



## Riassunto Ingegneria del Software

Ingegneria del sw (Università di Bologna)



Scansiona per aprire su Studocu

## **1. ANALISI E PROGETTAZIONE ORIENTATA AGLI OGGETTI**

### **Contenuti**

#### **UML e pensare a oggetti**

UML è una notazione standard per programmi. Non si tratta di una OOA/D o un metodo, ma solo di un linguaggio di notazione dei diagrammi.

#### **Progettazione orientata agli oggetti (OOD): principi e pattern**

Che cosa sono le classi? Che cosa sono gli oggetti? Ecc... Queste sono le domande critiche nella progettazione di un sistema software. La metafora della progettazione OO classica, la progettazione guidata dalle responsabilità, affronta i problemi della progettazione OO dal seguente punto di vista: come devono essere assegnate le responsabilità (di conoscere, di fare e di collaborare) alle classi di oggetti? Vi sono inoltre alcune soluzioni valide a problemi di progettazione che possono essere espresse come "pratiche migliori", o pattern, ovvero coppie problema-soluzione che codificano dei principi di progettazione esemplari.

#### **Studi di caso**

Questa introduzione è presentata tramite studi di caso progressivi.

#### **Casi d'uso e storie utente**

L'OOD (e tutta la progettazione del software) è fortemente correlata all'attività dell'analisi dei requisiti. Questa attività comprende spesso la scrittura della modalità d'uso del software, per esempio sotto forma di casi d'uso oppure di storie utente.

#### **Sviluppo iterativo e modellazione agile**

L'analisi dei requisiti e l'OOA/D vanno presentate e svolte nel contesto di quale processo di sviluppo. In questo caso, come processo di sviluppo iterativo di esempio viene usato un approccio agile (leggero e flessibile) al Unified Process (UP).

#### **Altre capacità importanti**

La costruzione del software comporta numerose altre capacità e passi: per esempio una buona riuscita sono fondamentali nell'ingegneria dell'usabilità.

### **Obiettivo di apprendimento**

Una capacità critica nello sviluppo OO è quella di assegnare in modo abile responsabilità agli oggetti software.



### **Che cosa sono analisi e progettazione**

L'analisi enfatizza un'investigazione del problema dei requisiti, anziché di una soluzione. "Analisi" è un termine ampio, con più accezioni, come analisi dei requisiti (un'investigazione degli oggetti di dominio).

La progettazione enfatizza una soluzione concettuale (software o hardware) che soddisfa i requisiti, anziché la relativa implementazione. Nella progettazione vengono spesso esclusi dettagli di basso livello o "ovvi". Infine, i progetti devono essere implementati, e l'implementazione esprime il progetto realizzato vero e completo.

Come nel caso dell'analisi, questo termine ha più accezioni, come *progettazione orientata agli oggetti* o *progettazione di basi di dati*.

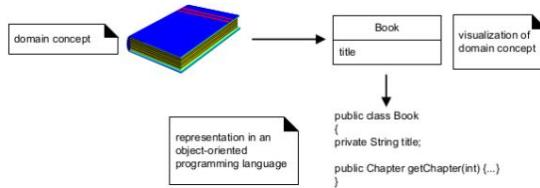
L'analisi e la progettazione possono essere riassunte nell'espressione *fare la cosa giusta (analisi)* e *fare la cosa giusta (progettazione)*.

### **Che cosa sono analisi e progettazione orientata agli oggetti**

Durante l'*analisi orientata agli oggetti*, c'è un'enfasi sull'identificazione e la descrizione degli oggetti, o i concetti, nel dominio del problema.

Durante la *progettazione orientata agli oggetti* l'enfasi è sulla definizione di oggetti software e del modo in cui questi collaborano per soddisfare i requisiti.

Analisi e progettazione, hanno dunque obiettivi diversi, che vengono perseguiti in modo diverso. Tuttavia analisi e progettazione sono attività fortemente sinergiche, che sono correlate fra loro e con le altre attività dello sviluppo del software.



### **Esempio**

Utilizziamo un semplice esempio, ovvero una partita di un gioco dei dadi in cui il software simula un giocatore che lancia due dati: se il totale è 7, ha vinto; altrimenti ha perso.

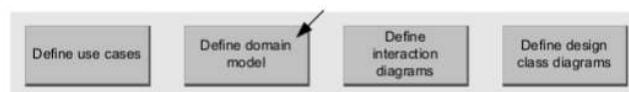
### **Definizione dei casi d'uso**



L'analisi dei requisiti può comprendere storie o scenari relativi al modo in cui l'applicazione viene utilizzata dalle persone; queste storie possono essere scritte come casi d'uso.

