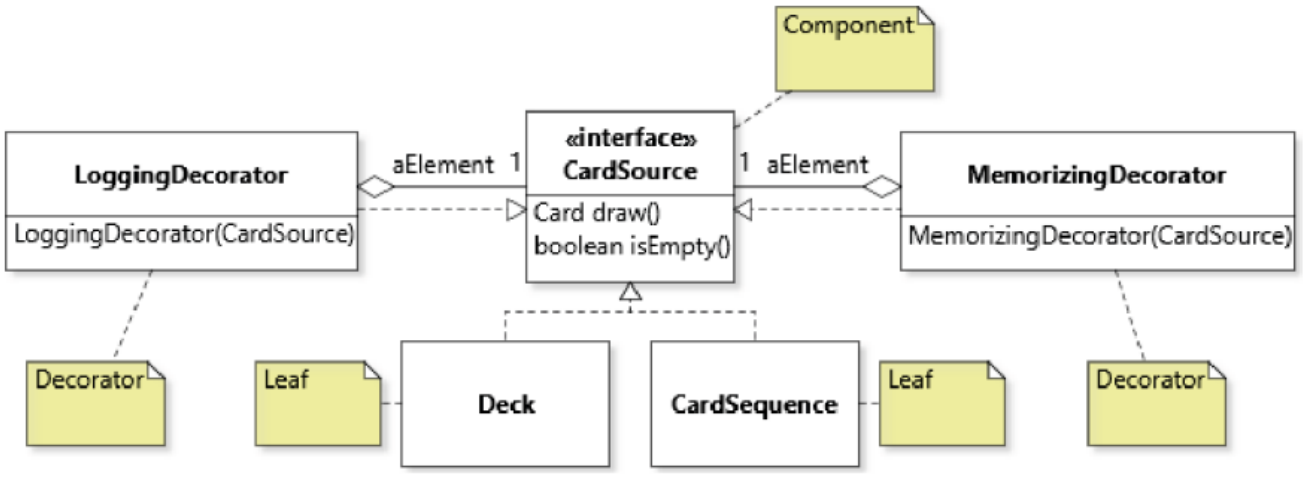
In alcuni casi vorremmo che alcuni oggetti abbiano dei comportamenti speciali o caratteristiche aggiuntive.
Quando è stata introdotto il paradigma OO il modello principale utilizzato per estendere la funzionalità dell'oggetto era l'ereditarietà.
Non sempre però questo è l'approccio più giusto. Infatti è stato dimostrato che estendere gli oggetti usando l'ereditarietà spesso si traduce in una gerarchia di classi che finisce per esplodere, un fenomeno noto come **exploding class hierarchy**.

Il design pattern **Decorator** fornisce un'alternativa flessibile all'ereditarietà per estendere la funzionalità degli oggetti, poiché consente di arricchire dinamicamente, a run-time, un oggetto con nuove funzionalità: è possibile impilare uno o più decorator uno sopra l'altro, dove ognuno aggiunge nuove funzionalità.
Principali differenze tra *Decorator* e ereditarietà:

- Un *Decorator* agisce a runtime a differenza dell'ereditarietà che estende con le sottoclassi il comportamento della classe padre in fase di compilazione.
- Un *Decorator* può operare su qualsiasi implementazione di una determinata interfaccia, eliminando la necessità di creare sottoclassi di un'intera gerarchia di classi.
- La sottoclasse aggiunge comportamento a tempo di compilazione e la modifica interessa tutte le istanze della classe originale; il decorator pattern può fornire nuovi comportamenti in fase di esecuzione per i singoli oggetti.
- L'uso del *Decorator* porta a un codice più pulito e testabile, mentre i servizi creati con l'ereditarietà non possono essere testati separatamente dalla sua classe padre perché non esiste un meccanismo per sostituire una classe padre con uno stub.

Il decorator pattern è quindi molto utile e utilizzato dai programmatori più esperti, i quali lo utilizzano al posto della ereditarietà in situazioni in cui è necessario aggiungere/modificare a runtime il comportamento di un oggetto senza scomodare l'ereditarietà. Un caso noto di utilizzo del decorator è quello di alcune classi del core di Java.

Esempio di applicazione del Pattern:



La struttura del pattern Decorator, molto simile a quella del pattern Composite, consta di 4 elementi principali:

- **Component** che rappresenta l'interfaccia dell'oggetto che deve essere decorato dinamicamente
- **ConcreteComponent**: la classe di oggetti *Component* a cui si vuole aggiungere nuove funzionalità
- **Decorator**: interfaccia tra *Component* e *ConcreteDecorator* che possiede un riferimento a un *Component* (aggrega un oggetto *Component*) e ne estende l'interfaccia
- **ConcreteDecorator**: classe che implementa il *Decorator* aggiungendo nuove funzionalità rispetto al *ConcreteComponent*

Nell'implementazione del pattern è buona norma specificare come *final* il campo che memorizza il riferimento all'oggetto decorato e inizializzarlo nel costruttore. Questo perchè di solito un oggetto decorator è destinato a decorare lo stesso oggetto durante tutto il suo tempo di vita.

Il pattern *Decorator* incoraggia il programmatore a scrivere codice che rispetta i principi **SOLID**: è un pattern utile infatti per rispettare sia il **principio di singola responsabilità**, perchè consente di dividere le funzionalità tra classi distinte, sia per l'**open-closed principle**, che consente di aggiungerne di nuove senza modificare le classi già esistenti.

Nonostante tutto il pattern ha anche degli svantaggi:

- Tutti i metodi dell'interfaccia *Decorator* devono essere implementati nel *ConcreteDecorator*, anche quelli che nn aggiungono alcun comportamento aggiuntivo; l'ereditarietà permette invece di

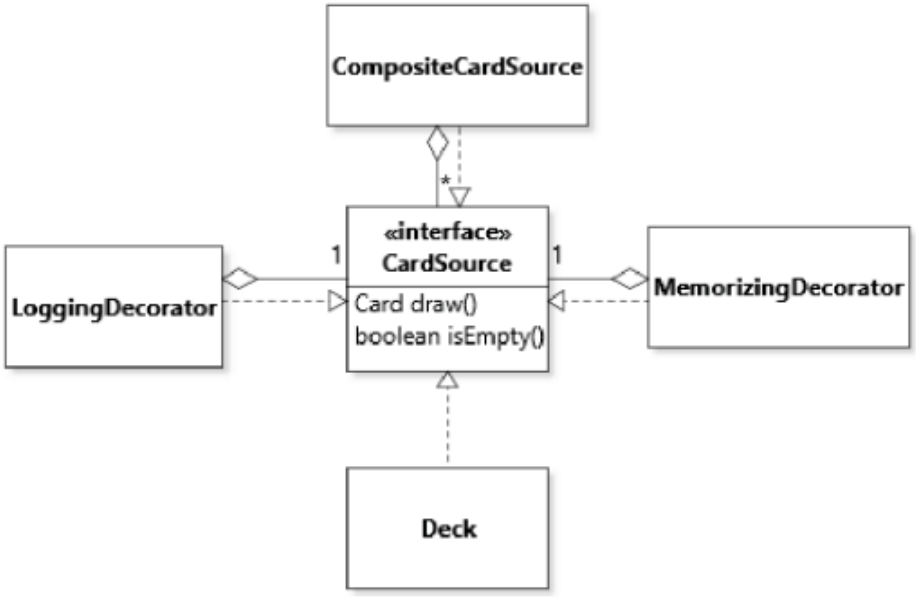
implementare solo metodi che modificano o estendono i comportamenti della superclasse
- I *Decorator* possono complicare il processo di creazione di un'istanza del *Component* perché non lo si deve solo istanzire ma anche "decorare" in un certo numero di *Decorator*
- Può essere complicato gestire *Decorator* che tengono traccia di altri *Decorator*

Si consiglia di usare il Decorator quindi nelle seguenti situazioni:

- Quando le responsabilità e i comportamenti degli oggetti dovrebbero essere modificabili dinamicamente
- Per disaccoppiare le implementazioni concrete da responsabilità e comportamenti

Combinare Composite e Decorator

Sebbene siano due pattern distinti essi possono coesistere in una gerarchia di tipo.
Se *Composite* e *Decorator* aggregano e implementano la stessa interfaccia *Component* possono funzionare insieme e supportare soluzioni basate sulla composizione.



In questo esempio vi è una gerarchia di tipo con una foglia, un composite e due decorator.

Esempio di applicazione: costruzione di un editor di disegno in cui un'interface Component "Figure" è implementata da un Composite e da un Decorator per supportare le funzionalità di aggregazione di figure e di decorazione.