I casi d'uso non sono un elaborato orientato agli oggetti, ma semplicemente delle storie scritte.

**Gioca una partita di dati:** il Giocatore chiede di lanciare i dati. Il Sistema presenta il risultato: se il valore totale delle facce dei dati è sette, il Giocatore ha vinto; altrimenti ha perso.



### **Definizione di un modello di dominio**

L'analisi orientata agli oggetti è interessata alla creazione di una descrizione del dominio da un punto di vista ad oggetti. Vengono identificati i concetti, gli attributi e le associazioni considerati significativi.

Il risultato può essere espresso come un *modello di dominio* che mostra concetti o gli oggetti *significativi* del dominio.



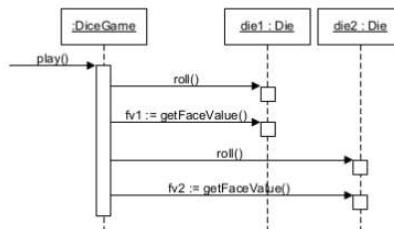
Si noti che un modello di dominio non è una descrizione di oggetti software, ma una visualizzazione dei concetti o del modello mentale di un dominio del mondo reale. Pertanto è stato anche chiamato modello concettuale a oggetti.

### Assegnare responsabilità agli oggetti e disegnare diagrammi di interazione



La progettazione orientata agli oggetti è interessata alla definizione di oggetti software, delle loro responsabilità e collaborazioni. Una notazione comune per illustrare queste collaborazioni è un diagramma di sequenza.

Esso mostra il flusso di messaggi tra oggetti e software, dunque l'invocazione di metodi.



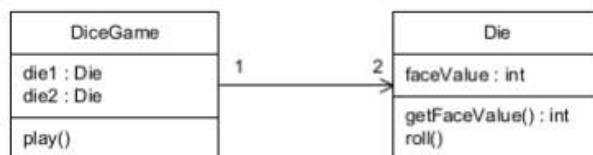
### Definizione dei diagrammi delle classi di progetto



Accanto a una vista dinamica delle collaborazioni tra oggetti, mostrata dai diagrammi di iterazione, è utile mostrare una *vista statica* delle definizioni di classi, mediante un diagramma delle classi di progetto. Questo mostra le classi software con i loro attributi.

Diversamente dal modello di dominio che illustra classi del mondo reale, questo diagramma mostra classi software.

I progetti e i linguaggi OO sono in grado di favorire un salto rappresentazione basso tra i componenti software e i nostri modelli mentali di un dominio, migrando la comprensione.



### Vantaggi

Disegnare e leggere UML implica un modo di lavoro più visivo, sfruttando le capacità del cervello di comprendere rapidamente simboli, unità e relazioni nelle notazioni.

## Storia

Un grande problema dei progetti software è che possono fallire. Il bisogno di cambiamento nell'informatica porta così alla richiesta di nuovi tipi di produzione software. Più la dimensione e la complessità del software crescono, più è probabile un fallimento nel progetto. Questo alla fine degli anni '60 porta alla cosiddetta "software crisis". Il dubbio era: lo sviluppo software è un'arte o una scienza? Ci fu una discussione sull'ingegneria del software vista come effettiva disciplina nel 1968 alla conferenza della NATO sull'ingegneria del software. All'inizio degli anni '70 era chiaro che lo sviluppo di software richiedesse più di un vago principio di informatica, esso necessita di strumenti analitici e descrittivi per garantire l'affidabilità degli artefatti creati. Si applicò dunque un approccio sistematico, disciplinato e quantificabile allo sviluppo, funzionamento e manutenzione dei software. Gruppi di persone iniziano a lavorare su multi-versioni dei software, si crearono inoltre principi ingegneristici per ottenere software affidabili e funzionanti su macchine reali.

Ingegneria del software → forma di ingegneria che applica i principi dell'informatica e matematica per raggiungere soluzioni economicamente vantaggiose per i problemi dei software.

Questo afferma dunque che l'ingegneria del software crea software di alta qualità in modo sistematico, controllato ed efficiente. Essa è diversa dalle altre discipline ingegneristiche a causa della natura intangibile dei software e dell'astratta natura delle operazioni. Essa cerca di integrare i principi matematici ed informatici con la pratica ingegneristica per produrre artefatti tangibili e fisici.

I sistemi software sono complessi, è difficile predirne il funzionamento senza prima crearli e testarli. Verranno utilizzati due strumenti chiave:

1. Astrazione → formulazione di idee generali per estrarre qualità comuni da specifici esempi. E' il meccanismo con cui la mente umana affronta la complessità. L'idea è definire dei concetti che aiutano a definire la complessità. Ci interessa creare mentalmente delle categorie/concetti per generalizzare dei ragionamenti. Questo aiuta a semplificare la realtà perché generalizza una categoria. L'intera storia dell'ingegneria del software è rappresentata da crescenti livelli di astrazione.
2. Modelli → rappresentazione di un sistema che risponda a come il sistema si sarebbe comportato per un dato insieme di domande. L'insieme delle domande definisce il mio punto di vista.

## Esempio di modello

Questo dipinto rappresenta una pipa, posso avere delle risposte utili alle mie domande solo in base al mio punto di vista. Se secondo il mio punto di vista voglio ad es. sapere che odore fa la pipa questo modello non è utile, ma se secondo il mio punto di vista voglio sapere che forma essa abbia allora questo modello si rivela utile.



## 2. ANALYSIS MODEL

Per realizzare un software si affronta un processo di operazioni dove si ha un elevato livello di astrazione. Questo processo può essere visto come una sequenza di attività che si abbassano man mano nel livello di astrazione fino ad arrivare al codice, anche se pure quello è una sorta di astrazione perché il computer non legge quello ma il codice assembly. Più il livello di astrazione è alto e più si tolgonon informazioni.

Questi diversi **livelli di astrazione** sono:

- Analisi
- Design
- Construction

Il processo software, inoltre, può essere visto come il raffinamento dei modelli:

- Mental model
- Analysis model
- Design model
- Construction model

→ I primi 2 rappresentano lo spazio del problema (capire cosa il programma deve fare), gli ultimi due lo spazio della soluzione (capire come fare).

**Obiettivi** del modello di analisi:

- Capire precisamente cosa è richiesto dal software in maniera non ambigua in quanto il linguaggio naturale presenta delle ambiguità;
- Comunicare questa comprensione ai membri del team di sviluppo e agli stakeholders;
- Definire un insieme di requisiti che possono essere validati una volta che il software è costruito.

Il modello di analisi necessita di artefatti/notazioni cioè documenti chiari detti **elaborati dei requisiti**:

- Documenti sui requisiti;
- Modello dei casi d'uso → Un insieme degli scenari tipici dell'utilizzo del sistema. Usato principalmente per i requisiti funzionali (comportamento).
- Specifiche supplementari → Essenzialmente tutto ciò che non rientra nei casi d'uso. Esso è principalmente per tutti i requisiti non funzionali, come prestazioni o licenze. E' anche il documento per registrare delle caratteristiche funzionali non espresse come casi d'uso, es. generazione di un report.
- Modello di dominio → Modello più importante e classico dell'analisi, esso usa i concetti significativi di un dominio. Può fungere da sorgente di ispirazione per progettare alcuni oggetti sw. Un modello di dominio è dunque una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio.  
→ Rappresentazione di classi concettuali del mondo reale, non di oggetti sw. Il termine NON indica un insieme di diagrammi che descrivono classi software, lo stato del dominio di un'architettura software o oggetti software con responsabilità.

Un modello di dominio è illustrato con un insieme di diagrammi delle classi in cui non sono definite operazioni. Esso fornisce un punto di vista concettuale e può mostrare:

- Oggetti di dominio o classi concettuali;
- Associazioni tra classi concettuali;
- Attributi di classi concettuali.

Per concludere, un modello di dominio è una visualizzazione di oggetti del mondo reale di un dominio di interesse, NON di oggetti sw o oggetti sw con responsabilità.

- Glossario → Definisce i termini significativi. Ha anche il ruolo di dizionario dei dati, che registra i requisiti dei dati, come regole di validazione, valori accettabili ecc. Il glossario può definire nel dettaglio qualsiasi elemento ad es. attributo di un oggetto, paramentro di un'operazione ecc.

- Regole di business (o di dominio) → Descrivono i requisiti o le politiche che trascendono un progetto software; sono richieste nel dominio o nel business.

→ Questi artefatti hanno un costo, c'è un trade-off tra tempo, risorse e costi.

I requisiti software sono i bisogni e le restrizioni di un prodotto software. La requirements engineering (RE) è la gestione sistematica dei requisiti, serve per definire cosa ci aspettiamo che faccia il programma e con quali restrizioni.

Le **aree di conoscenza** della software requirements sono:

- Elicitazione → “Tirare fuori” la conoscenza per capire cosa bisogna fare;
- Analisi → Come bisogna modellare;
- Specificazione;
- Validazione dei requisiti software → Vedere se sono stati rispettati.