Copia di oggetti polimorfici

Immaginiamo un contesto in cui abbiamo una collezione di oggetti referenziati da un tipo interface: in questo caso, se volessimo effettuare una copia, dovremmo conoscere il tipo concreto degli oggetti che compongono la collezione per poterne fare delle copie, invocandone i rispettivi costruttori.

Per evitare di ricadere nell'antipattern *switch statement* nell'implementazione della copia rompendo così i benefici del polimorfismo, vorremmo un meccanismo che ci permetta di copiare tali oggetti polimorfici, ossia poter creare copie di oggetti senza conoscere il loro tipo concreto.

In Java tale funzionalità è supportata mediante un meccanismo chiamato **cloning**. Per creare oggetti **cloneable** occorrono 4 step obbligatori (più un quinto opzionale):

- Dichiarare o implementare una interface **Cloneable**
- Fare overriding del metodo *Object.clone()*

- Invocare super.clone() nel metodo clone(): in questo modo si fa uso del metaprogramming per ottenere esattamente una nuova istanza della classe da cui il metodo origina copiandone tutti i campi; qualora la copia non sia sufficiente è possibile implementare altre operazioni, ad esempio nel caso di classi Collezione, in cui copiare il riferimento non basta ma serve una copia più profonda (usare ad esempio un costruttore copia per la collezione passando la collezione stessa); distinguiamo quindi i **deep-clone** (copia profonda) dagli **shallow-clone** (copia superficiale).
- Catturare l'eccezione *CloneNotSupportedException*: questo perchè tale eccezione viene sollevata qualora il metodo *clone* venga invocato su oggetti di classi che non implementano l'interface *Cloneable* direttamente o transitivamente; si tratta quindi in un metodo per forzare il programmatore a ricordare questa caratteristica.
- Opzionalmente dichiarare il metodo clone() nel supertipo radice (classe/interfaccia) della gerarchia; dichiarare *Cloneable* non basta, perchè tale interface non dichiara il metodo clone ma ne abilita solo l'utilizzo.

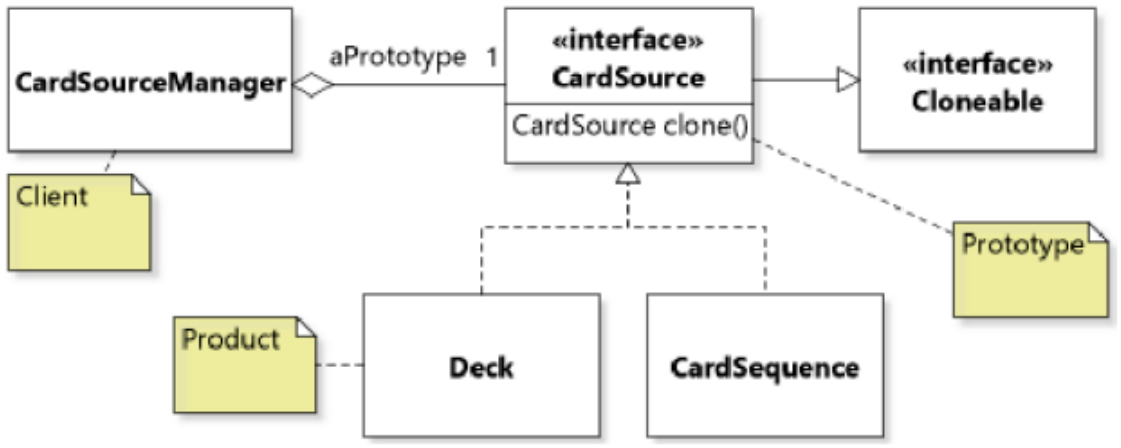
Esempio di implementazione:

```
public Deck clone() {
        Deck clone = (Deck) super.clone();
        clone.aCards = new CardStack(aCards);
        return clone;
}
```

Design Pattern: Prototype

Un particolare uso della copia di oggetti polimorfici è il supporto all'*istanziazione polimorfica*. Immaginiamo di voler creare nuovi oggetti polimorfici a runtime senza utilizzo dell'antipattern *switch statement*. Possiamo avvalerci della clonazione e creare cloni a run-time.

Il contesto di utilizzo del pattern *creazionale* **Prototype** è proprio la creazione di oggetti, i cui tipi non sono conosciuti a tempo di compilazione.
Esempio di applicazione del pattern:



La struttura è composta da 3 partecipanti:

- **Prototype**: classe astratta/interfaccia per gli oggetti da clonare che estende *Cloneable* e dichiara il metodo *clone*
- **ConcretePrototype** (o *Product*): classe che implementa l'operazione di clonazione di sé stessa
- **Client**: invoca la clonazione dell'oggetto usando un riferimento all'interface *Prototype*

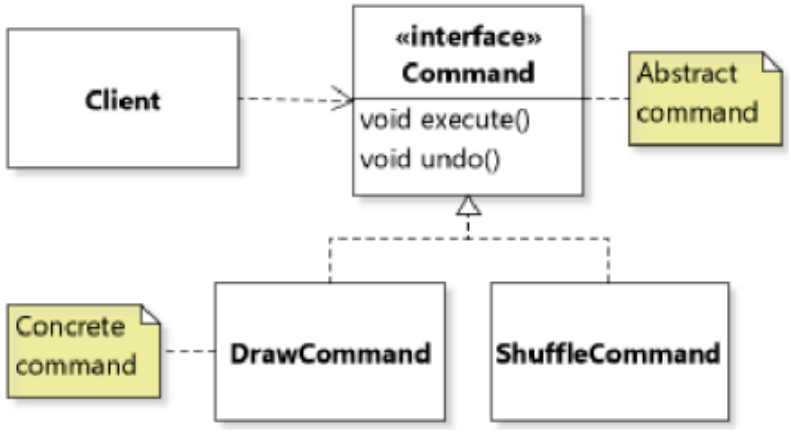
Questo pattern permette quindi di utilizzare il polimorfismo per creare istanze di classi senza alcuno statement di controllo.

Esempio di implementazione: classe HashTable che implementa Cloneable e viene implementata da classi concrete come HashMap, LinkedHashMap, etc.
L'implementazione di base del metodo clone permette una copia superficiale dell'istanza, ma è possibile ridefinire il comportamento prevedendo una copia deep, andando a copiare anche ogni elemento contenuto nella struttura dati.

Design Pattern: Command

Concettualmente un *command* è un pezzo di codice che esegue un certo task, ad esempio l'esecuzione di un metodo o di una funzione.

Il contesto di utilizzo di questo pattern *comportamentale* è quello di disaccoppiare l'invocazione di un "comando" dai suoi dettagli implementativi, separando quindi chi invoca il comando da colui che esegue l'operazione.
Tale operazione viene realizzata attraverso la catena: *Client* → *Invocatore* → *Ricevitore*



- Il *Client* chiama il metodo dell'invocatore senza conoscere i dettagli del comando ed imposta il *Receiver*
- L'*invocatore* ha l'obiettivo di incapsulare e nascondere i dettagli della chiamata ed effettua l'invocazione del comando
- Il *Command* funge da interfaccia generica per l'invocazione del comando
- Il *ConcreteCommand* è l'implementazione specifica del comando che collega l'*invoker* con il *receiver*
- Il *Receiver* è colui che riceve il comando e sa come eseguirlo

Di solito il *Command* memorizza una referenza del target per poter invocare il comportamento, ma sono possibili alternative, come il passaggio di argomenti.
Inoltre i metodi di interfaccia di un *Command* sono solitamente *void* ma possono anche includere la restituzione di un output.

Qualora sia necessario memorizzare dei dati a supporto delle operazioni invocate dai *Command* è possibile farlo direttamente negli oggetti *Command* o usando strutture esterne da essi accessibili.

Vantaggi del pattern:

- Riduzione dell'accoppiamento: il *Command* disaccoppia l'invocatore dal ricevitore, ossia colui che invoca l'operazione da colui che la esegue; i dettagli implementativi sono a conoscenza soltanto del receiver.
- Facile estendibilità: è possibile aggiungere facilmente nuovi comandi creando nuove classi che implementano l'interface *Command*.

Legge/Principio di Demeter/Demetra

Nel progettare pezzi di software usando l'aggregazione si rischia di finire in una lunga catena di delegazioni tra oggetti sfociando nell'antipattern che prende il nome di **message chain**.

Per evitare questo antipattern il principio di Demeter offre delle linee guida affermando che un metodo deve avere accesso soltanto a:

- Variabili di istanza del suo parametro implicito
- Argomenti passati al metodo
- Qualsiasi nuovo oggetto creato all'interno del metodo
- Eventuali oggetti disponibili globalmente (se necessario)

In sintesi, ogni oggetto dovrebbe conoscere il minimo indispensabile di un insieme di oggetti che sia più piccolo possibile.