#### Tipi di requisiti:

- Funzionali → Descrivono l'interazione tra il sistema ed il suo ambiente, è indipendente dall'implementazione. Essi sono indirizzati dal design del sistema, esprimono cosa dovrebbe fare il sistema.  
**Esempio:** Il tasto “=” mi dà il risultato nella calcolatrice.
- Non funzionali → Proprietà non misurabili del sistema e quindi non direttamente correlate agli aspetti funzionali. Sono indirizzati dall'architettura del sistema, esprimono come il sistema dovrebbe funzionare.  
**Esempio:** Rapidità, sicurezza, efficienza. Soddisfarli è spesso la cosa più complicata.

I requisiti inoltre sono divisi in categorie:

- Funzionalità → Caratteristiche funzionali, capacità, sicurezza.
- Usabilità → Fattori umani, help, documentazione.
- Affidabilità → Frequenza di fallimento, ripristinabilità, prevedibilità.
- Prestazioni → Tempo di risposta, throughput, precisione, disponibilità, uso di risorse.
- Sostenibilità → Adattabilità, manutenibilità, configurabilità.

È utile utilizzare liste di controllo per la copertura dei requisiti, per ridurre il rischio di non tenere conto di qualche importante aspetto del sistema. Alcuni di questi requisiti sono chiamati attributi/requisiti di qualità di un sistema e hanno una forte influenza sull'architettura di un sistema.

Per formalizzare i requisiti ci sono vari metodi. Nei processi di sviluppo strutturati un buon punto di inizio è l'ISO/IEC/IEEE – System and software engineering – Life cycle processes – Requirements engineering. I documenti principali sono: lo stakeholder requirements specification document (StRS) cioè il documento di specificazione dei requisiti per gli stakeholders, System requirements specification document (SyRS) cioè il documento di specifica dei requisiti, Software requirements specification document (SRS) cioè documento di specifica dei requisiti software.

### **3. SOFTWARE PROCESS MODEL**

Processo → insieme di attività coordinate che portano ad un obiettivo.

Software process → l'obiettivo è la produzione/sviluppo/evoluzione/mantenimento del software.

Si deve quindi pianificare, organizzare e far funzionare un progetto software rispettando determinati vincoli:

- Costo
- Tempo
- Risorse

Normali attività di un processo software:

- Specificazione
- Design
- Implementazione
- Validazione
- Evoluzione

Il software è intangibile, per questo bisogna produrre artefatti addizionali:

- Documenti/prototipi del design
- Rapporti
- Incontri per aggiornarsi sullo stato del progetto
- Rilevazioni/sondaggi dai clienti (es. livello di soddisfazione)

L'obiettivo del progetto software è rispettare vincoli di costo, tempo e di risorse. Inoltre, bisogna ottimizzare e verificare il progresso. Bisogna coordinare le attività legate a:

- Specifiche
- Progettazione
- Design
- Validazione
- Evoluzione

Queste attività devono dare risultati "tangibili", si usano documenti/artefatti addizionali per aiutare in questo compito. Ci saranno alla fine tanti documenti per diversi tipi di attività, ciò può dare luogo a processi complessi.

La sequenza giusta di attività non esiste, ci sono strade diverse per raggiungere l'obiettivo cioè diversi modelli di processo:

1. Modello a cascata → Vengono definite le attività e realizzate a sequenza cioè una dietro l'altra. C'è un livello di astrazione che si abbassa ad ogni livello.

Questo modello funziona solo se non si sbaglia mai nulla: quando si trova un errore, per esempio nella fase testing, magari c'è qualcosa di sbagliato nella fase coding, allora si deve tornare indietro a ritroso per poi scoprire che magari non era quella la fase sbagliata. Si torna indietro man mano fino a capire quale fase è stata sbagliata (**Esempio**: progettazione). → Questo processo è detto raffinamento ed è il processo opposto all'astrazione. Più l'errore è "in alto" nell'astrazione e più esso sarà grande, quindi il raffinamento da fare sarà anch'esso grande.

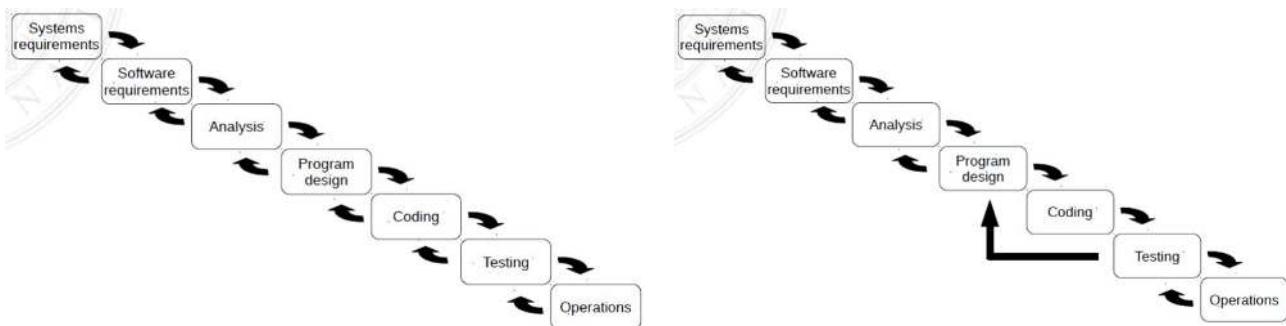
Pro:

- Modello facile da comprendere
- Divisione per gruppi di attività
- Identifica gli artefatti, i risultati e le pietre miliari delle attività

Contro:

- Modello irrealistico
- Consegna molto lontana nel tempo e non si vede mai il committente nel frattempo

- Ci si accorge tardi degli errori
- Difficile gestione dei cambiamenti
- Alto overhead



*Il modello a cascata è mai davvero esistito?* Nel suo riferimento più classico, questo modello viene descritto come un “fondamentalmente valido” ma viene suggerita una sua estensione per includere anche le iterazioni. Esso divenne tuttavia uno standard militare (DOD-STD-2167A).

2. **Modello a spirale** → Si introducono le iterazioni, questo per non accorgersi tardi degli errori, l'approccio è guidato dai rischi.

Si ha una ripetizione iterativa delle attività: si fanno dei prototipi per le singole attività e si va alla fase successiva solo se tutto funziona nel prototipo attuale.

Il lavoro è fatto “a fette” e alla fine di ogni ciclo c’è una revisione da parte del committente sui prototipi.

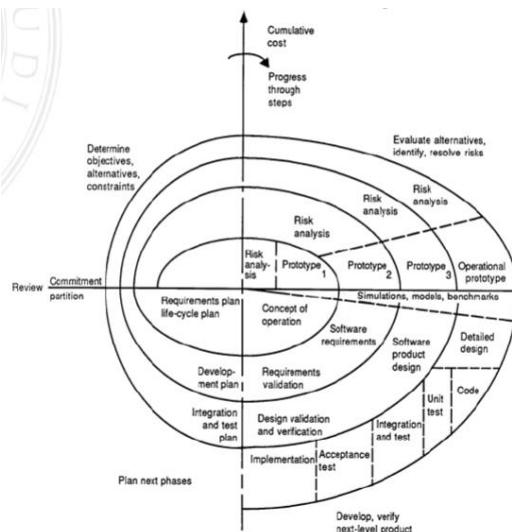
→ Si dà la priorità alle attività più difficili/rischiose, si usano dei modelli di valutazione del rischio. Per ogni ciclo si definiscono obiettivi, alternative e vincoli.

#### Pro:

- Buona visibilità
- Si usano definizioni dei rischi
- Modello iterativo ed incrementale
- Riflette la natura iterativa dello sviluppo software
- Assistenza al rischio

#### Contro:

- L’analisi dei rischi non è banale
- Modello complicato e rischio che la consegna si protragga più avanti nel tempo
- Alto overhead



### Sviluppo iterativo incrementale

Al contrario di ciò che avviene nel ciclo di vita sequenziale o “a cascata”, in questo tipo di sviluppo c’è fin da subito la programmazione ed il testing delle versioni parziali del sistema, in cicli che si ripetono. Si presume che lo sviluppo inizi prima che tutti i requisiti siano stati definiti in modo dettagliato; viene usato il feedback per chiarire e migliorare le specifiche in evoluzione del sistema.

Per chiarire i requisiti e il progetto ci si affida a passi di sviluppo brevi e rapidi, al feedback e all’adattamento. In confronto, il pensiero a cascata incoraggiava prima di iniziare la programmazione lo svolgimento di passi in cui effettuare un grande lavoro preliminare, prima sui requisiti e poi sulla progettazione. Gli studi dimostrano che al metodo a cascata sono correlate le percentuali di fallimento più elevate e dimostrano invece che i metodi iterativi sono associati a percentuali di successi e produttività più elevate, nonché a minori livelli di difetti.

Un processo per lo sviluppo software descrive un approccio alla costruzione, al rilascio ed eventualmente alla manutenzione del software.

### Processo unificato

L’unified process (UP) è una struttura di sviluppo software per la costruzione di sistemi orientati agli oggetti ed è iterativa ed incrementale. È un processo di sviluppo relativamente diffuso, è molto flessibile ed aperto, inoltre incoraggia l’uso di pratiche tratte da altri metodi iterativi.