Ad esempio in una catena di invocazioni, le classi che occupano posizioni intermedia nella catena dovrebbero fornire servizi addizionali per evitare che i client manipolino oggetti interni incapsulati da rssi.
Gli oggetti interni non devono restituire referenze a strutture interne ma soltanto chiamare altri servizi per completare la richiesta del client.

Sintesi di capitolo

- Semplificare classi grandi introducendo classi i cui oggetti forniscono servizi alla classe iniziale
- Usare *Composite* quando si vuole manipolare collezioni di oggetti *Leaf* come un'unica entità
- Usare *Decorator* quando si vuole aggiungere funzionalità a certi oggetti ma continuando a poterli usare allo stesso modo degli oggetti regolari
- Possibile combinare *Composite* e *Decorator*, specialmente se implementano/estendono un tipo in comune
- Implementare i pattern non basta per assicurare la correttezza del codice; i client sono sempre responsabili di assicurare che il pattern non degeneri
- I sequence diagram sono utili per capire le catene di chiamate tra gli oggetti
- Usare il cloning (copia polimorfica) per creare copie di oggetti di cui non si conosce il tipo concreto a runtime, altrimenti preferire i costruttori copia
- Usare *Prototype* se si vuole creare nuove istanze di oggetti a runtime, sfruttando il cloning
- Usare *Command* per fornire funzioni oggetto gestibili esplicitamente dal codice client (?), ma fare attenzione a non rompere l'incapsulamento: solo gli oggetti *command* possono operare sugli oggetti target
- Nell'implementazione dei metodi, seguire le linee guida fornite dal Principio di Demeter

7. Ereditarietà

Fino ad ora abbiamo analizzato il polimorfismo come strumento utile per un design estensibile che permetta il disaccoppiamento tra codice client e concrete implementazioni di funzionalità richieste.

Talvolta però bisogna stare attenti a non cadere nell'antipattern **duplicated code** in cui si creano molte classi che implementano la stessa interfaccia ma presentano ridondanza di codice.
Un meccanismo che permette il riutilizzo di codice già scritto è proprio l'**ereditarietà**, la quale permette di creare classi a partire dalla definizione di altre classi.

L'idea chiave è definire una **sottoclasse** che estende i behavior ed eventualmente lo stato di una **superclasse**.
L'ereditarietà permette di evitare la ridichiarazione dei campi in comune, i quali vengono automaticamente ereditati dalle istanze delle sottoclassi.

Anche nell'ereditarietà vi è una **relazione di tipo** "is a": ogni istanza delle sottoclassi è un'istanza della propria superclasse (ed eventuali successive superclassi nella gerarchia).

Nei class diagram rappresentiamo l'ereditarietà con una linea non tratteggiata (come invece è nel polimorfismo di interfaccia) e un triangolo che termina nella superclasse.
In Java realizziamo questa relazione usando la keyword **extend** in fase di definizione di una sottoclasse.

Tipo statico e tipo dinamico

Quando istanziamo un oggetto di una sottoclasse il suo tipo a run-time sarà il tipo specificato nello statement *new*, tuttavia grazie all'ereditarietà tale oggetto può essere referenziato anche da una variabile di una superclasse (o interfacce).
Distinguiamo quindi:

- **Tipo statico**: tipo della variabile di riferimento, può essere uguale al tipo dinamico o diverso (tipo della superclasse o di una interfaccia implementata)
- **Tipo dinamico**: tipo dell'oggetto a run-time stabilito dallo statement new; non può cambiare dopo essere stato assegnato; è il tipo di ritorno del metodo getClass()

Possibile usare il **downcasting** per permettere una conversione del tipo statico di un oggetto, anche se non sicura, poiché bisognerebbe assicurarsi che il tipo dinamico coincida con il tipo statico di destinazione.
Per fare questo è possibile usare l'operatore **instanceof** o il metodo getClass(), che permette di verificare il tipo dinamico prima di un'operazione di downcasting.

Ereditarietà singola

Java supporta l'erediterietà singola (single rooted), ossia ogni sottoclasse può estendere al più una superclasse, ma ci sono linguaggi che hanno la possibilità di avere più radici → ereditarietà multipla (es. Python, C++). Lo stesso non vale per le interface.
Nonostante questo è sempre possibile creare gerarchie di sottoclassi, creando una relazione di sottotipo transitiva.

Se una classe non dichiara di estendere alcuna classe, di default essa sarà una sottoclasse della classe *Object*, la radice di ogni classe.

Ereditarietà dei campi

Le sottoclassi ereditano i campi dichiarati nella superclasse e possono memorizzare informazioni in essi, oltre che aggiungerne di nuovi.

Se un campo è *private* la sottoclasse non può però accedervi se non usando metodi get e set, ma può comunque inizializzarne il valore sfruttando il costruttore.
Un'alternativa è dichiarare tale variabile *protected*, specificatore di accesso che rende accessibile un campo o un metodo alle sottoclassi e alle classi dello stesso package.

I costruttori di una gerarchia di classi ereditaria devono seguire un ordine *top down*: i campi più generali devono essere inizializzati nella superclasse.
Per permetterne l'inizializzazione anche nelle sottoclassi si usa *super()* come prima istruzione del costruttore, che permette di invocare il costruttore della superclasse e inizializzare quel pezzo di stato non accessibile.
N.B: usare super è diverso da usare new perchè non crea un'altra istanza!

Ereditarietà dei metodi

A differenza dei campi i metodi non memorizzano informazioni che rappresentano lo stato di un oggetto e non richiedono pertanto inizializzazione.

Quando creiamo un oggetto della sottoclasse egli può invocare metodi della superclasse, ma ovviamente non il viceversa.
Inoltre, non è possibile invocare un metodo esclusivo della sottoclasse qualora il tipo statico dell'oggetto coincida con il tipo della superclasse.

Questo accade perchè il tipo statico cambia la visibilità.

Overriding

In alcuni casi potremmo voler ridefinire un metodo della superclasse cambiandone il comportamento esclusivamente per le istanze delle sottoclassi.

Comodo e utile utilizzare l'annotazione @Override quando sovrascriviamo un metodo di una superclasse; si tratta di un check di qualità, il compilatore controlla se la signature è stata rispettata.

Per capire quale implementazione viene scelta a runtime per l'invocazione di un metodo overriden occorre introdurre il concetto di **binding statico** o **dinamico**.
Java utilizza il **late binding** (dinamico), ossia è il tipo dinamico dell'oggetto che determina il metodo da invocare. L'accoppiamento tra messaggio e codice effettivo da eseguire avviene in ritardo, a tempo di esecuzione.
L'unico modo per forzare l'esecuzione del metodo di una superclasse è usare come parametro implicito *super* e chiamare su di esso il metodo.

In sintesi: il tipo statico determina quali metodi possiamo invocare su un oggetto, il tipo dinamico determina il codice effettivo che verrà eseguito.
Ogni oggetto ha un tipo statico e un tipo dinamico: la gerarchia di tipo sui riferimenti (parte statica) non sovrascrive mai la natura dinamica di un oggetto.

Overloading

Mentre l'overriding permette di ridefinire il comportamento di un metodo nelle classi di una gerarchia ereditaria, l'**overloading** è un meccanismo per specificare diverse implementazioni dello stesso metodo nella stessa classe, con la possibilità di parametri espliciti differenti (sia in numero, che nei tipi).

In questo caso il binding è **statico** (early binding), perchè la chiamata del metodo può essere risolta a tempo di compilazione grazie al controllo statico dei tipi degli argomenti passati nel messaggio.

I fenomeni di binding statico e dinamico si intersecano quando nell'overloading si usano parametri che possono essere oggetto di late binding.
In questo caso, pur passando un oggetto che è dinamicamente di un sottotipo, viene scelto il metodo dettato dalla natura statica.
In sintesi:

- Nel caso di interferenza tra overloading e overriding → vince il tipo **statico**
- Quando non c'è overloading → vince il tipo **dinamico**.

Capiamo quindi che il polimorfismo complica le strategie di testing, perchè rende più difficile seguire il controllo del flusso e dà vita a più branch.
Possibile però una soluzione conservativa: in una espressione polimorfica, essendo la gerarchia statica, posso provare a fare una strategia esaustiva in cui testo tutte le possibili implementazioni del metodo, poiché il numero di binding, per quanto alto, è comunque finito.
Es. in Android non è possibile la soluzione conservativa perchè (?)