Il UP combina delle best practice comunemente accettate, come un ciclo di vita iterativo ed uno sviluppo guidato dal rischio (risk-driven), nel contesto di una descrizione del processo organica e ben documentata.

L’unified process è quindi:

- Interativo ed incrementale;
- I suoi casi d’uso sono guidati;
- Ha una architettura centrica;
- È focalizzato sui rischi;
- È flessibile e può essere applicato usando un approccio leggero ed agile che comprende pratiche di altri metodi agili.

Una pratica fondamentale di UP è lo sviluppo iterativo evolutivo

Sviluppo iterativo evolutivo → Lo sviluppo è organizzato in una serie di mini-progetti brevi, di lunghezza fissa, chiamati iterazioni, il risultato di ciascuna è un sistema eseguibile, testato ed integrato, ma parziale. Ciascuna iterazione comprende le proprie attività di analisi dei requisiti, progettazione, implementazione e test.

Il ciclo di vita iterativo si basa sull’ampliamento e il raffinamento successivi di un sistema attraverso molteplici iterazioni, con feedback e adattamento ciclici come guide essenziali per convergere verso un sistema appropriato. Il sistema cresce incrementalmente nel tempo, iterazione dopo iterazione, pertanto questo approccio è noto come sviluppo iterativo incrementale. Poiché il feedback e l’adattamento fanno evolvere le specifiche e il progetto, è noto come sviluppo iterativo ed evolutivo.

Vantaggi dello sviluppo iterativo:

- Minore probabilità di fallimento del progetto, migliore produttività, percentuali più basse di difetti; dimostrato da ricerche su metodi iterativi ed evolutivi.
- Riduzione precoce anziché tardiva dei rischi maggiori.
- Progresso visibile fin dall’inizio.
- Feedback precoce, coinvolgimento dell’utente e adattamento, che portano a un sistema che soddisfa meglio le esigenze reali delle parti.
- Gestione della complessità.

- Ciò che si apprende nel corso di un'iterazione può essere usato metodicamente per migliorare il processo di sviluppo stesso, iterazione dopo iterazione.

Molti metodi iterativi raccomandano una lunghezza delle iterazioni da 2 a 6 settimane. Passi piccoli, feedback rapido ed adattamento sono idee centrali dello sviluppo iterativo. Iterazioni più lunghe sono contrarie allo spirito dello sviluppo iterativo ed aumentano il rischio del progetto. In una sola settimana è spesso difficile completare del lavoro sufficiente a produrre abbastanza software ed ottenere feedback significativi; in più di 6 settimane la complessità diventa eccessiva ed il feedback viene ritardato.

Una idea chiave è il timeboxing.

Timeboxing → Le iterazioni hanno una lunghezza fissata. Per es. se si decide che l'iterazione successiva abbia lunghezza 3 settimane, allora il sistema parziale deve essere integrato, testato e stabilizzato in base alla data programmata; non è consentito ritardare la fine dell'iterazione. Un'iterazione di lunghezza prefissata è detta timeboxed.

L'UP divide il progetto in 4 fasi:

1. Ideazione (inception): visione approssimativa, studio economico, portata, stime approssimative dei costi e dei tempi.

Gli obiettivi sono:

- Business case/scope
- Casi d'uso
- Architetture candidate
- Identificazione dei rischi

2. Elaborazione: visione raffinata, implementazione iterativa del nucleo dell'architettura, risoluzione dei rischi maggiori, identificazione della maggior parte dei requisiti e della portata, stime più realistiche.

Gli obiettivi sono:

- Indirizzare i rischi
- Validare l'architettura

È implementata una baseline dell'architettura eseguibile. Questa fase termina con un piano per la fase della costruzione, includendo costi e tempi.

3. Costruzione: implementazione iterativa degli elementi rimanenti, più facili e a rischio minore, preparazione al rilascio. L'obiettivo è implementare le caratteristiche del sistema. Vengono usate operazioni pianificate producendo un rilascio. C'è inoltre un raffinamento incrementale.