Ereditarietà vs Composizione

L'ereditarietà fornisce una alternativa alla composizione come approccio di design per gestire situazioni dove alcuni oggetti sono delle "estensioni" di altre.

La principale differenza tra i due approcci è che l'approccio basato sulla composizione richiede di combinare il lavoro di due oggetti: un oggetto base e un wrapper (o decorator).

Oltre ad essere possibile la combinazione di entrambe le soluzioni, la scelta dell'approccio migliore dipende al contesto:

- La composizione permette maggiore flessibilità a run-time, preferibile in contesti che richiedono molte possibili configurazioni, o l'opportunità di cambiare configurazione a run-time, ma al tempo stesso fornisce meno soluzioni per ottenere accesso allo stato interno di oggetti ben incapsulati.
- L'approccio con ereditarietà è preferibile in contesti che richiedono molte configurazioni a tempo di compilazione, perchè la gerarchia può essere facilmente progettata per fornire accesi privilegiati alla struttura interna di una classe alle sottoclassi

---

Classi astratte

La **classe astratta** risponde a una precisa esigenza: dare visibilità a livello della superclasse di ciò che è proprio delle sottoclassi concrete.

Una classe astratta è una classe che presenta almeno un metodo dichiarato come *abstract*, ossia non implementato.
Principali conseguenze di dichiarare una classe astratta:

- La classe non potrà essere istanziata; il costruttore di una classe astratta può essere invocato soltanto dai costruttori delle sottoclassi (con *super*) per questo può avere senso dichiarare i costruttori della classe astratta come *protected*
- Tutte le sottoclassi sono obbligate a fornire una implementazione del/dei metodi dichiarati abstract

Perchè usare classi astratte e non interfacce? Le classi astratte possono avere delle variabili di stato. Per i pattern che abbiamo visto in cui la radice era una interfaccia, se le implementazioni complete hanno bisogno di una stessa struttura dati, sarò costretto ad implementare più volte lo stesso codice degenerando nell'antipattern **duplicated code**.

Classi astratte e interfacce hanno quindi due principi di design diversi:

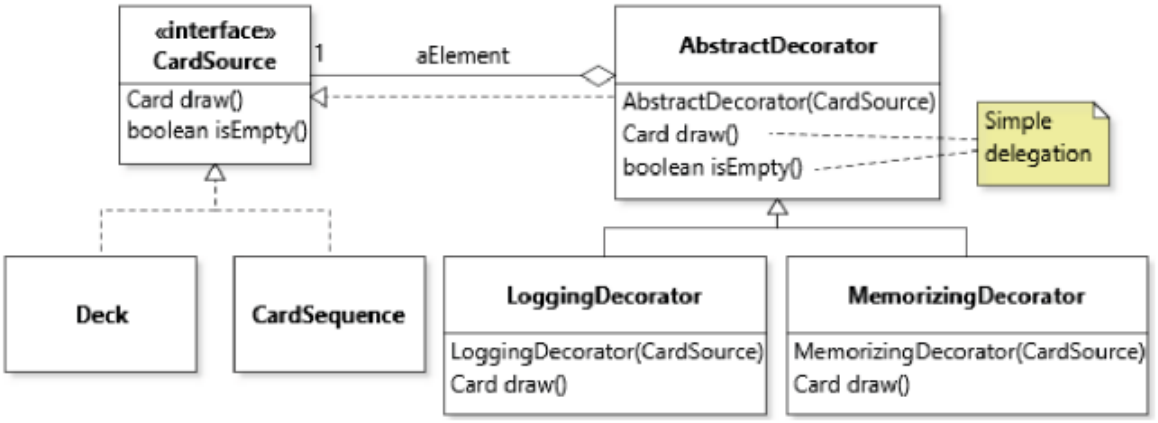
- L'interfaccia è un contratto: obbliga a implementare dei behaviour
- La classe astratta è un factoring: fattorizzo in una classe tutto ciò che è comune ai sottotipi, mentre nei sottotipi implemento tutto ciò che è specifico

Rivisitazione del pattern Decorator

Quando un design coinvolge più tipi di *Decorator* può essere utile fare un refactoring introducendo una classe astratta che viene ereditata da ognuno di essi, per ridurre la ridondanza di codice.

In particolare è possibile inserire il riferimento all'oggetto decorato a livello della superclasse astratta ed eventuali metodi con implementazioni comuni.

Esempio di rivisitazione del pattern con classe astratta:



Tra l'oggetto decorato (in questo caso un qualsiasi oggetto che implementa l'interface CardSource) e i Decorator si pone una classe astratta *AbstractDecorator* che dichiara un campo che referenzia l'oggetto aggregato da decorare, e i metodi da implementare come forma di delega.

Il campo che referenzia l'oggetto decorato può essere dichiarato anche *private*, poiché i *ConcreteDecorator* normalmente devono avere accesso ad esso solo mediante i metodi dell'interfaccia *Component*.

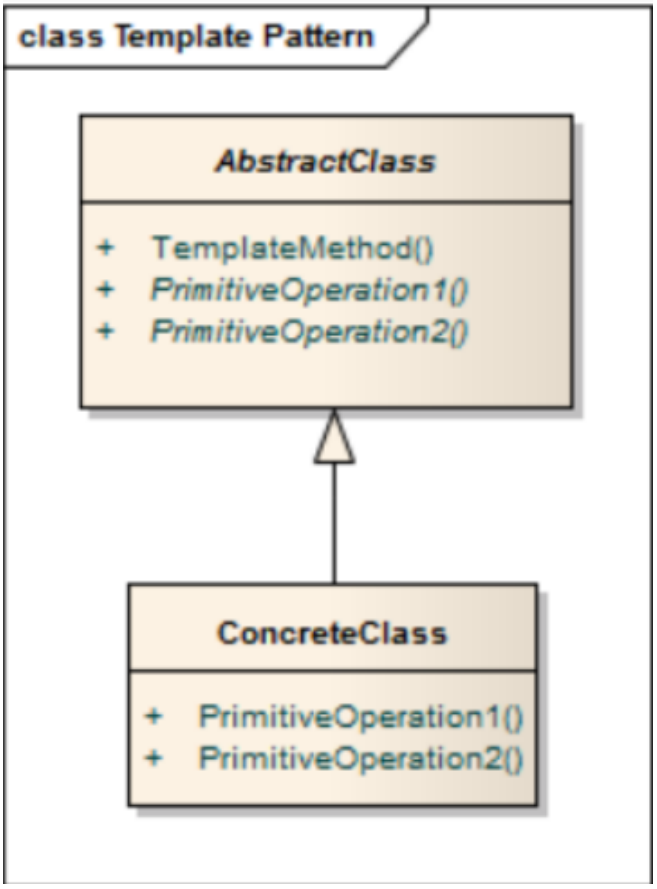
La stessa rivisitazione può essere applicata ad esempio al pattern Command sostituendo l'interfaccia con una classe astratta.

Design Pattern: Template Method

L'utilizzo delle classi astratte porta alla nascita di un vero e proprio pattern **comportamentale** che si basa su 3 principi fondamentali:

- Fattorizzare tutto ciò che è comune alle sottoclassi in una superclasse astratta, ad esempio le variabili di stato
- Implementare un metodo di **Template** per i comportamenti comuni delle sottoclassi e dichiararlo **final** per impedirne l'overriding
- Dichiarare abstract i **metodi primitivi** specifici che dovranno essere implementati dalle sottoclassi; possibile e concettualmente preferibile usare come specificatore d'accesso *protected* poiché il client potrà invocare solo le implementazioni effettive delle sottoclassi.
- Creare un **metodo concreto hook** che può essere eventualmente sovrascritto implementando logica aggiuntiva

Esempio di schema del pattenr:



Come visto la keyword **final** che permette di dichiarare dei campi costanti, se utilizzata per i metodi permette di impedire l'overriding, ma può essere anche usata per dichiarare classi, impedendone l'estensione tramite ereditarietà.

Principio di sostituzione di Liskov

L'ereditarietà è un meccanismo di estensione di funzionalità e riutilizzo di codice, ciò significa che non dovebbe essere utilizzato per restringere i comportamenti di una superclasse, né quando non esiste una relazione di tipo tra i sottotipi e la superclasse.

In un articolo del 1987 Barbara Liskov introdusse il cosiddetto **principio di sostituzione di Liskov** (**LSP**), il quale afferma che le sottoclassi non dovrebbero restringere ciò che un client può fare con una superclasse. In particolare nell'articolo vengono enunciate una serie di euristiche secondo cui i metodi di una sottoclasse non devono:

- Avere precondizioni più ristrette
- Avere postcondizioni meno restrittive
- Accettare parametri di tipi più specifici
- Rendere i metodi meno accessibili
- Lanciare più eccezioni di tipo *checked*
- Avere un tipo di ritorno meno specifico

8. Inversione del Controllo

L'**inversione del controllo** è un'idea del software design che consente di invertire il flusso di controllo che di solito va dal chiamante al chiamato per ottenere maggiore separazione delle responsabilità e diminuzione del coupling.