4. Transizione: beta test e rilascio. Gli obiettivi sono:

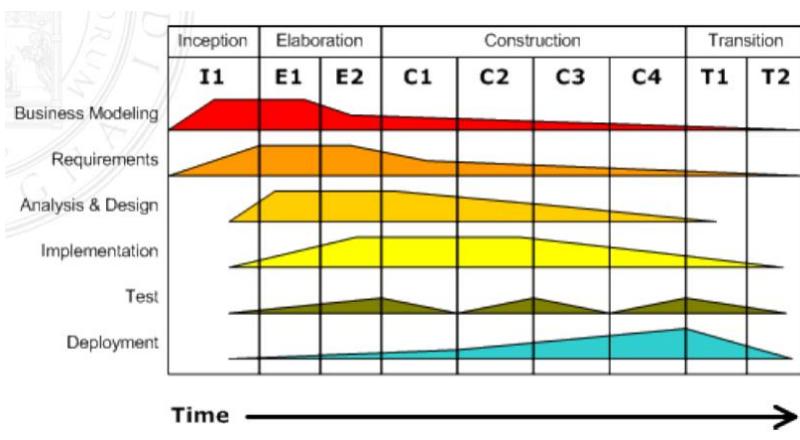
- Schierare un sistema
- Assunzione degli user
- Raccoglimento dei feedback

È inclusa anche una formazione/istruzione.

### Relazioni tra discipline e fasi

Durante un'iterazione viene svolto del lavoro in quasi tutte le discipline. L'impiego, tuttavia, cambia nel tempo.

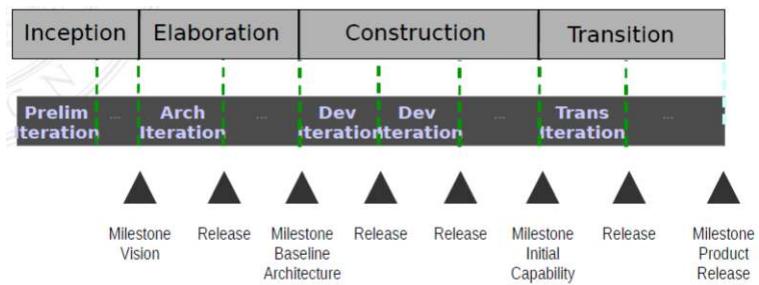
Le iterazioni iniziali tendono in modo naturale a dare una maggiore enfasi relativa sui requisiti e sulla progettazione, mentre quelle successive lo faranno in misura minore, poiché i requisiti e il progetto si stabilizzano attraverso un processo di feedback ed adattamento.



Nella figura viene illustrato il cambiamento dell'impegno relativo rispetto alle fasi.

**Esempio:** durante l'elaborazione le iterazioni tendono ad avere un livello relativamente alto di lavoro sui requisiti e la progettazione, sebbene prevedano anche un certo livello di implementazione. Durante la costruzione, l'enfasi è maggiore sull'implementazione e minore sull'analisi dei requisiti.

#### Fasi ed iterazioni:



#### Più conosciute implementazioni dell'UP:

- Rational Unified Process (RUP);
- Agile Unified Process (OpenUP);
- Oracle Unified Method.

## 4. UML INTRO

Modello → rappresentazione astratta della realtà che serve ad individuare le proprietà salienti secondo un punto di vista. Possono essere utilizzati all'interno di un team di sviluppo per coordinarlo.

I linguaggi servono per convogliare queste istruzioni, per definire quali sono gli elementi che definiscono il soggetto che voglio modellare, proprietà e le relazioni.

Linguaggi di modellazione → definiscono entità proprietà relazioni del soggetto modellato.

### Object Orientation

UML si basa sull'approccio orientato agli oggetti per l'analisi e la progettazione, in particolare, UML (Unified Modeling Language) è un linguaggio visuale per la specifica, la costruzione e la documentazione degli elaborati di un sistema software.

- Con visuale si intende che UML è per la notazione di diagrammi, per disegnare o rappresentare figure relative al software e in particolare al software OO.
- L'analisi si concentra sul dominio del problema.
- Il design si concentra sul dominio della soluzione

Paradigma O-O → assumiamo un punto di vista basato sugli oggetti.

Oggetti → entità autonome contraddistinte da uno stato e da un comportamento. Voglio modelli in cui le entità sono gli oggetti, cioè modelli Object Oriented (O-O) quindi è importante tener conto delle proprietà, delle caratteristiche e di come si realizzano.

I principi fondamentali O-O sono:

- Astrazione → l'oggetto che voglio modellare, corrisponde all'astrazione.
- Incapsulamento → ogni entità che modelliamo nasconde agli altri il proprio stato, che quindi non conoscono i dettagli di come è strutturato internamente, ma interagiscono con me solo con i comportamenti.
- **Esempio:** in Java le variabili di istanza non devono essere pubbliche.
- Ereditarietà → il comportamento e lo stato possono essere specializzati. A livello più basso, l'ereditarietà è una maniera per riutilizzare oggetti/classi esistenti (soprattutto nei linguaggi di programmazione O-O, ma non tutti).
- **Esempio:** Java è class based, quindi dalla classe posso creare delle istanze o delle classi derivate, più aggiungere altro che specializza.
- Polimorfismo → il comportamento dipende da chi sei. A partire da una singola classe, che serve per specificare come sono fatti degli individui, si possono creare tante istanze diverse, ognuna delle quali avrà lo stesso comportamento (stessa caratteristica) e una propria versione della struttura dello stato (ogni istanza ha la propria copia). Agli oggetti si può accedere tramite un nome, inoltre essi hanno dei tipi e delle references che vengono utilizzate l'accesso.