Un tipico contesto di utilizzo dell'inversione del controllo è nello sviluppo di interfacce grafiche.
In generale una situazione concreta che motiva l'utilizzo di questa tecnica è la necessità di mantenere consistente lo stato di un certo numero di oggetti senza produrre situazioni di forte dipendenza o elevato accoppiamento.

Ad esempio in un IDE come Eclipse quando un utente cambia qualche elemento nel codice sorgente, tale cambiamento deve essere visibile in tutte le possibili viste, ossia vi è un problema di *sincronizzazione* di oggetti diversi ma che devono essere consistenti tra loro.

Per implementare l'inversione del controllo viene usato principalmente lo stile architetturale **Model View Controller** (**MVC**) che fornisce tre tipi di astrazioni:

- **Model**: astrazione che mantiene la copia unica del dato di interesse
- **View**: astrazione che rappresenta una vista del dato
- **Controller**: astrazione che implementa la funzionalità necessaria per cambiare quel dato

Non esiste solo MVC, in realtà esistono molteplici modi in letteratura per implementare questa idea.

Design Pattern: Observer

L'idea MVC viene concretizzata in questo design pattern, che consente di memorizzare dei dati di interesse in un oggetto specializzato e permettere ad altri oggetti di **osservare** quel dato.

L'oggetto incaricato di mantenere il dato è un'istanza di *Model* (o **Subject**), mentre gli oggetti interessati ad essere aggiornati sui cambiamenti sono istanze di classi che implementano un'interface *Observer*.

Il *Model* deve fornire un metodo di interfaccia per aggiungere (o rimuovere) degli *Observer* dalla sua collezione: questa operazione prende il nome di **registrazione** degli observer.
Tale registrazione può essere fatta anche a run-time.
Un model sa di poter essere "osservato" ma la sua implementazione non dipende da alcun observer concreto.

Per venire a conoscenza del cambiamento di un dato gli *Observer* hanno un metodo di interfaccia *update()*, detto metodo di **callback**, proprio per il meccanismo di inversione del controllo: infatti non sono gli observer a chiamare il model, ma essi attendono che sia il model a chiamarli.
Questa idea prende il nome di **principio di Hollywood**: "Don't call us, we'll call you".

Il model non dice agli observer cosa fare ma li informa sull'avvenimento di un dato cambiamento per il quale essi sono registrati come osservatori/ascoltatori (listener) con un metodo specifico (es. *notifyObservers()*). Il metodo viene solitamente implementato con un ciclo che itera sugli observer registrati e invoca il metodo di aggiornamento.
Due possibili strategie su quando notificare gli observer:

- Inserire una chiamata al metodo di notifica in ogni metodo che può cambiare lo stato; in questo caso il metodo di notifica può essere dichiarato *private*
- In alternativa può essere chi modifica il model a decidere se una modifica è da propagare agli observer; questa soluzione è, però, meno significativa, ha senso solo quando si istanzia un oggetto complesso; se il client non fa il trigger, il model non notificherà gli observer.

La prima strategia non è ideale quando il model contiene una grande collezione di dati che, dovendo essere aggiornati frequentemente, richiederebbero frequenti notifiche agli observer con conseguente degrado delle prestazioni.
In queste situazioni preferire una strategia *silente* in cui gli observer vengono notificati solo al termine di una serie di operazioni.

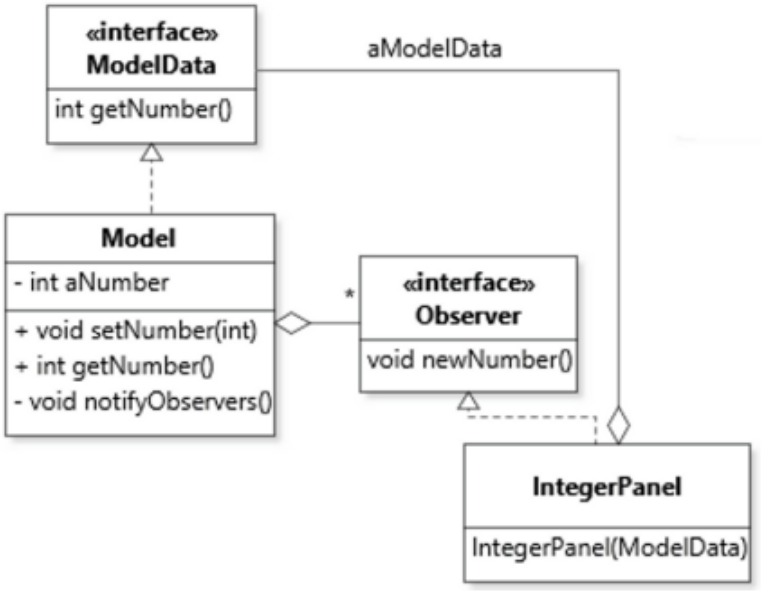
Flusso di dati tra Model e Observer

Come fanno gli observer ad avere accesso alle informazioni di cui hanno bisogno dal model? Due possibili strategie:

- Rendere l'informazione di interesse disponibile attraverso uno o più parametri della callback → **push data-flow strategy**; questa strategia però richiede che il *Model* conosca il tipo di dato di cui l'*Observer* è interessato
- Gli *Observer* effettuano delle query definite dal *Model* → **pull data-flow strategy**; in questo caso gli *Observer* devono possedere un riferimento al model (es. in fase di inizializzazione)

Possibile utilizzare il **principio di segregazione delle interfacce** anteponendo un'interfaccia tra i *ConcreteObserver* e i *ConcreteModel* che permetta di fornire agli observer solo un sottoinsieme dei metodi messi a disposizione del Model, riducendo quindi il coupling tra le due classi.

Schema di esempio di utilizzo del pattern:



Programmazione orientata agli eventi

Paradigma di programmazione che si basa proprio sul meccanismo dell'inversione del controllo: il *model* rappresenta la sorgente degli **eventi** e gli observer sono gli **Handler** che gestiscono tali eventi.

Un evento è ad esempio un cambiamento di stato di interesse per degli handler che vi si registrano.
Un particolare tipo di observer è l'**Adapter**, il quale implementa dei comportamenti di tipo *doNothing*, ossia non esegue alcuna operazione quando avvengono certi eventi.

Sviluppo di interfacce grafiche

In Java, così come in altri linguaggi, lo sviluppo di GUI si basa proprio su questo design pattern.
Il framework Java Swing (estensione del framework AWT scritta completamente in Java) sfrutta proprio questi principi.

Nei framework GUI le componenti grafiche sono oggetti che possono emettere **eventi** in modo asincrono. Le componenti sono estendibili e modificabili.

La libreria Swing fa un forte uso dello stile MVC per disaccoppiare i dati e i controlli della UI attraverso i quali vengono mostrati.
Le componenti sono associate a dei modelli, specificati tramite delle *interface* ed è possibile istanziare degli **event listener** associati agli oggetti per catturare determinati eventi.

Per gestire gli eventi occorre:

- Definire un handler per l'evento, ossia una classe che gestisce tale evento
- Istanziare l'handler
- Registrare l'handler

Quasi tutti i framework grafici, come Swing, gestiscono la vista grafica in maniera simile alla renderizzazione dei documenti HTML in un browser, ossia con una struttura ad albero che prende il nome di **document object model** (**DOM**).
Per passare dalla vista gerarchica a quella grafica si usa il concetto di **layout manager**.
Il mapping tra le due viste può avvenire anche tramite ad esempio XML.
Ogni *widget* dell'albero (foglia e non) è potenzialmente sorgente di moltissimi eventi.

Un esempio di oggetto che modella eventi è *ActionEvent*, un evento semantico che indica che una componente ha generato un'azione.

Posso quindi interagire con un oggetto ActionEvent per ottenere delle informazioni usando i suoi metodi, come *getActionCommand()*, che restituisce la stringa del comando associato all'azione.

Ogni volta che, ad esempio un Button, viene premuto, genera un ActionEvent e lo lancia consegnandolo a tutti i metodi callback registrati.

Design Pattern: Publish - Subscribe

Una variante del pattern *Observer* è il design pattern **publish-subscribe**. Si tratta di uno stile architetturale per la comunicazione asincrona fra diversi processi o oggetti.
Tale pattern può essere considerato un **middleware**, ossia un intermediario tra diverse componenti software.

La differenza principale con l'observer è che in questo caso né il model, né gli observer hanno conoscenza dell'altro ma comunicano usando un *dispatcher/broker*.
Il *publisher* "pubblica" il proprio messaggio ad un dispatcher, i *subscriber* dovranno quindi rivolgersi ad esso "sottoscrivendosi" per la ricezione di uno o più specifici messaggi.
Ogni qual volta il dispatcher riceve un messaggio dal *publisher* lo inoltrerà a tutti i *subscriber* interessati a quel messaggio.

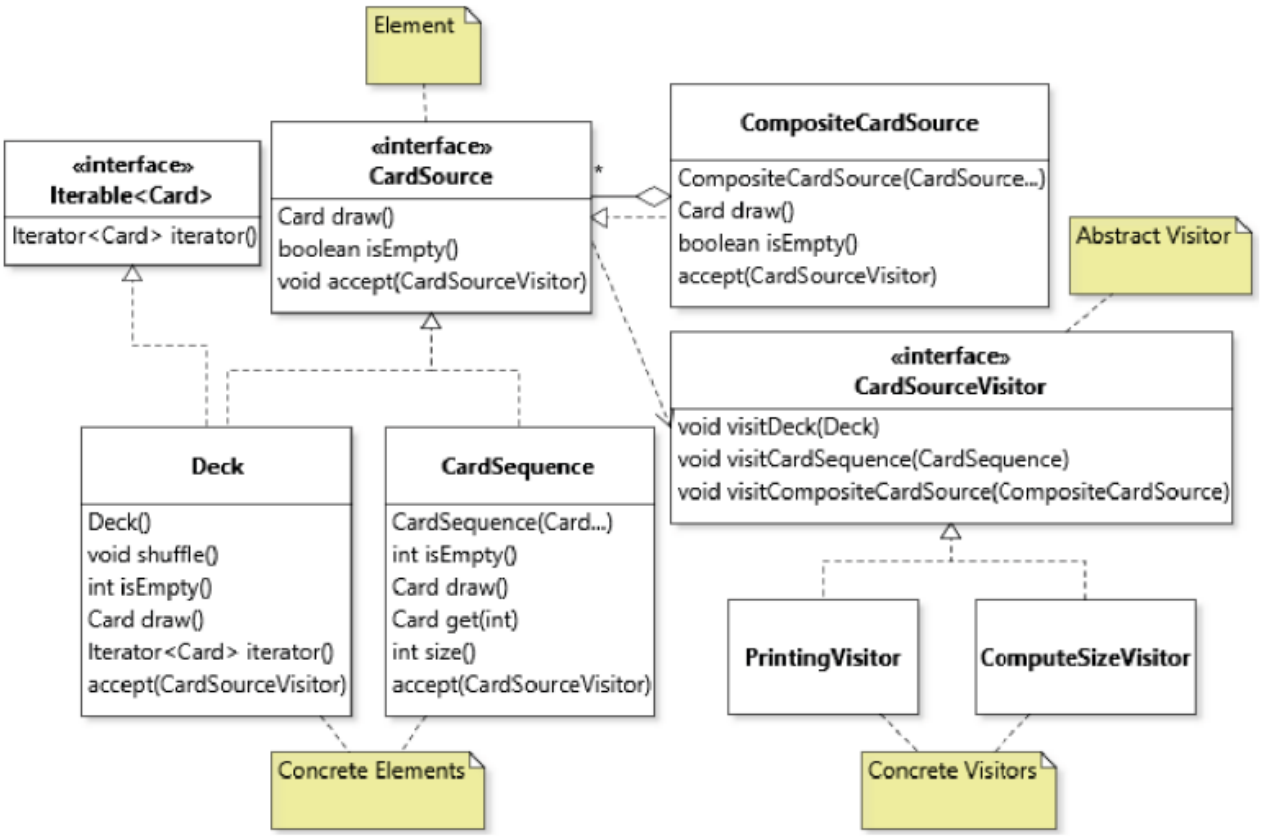
Un esempio di applicazione di questo pattern è per la creazione di newsletter.

Design Pattern: Visitor

Il contesto di utilizzo di questo pattern è la necessità di supportare delle operazioni su una collezione di elementi di una struttura senza alterare l'interfaccia di questi ultimi.

Consideriamo una struttura costituita da un insieme eterogeneo di oggetti sui quali vogliamo applicare la stessa operazione, implementata in modo diverso per ogni classe di oggetto.

Esempio di applicazione del pattern:



Gli elementi principali del template sono:

- **Visitor** (o *AbstractVisitor*): interfaccia che descrive oggetti in grado di "visitare" oggetti di una classe di interesse in una struttura; dichiara dei metodi di visita per ogni tipo di oggetto contenuto nella collezione.
- **ConcreteVisitor**: classe che implementa l'interfaccia *Visitor* e che rappresenta un visitatore concreto in grado di visitare i diversi tipi di oggetti e implementare un certo comportamento per ognuno di essi
- **Element**: rappresenta un elemento generico della struttura che verrà visitata; implementa un'interfaccia **Visitable** che dichiara un metodo per accettare dei *Visitor*: *accept(Visitor)*; tale metodo accetterà il visitatore invocando il metodo di visita specifico per quel tipo di oggetto passando come parametro sé stesso.
- **ConcreteElement**: oggetto concreto della classe specifica, contenuto nella collezione eterogenea da visitare

Come nel pattern *Observer* anche in questo caso abbiamo dei metodi di **callback**: i metodi *visit* vengono infatti chiamati all'atto dell'accettazione da parte degli *Element* di tipo *Visitable*.

Per evitare l'antipattern **duplicated code** possiamo utilizzare una classe astratta per i *visitor* (continuando ad implementare l'interfaccia) che implementi un metodo di attraversamento.

Vantaggi e svantaggi del pattern *Visitor*:

- Se la logica di un'operazione cambia basta apportare modifiche solo al visitatore
- Aggiungere un nuovo elemento al sistema è semplice, bisognerà però modificare l'interfaccia Visitor e dell'implementazione dei visitatori
- Uno svantaggio è che bisogna conoscere a priori i tipi di ritorno dei metodi di visita
- Un altro svantaggio è la verbosità del pattern: i *ConcreteElement* devono implementare un metodo che permetta di accettare visitatori, mentre nei *ConcreteVisitor* si dovrà implementare un metodo di visita per ciascun tipo attraversato

Alcuni di questi problemi si possono arginare facendo uso della Reflection, in modo tale che i *ConcreteElement* da visitare non siano costretti a implementare una particolare interfaccia e rendendo quindi non necessaria l'interface *Visitable*.

Si consiglia l'utilizzo di questo pattern quando:

- Si vuole operare in modo simile su oggetti diversi in una collection
- Si vuole separare delle operazioni dalle classi specifiche e poter cambiare tali comportamenti dinamicamente senza modificare le classi stesse
- Si ha una gerarchia di oggetti nota a priori e con bassa probabilità di modifiche ma con forte probabilità di aggiunta di nuove operazioni in futuro

Il pattern risulta poco utilizzabile in situazioni in cui i tipi vengono modificati di frequente o quando i comportamenti da implementare sono correlati alla singola istanza di oggetti e non all'intera gerarchia, poiché il visitor è utilizzato per defiire comportamenti applicati all'intera gerarchia.

9. Functional Design - Programmazione funzionale

Stile di design che tratta il concetto di funzione e la possibilità di scrivere funzioni che possono accettare come parametri altre funzioni.

Così come la programmazione ad eventi è conciliabile con la programmazione OO, anche lo stile funzionale è integrabile in un sistema complesso dove alcuni parti si prestano meglio per essere realizzati con questo stile.
Esistono però anche interi sistemi che sono pensati completamente per essere di tipo funzionale.

First-Class Functions

Ci sono delle situazioni in cui l'uso di oggetti per realizzare delle soluzioni di design può essere eccessivamente artificioso: ad esempio nel caso di un algoritmo di Sort di una collezione, esso richiederà come parametro anche un oggetto Comparator e siamo costretti a creare classi che implementano questa interface, che prevedono esclusivamente un metodo di comparazione tra due oggetti: una vera e propria funzione.

Da un punto di vista del design il metodo sort dovrebbe ricevere come parametro un comportamento e non un riferimento ad un oggetto.

Per permettere il passaggio di funzioni come parametro ad altre funzioni è necessario che il linguaggio di programmazione supporti le cosiddette **first-class functions**: la possibilità di trattare le funzioni come dei valori, in modo da poterle passare come parametri, memorizzarle in variabili e usarle come valore di ritorno di altre funzioni.

Le funzioni che accettano altre funzioni come parametro sono dette **higher-order functions**.
Usare funzioni di tipo higher-order non significa che l'intero design dell'applicazione diventa funzionale.

Functional Interfaces, Lambda Expressions e Method References

A partire da Java 8 esistono 3 meccanismi per utilizzare le first-class function.

Functional Interfaces

Una **functional interface** è un tipo di interfaccia che dichiara un singolo metodo astratto. Ad esempio immaginiamo di voler definire un'interfaccia per filtrare una collezione.

```
public interface Filter {
        boolean accept(Card pCard);
```

```
}
```

A partire da questa interface possiamo dichiarare classi (anche anonime) che la implementano.
Le functional interface possono essere viste come un tipo di funzione, in quanto non hanno stato e implementano un singolo metodo.

A partire da Java 8 è possibile definire all'interno delle interface dei metodi **static** (già definiti e non sovrascrivibili nelle classi che implementano l'interface) e **default** (già definiti ma con possibilità di overriding). Per definizione questi metodi NON sono astratti.
Per essere una functional interface essa deve però anche avere necessariamente UN solo metodo astratto.

Es. l'interface Comparator<T> definisce molti metodi static e default e un singolo metodo astratto *int compare(T, T)* che accetta due parametri qualsiasi T.
Si può dire che Comparator è una functional interface che definisce un tipo di funzione:
(T, T) → int
Il package java.util.function fornisce una libreria di functional interface per i tipi più comuni di funzione.

Lambda Expression

Anche con le functional interface siamo costretti a creare delle classi (seppur anonime) per implementare una singola funzione.
Le **lambda expression** sono una forma compatta per esprimere dei comportamenti funzionali.

Le lambda expression sono funzioni senza alcun nome che vengono chiamate in maniera polimorfica attraverso il nome del metodo in cui sono passate come argomento.

La sintassi di una lambda expression è composta da tre parti:

- Una lista di parametri, se presente, altrimenti si aprono e chiudono le parentesi ()
- Una freccia →
- Un corpo della funzione: se accetta più di una istruzione si usa un blocco di parentesi graffe {}

Questa sintassi è molto più compatta rispetto alla definizione di una classe anonima e rende più visibile il passaggio di un comportamento come parametro, piuttosto che di un dato (oggetto).

Non è necessario usare una espressione *return* nel corpo della lambda expression poiché questo è implicito, in quanto si tratta di un parametro di un metodo.

Le lambda expression sono controllate dal compilatore e convertite in function object attraverso un processo di inferenza: quando il compilatore vede una lambda expression prova a matcharla con una functional interface.
Il compilatore cerca il tipo della variabile a cui la lambda è assegnata per verificare che:

- Il tipo di variabile sia una functional interface
- I tipi dei parametri siano compatibili con quelli della functional interface; non è necessario esplicitarli nella lambda expression poiché sono già codificati nel metodo astratto della corrispondente functional interface
- Il tipo del valore di ritorno dal corpo della funzione sia compatibile con uno dei metodi astratti della functional interface

Essenzialmente quindi le lambda expression sono una sintassi più veloce e semplice per istanziare functional interface.

Method Reference

Permettono di riutilizzare i metodi di una classe per realizzare una funzione: si isola così un metodo dal contesto e lo si utilizza per avvalorare una funzione.

La sintassi di una method reference è composta dal nome della classe seguita da :: e infine il nome del metodo referenziato.

Buona norma utilizzare le lambda expression ogni volta che si può, stare invece molto attenti nell'utilizzo dei method reference perchè è possibile fare riferimento anche a metodi non statici.

Principali contesti di utilizzo del Functional Design

Funzioni per comporre behavior

Le first-class function rendono semplice la definizione di piccoli pezzi di behavior, ad esempio per operazioni di confronto/filtro di oggetti in una struttura dati complessa.
In questo modo è possibile comporre più livelli di comparazione/filtro molto facilmente.

Funzioni come sorgenti di dati (Functions as a data source)

Sfruttare un flusso di informazioni filtrate che diventano sorgente di dati.

Classi di problemi di Map Reduce

Quando si ha un flusso di dati che viene aggiornato continuamente e mappo i dati su alcune caratteristiche e poi li riduco a pochi indici di sintesi.

Pattern Strategy con design funzionale

Invece di creare classi che implementano una stessa interface è possibile usare le first-class functions che definiscono behavior in maniera più compatta, sia nel caso di lambda expression che nel caso di method reference.

Per strategie più elaborate è possibile definire una collezione di strategie in una classe di utility e usare delle method reference per selezionarle.

Pattern Command con design funzionale

In maniera simile al pattern *Strategy*, anche il pattern *Command* può beneficiare di una parametrizzazione con dei behavior grazie alle first-class function.
Possiamo infatti parametrizzare il comportamento di una singola classe *ConcreteCommand* definendo un campo che memorizza una funzione chiamata all'esecuzione di un command.

Data Processing in stile funzionale

Lo stile funzionale fornisce un ottimo supporto per l'applicazione di trasformazioni a grandi sequenze di dati.
Le higher-order function implementano le strategie di data-processing generali e sfruttano le first-class function parametrizzate per un particolare contesto.

Concetto di **Data as a stream**: uno *stream* è una sequenza di dati, e si differenzia da una collection perchè rappresenta un flusso, i dati non devono essere precedentemente aggiunti per esistere in essi, come invece avviene nelle collection.
Inoltre gli stream possono avere tecnicamente un numero infinito di dati.

A differenza delle Collection, introdotte prima in Java prima del supporto alle first-class function, a partire da Java 8 esistono le Stream API, le quali forniscono numerosi strumenti per la gestione di

stream di dati supportando anche il design funzionale.

15. Cicli di vita del software

Un modello di ciclo di vita del software rappresenta l'insieme delle attività e degli artefatti necessari per lo sviluppo di un sistema software.

Molti modelli si concentrano sulle attività di sviluppo software e sono pertanto detti **activity centered**. Un'altra possibile vista è quella relativa agli artefatti creati da queste attività, detta **entity centered**.

Forma dello sviluppo del software

L'esperienza industriale ci dimostra che un artefatto è completamente progettato e pronto alla realizzazione solo quando esiste un processo produttivo che ne rende possibile la realizzazione e commercializzazione a costi industriali compatibili con il target di mercato che ci si è posti.
C'è bisogno quindi di un **processo produttivo**.

Nell'ingegneria del software si è pensato di dover dare la stessa impronta industriale presente in altri campi ingegneristici (es. ingegneria industriale).
Ma produrre software è profondamente diverso poiché è tutta attività di tipo progettuale.

Modello di ciclo di vita del software

Modello prescrittivo o descrittivo di come un'organizzazione si struttura per lo sviluppo del software.
Prescrittivo significa che un'azienda si dà un suo ciclo di vita del software e lo comunica ai team.
Descrittivo se non c'è un modello predefinito ma lascio ai team la scelta e derivo un modello.

Si pensò inizialmente di andare per analogia con altri settori ingegneristici: processi lineari.
I primi esempi di modello di ciclo di vita del software furono quindi lineari → in ingegneria del software si parla di modello **waterfall**, o a cascata.
Costituito da un insieme di fasi, ognuna delle quali è concepita come propedeutica alle altre.

Negli anni '90 iniziò a svilupparsi la necessità di non dare una forma, ma degli strumenti per permettere alle aziende di definire la propria forma di processo.
Tutti gli approcci sfociarono in una serie di standard che non definiscono la forma del ciclo di vita del software, ma aiutano le aziende a darsi un proprio ciclo di vita attraverso delle linee guida, delle euristiche.
L'idea di fondo è sempre che nelle attività di sviluppo un'organizzazione deve produrre e consumare artefatti.

Standard IEEE 1074

Nel 1997 nasce lo standard **IEEE 1074**, aggiornato periodicamente fino al 2006, e successivamente sostituito da nuovi standard ISO IEC.

Questo standard descrive l'insieme di attività e processi obbligatori per lo sviluppo e la manutenzione di software di un'azienda di qualità.

Questo standard elencava 17 processi, raggruppati in **process groups** a diversi livelli di astrazione e forniva linee guida per ciascuno di quei processi e una serie di dipendenze fra i processi.
Ognuno di questi processi è a sua volta suddiviso in **attività** e **task** e consuma **tempo**, **denaro** e **risorse umane** per la produzione di **artefatti** (work product).

Si passò da un modello unico (anni '70) ad un modello tailored: costruire un proprio ciclo di vita basato su quelle linee guida.

Questo approccio di fornire alle organizzazioni gli strumenti per organizzare il proprio modello di ciclo di vita è stato molto in voga dagli anni '90.

Ci fu poi il tentativo di certificare il livello di qualità di un'organizzazione (non di un progetto o di un software). Nacque il **Capability Maturity Model** (**CMM**): era lo schema di valutazione del livello di maturità di un'organizzazione che sviluppa sistemi software.
A questo schema era associato un processo di certificazione: un'azienda poteva chiedere di essere valutata rispetto al CMM e ottenere un grado.

Il CMM classificava le aziende su una scala di 5 livelli:

1. **Initial**: identifica un'azienda *caotica*, ossia basata esclusivamente sulle skill delle risorse individuali e non sull'organizzazione e definizione delle attività di sviluppo; il cliente non ha modo di interagire con il project management.
2. **Repeatable**: organizzazione che ha un modello di ciclo di vita del software ben definito e processi basilari di project management per tracciare costi e schedule; in questo caso il cliente può interagire con l'organizzazione in determinati momenti, come ad esempio durante le review e l'acceptance test, prima della consegna.
3. **Defined**: l'organizzazione usa un modello di ciclo di vita ben documentato e conosciuto da tutti i reparti aziendali; il cliente conosce tale modello e l'eventuale modello personalizzato creato per lo specifico progetto.
4. **Managed**: aziende che collezionano dati costantemente e usano processi di quality management e gestione dei processi quantitativi; il cliente può essere informato sui rischi del progetto e conosce le metriche usate dall'organizzazione.
5. **Optimized**: i dati misurati dall'organizzazione vengono usati come meccanismo di feedback per migliorare il modello di ciclo di vita utilizzato nel corso del tempo; clienti, project manager e sviluppatori lavorano insieme comunicando costantemente durante tutto lo sviluppo di un progetto.

L'IEEE non ha mai previsto schemi di certificazioni (linee guida non prescrizioni) mentre il CMM è nato per essere oggetto di certificazione.
Lo standard di riferimento ad oggi è l'ISO/IEC IEEE 24774 firmato congiuntamente da ISO/IEC e IEEE. L'ultima edizione è del 2021.

La novità rispetto al passato (fin dalla prima versione del 2010) è l'introduzione di controllo e constraint: ogni attività e processo può avere associati dei controlli e dei constraint.
Sulla base di queste raccomandazioni le aziende possono creare i propri processi.

Modelli di ciclo di vita

Modello waterfall

Descritto la prima volta da Royce nel 1970, si tratta di un modello **activity-centered** che si basa prescrive una sequenza di attività.
Questo modello è l'ideale dal punto di vista del project management perchè è un modello additivo, ossia rende molto semplice prevedere il tempo totale necessario per l'intero progetto in quanto somma delle durate delle singole attività, lo stesso vale per le risorse.

Il problema principale di questo modello è che per funzionare in maniera additiva gli output di ogni box (input del successivo) devono fluire una sola volta senza cambiamenti successivi.
Se il modello ammette feedback perde l'additività: non è più prevedibile.

Per molto tempo si è pensato di lavorare quindi sui punti di transizione del modello: effettuare un check di ogni attività per consolidarla ed evitare successivi cambiamenti.

V-Model

Variante del modello *waterfall* che unisce il concetto di cascata e cascata inversa, associando ad ogni attività di sviluppo un'attività di verifica.
Il ramo sinistro comprende le fasi di **forward engineering**, quello destro una cascata inversa con attività di **Verification & Validation**.
Anche questo modello però soffre delle stesse debolezze del modello da cui deriva: ogni attività può essere svolta soltanto al termine dell'attività precedente e ciò è applicabile solo in contesti in cui i requisiti sono fortemente stabili ed affidabili.
Se l'additività non funziona l'alternativa è passare a modelli di tipo incrementale e iterativo.

Modelli iterativi

Modello a spirale di Boehm

Basato sull'idea evolutiva attraverso prototipazione e la risoluzione dei rischi in maniera incrementale.
Ogni ciclo della spirale è incrementale e comprende attività di:

- Determinazione degli obiettivi, specifica dei constraint e generazione di alternative
- Identificazione e valutazione di rischi, con eventuale risoluzione
- Sviluppo e verifica
- Pianificazione del ciclo successivo

Così come per il modello waterfall esistono varianti anche del modello a spirale.

Leggi di Lehman sull'evoluzione del software

Manny Lehman capì che il modello waterfall non può funzionare perchè è irrealistico.
Egli introdusse il concetto di **software evolution**, ossia la possibilità di continuare lo sviluppo di un sistema anche dopo la sua release sul mercato.
Inoltre secondo Lehman esistono tre tipi di sistemi:

- **S-Program**: sono scritti esattamente secondo le specifiche
- **P-Program**: sono scritti per implementare procedure che determinano completamente ciò che il programma può fare
- **E-Program**: scritti per compiere attività del mondo reale, quindi fortemente legati all'ambiente in cui sono utilizzati e necessitano un adattamento continui dei requisiti; sono i sistemi di nostro interesse

Egli scrisse inoltre nel corso degli anni una serie di leggi (8) sull'evoluzione del software:

1. **Evoluzione continua** (*Continuing change*): ogni sistema utile è soggetto ad evoluzione a meno che non lo si voglia far diventare via via meno utile fino alla morte (obsolescenza)
2. **Entropia crescente** (*Increasing complexity*): man mano che si cambia il sistema la sua struttura degrada perchè aumentano entropia e complessità; come l'entropia può essere corretto solo tramite investimenti per mantenere o ridurre questa complessità; fondamentale il **refactoring**
3. **Self regulation**: i processi organizzativi legati allo sviluppo del software tendono ad autoorganizzarsi; qualche anno prima fu introdotto il concetto che aggiungere risorse a un progetto in ritardo potrebbe aumentarne il ritardo.

Sviluppo Agile

Per **metodologia agile** o **sviluppo agile** si intende un insieme di metodi per lo sviluppo del software nati a partire dai primi anni 2000 fondati da un movimento di sviluppatori che scrisse e firmò il **Manifesto per lo sviluppo agile del software**.

Manifesto per lo sviluppo Agile di software

L'idea del manifesto è mettere al centro la risorsa umana, descrivendo 4 valori fondamentali:

- **Individui e interazioni** più che processi e strumenti
- **Software funzionante** più che documentazione esaustiva
- **Collaborazione col cliente** più che negoziazione dei contratti
- **Risposta al cambiamento** più che seguire un piano

Il manifesto ha trovato poi realizzazione in una serie di pratiche, ognuna applicabile o meno a seconda del contesto e delle necessità di un'azienda: una delle più famose è l'**extreme programming** (**XP**).
L'**XP** fa riferimento a una metodologia di sviluppo dove si enfatizza la scrittura di codice di qualità e la rapidità di risposta ai cambiamenti dei requisiti.
12 regole (o pratiche) alla base di Extreme Programming raggruppati in 4 aree:

- **Feedback a scala fine:
  - **Planning game**: riunioni di pianificazione tra manager, sviluppatori e clienti, una volta per iterazione, tipicamente 1 volta a settimana;
  - **Pair programming**: due programmatori per ogni workstation, un *driver* che scrive codice, un *navigator* che ragiona sull'approccio; tutto ciò per produrre codice di qualità migliore
  - **Test driven development**: usare test automatici (unit test e acceptance test) scritti prima di scrivere il codice
  - **Whole team**: il cliente deve essere presente e disponibile a fare verifiche
- **Processo continuo**
  - **Continuous Integration**: integrare continuamente i cambiamenti al codice per evitare ritardi nel ciclo del progetto per cause di integrazione
  - **Refactoring**: riscrivere il codice senza alterare le funzionalità esterne, cambiando l'architettura per renderlo più semplice e generico
  - **Small releases**: frequenti rilasci del sistema con introduzione di piccole funzionalità
- **Comprensione condivisa**
  - **Coding standard**: scegliere e usare un preciso standard di scrittura, una serie di regole concordate dal team di sviluppo
  - **Collective code ownership**: ognuno è responsabile di tutto il codice, chiunque coinvolto nel progetto contribuisce alla stesura
  - **Simple design**: approccio "semplice è meglio"; progettare al minimo e insieme al cliente
  - **System metaphor**: descrivere il sistema con una metafora
- **Benessere dei programmatori**
  - **Sustainable pace**: gli sviluppatori non dovrebbero lavorare più di 40 ore settimanali

In questo momento la nuova frontiera è il concetto di **DevOps**, metodologia agile che punta alla comunicazione, collaborazione e integrazione tra sviluppatori.
Anche DevOps ha come obiettivi il rilascio frequente del software, l'evoluzione continua incrementale e molte altre caratteristiche agili.