**Esempio:**

```

Class animal {
    //Proprietà che vengono ereditate
    int age;
    ..
    String makeVerse() {
        return "my verse";
    }
}

Class Dog extends Animal {
    ...
    //Faccio l'override del metodo
    String makeVerse() {
        return "bau bau";
    }
    Dog spike = new Dog();
    Animal spike = new Dog();
    spike makeVerse;
    //ne ho due versioni, ma viene chiamata quella di dog per il
    polimorfismo.
}

```

// Perciò anche se la reference è di tipo Animal, l'oggetto è di tipo Dog quindi stampa bau bau.

→ La reference può essere associata a oggetti di tipi diversi. Il comportamento però non è quello del tipo della reference → questo prende il nome di dynamic dispatching.

## Storia di UML

Negli anni '80 l'adozione di metodi di analisi e progettazione orientata agli oggetti, portò allo sviluppo di diversi linguaggi di modellazione. → Fra il 1989 e il 1994 il numero di metodi O-O passò da 10 ad oltre 100, i metodi più promettenti erano quello di Booch, OOSE (Object-Oriented Software Engineering) di Jacobson e OMT (Object-Modeling Technique) di Rumbaugh.

Nel 1994 il metodo di Booch e OMT iniziarono ad unificarsi, l'anno successivo si unì anche Jacobson e nel 1996 venne realizzata la prima versione di UML. Ci fu un importante aggiornamento qualche anno dopo, e nel 2005 venne realizzata l'ultima versione che è quella tutt'ora in uso.

## UML

UML nasce come linguaggio di programmazione software intensive, che parte dall'idea che si possa modellare sia nel dominio del problema che nel dominio della produzione e in ogni caso si assume una produzione O-O.

→ È un linguaggio visuale (grafico), che si esprime quindi attraverso diagrammi → linguaggio semi-formale, che quindi ha regole sintattiche e semantiche, per disegnare diagrammi validi e significativi.

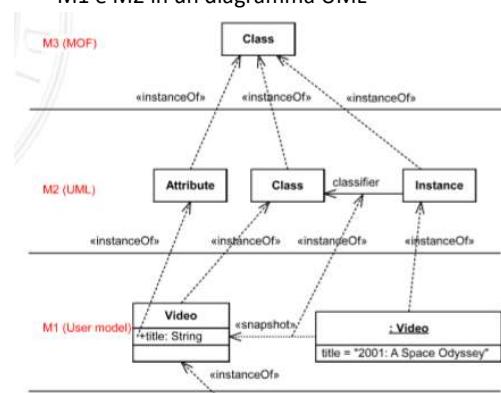
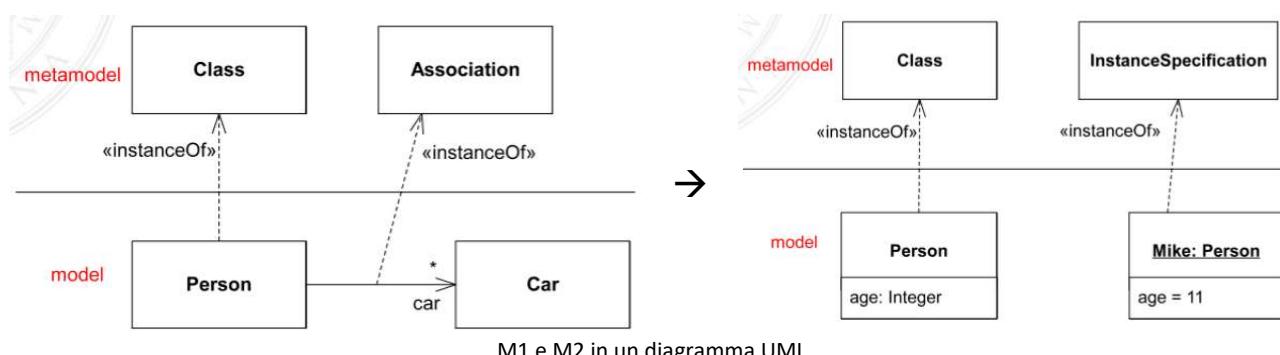
→ NON è un software per lo sviluppo di progetti!

3 modi per applicare UML:

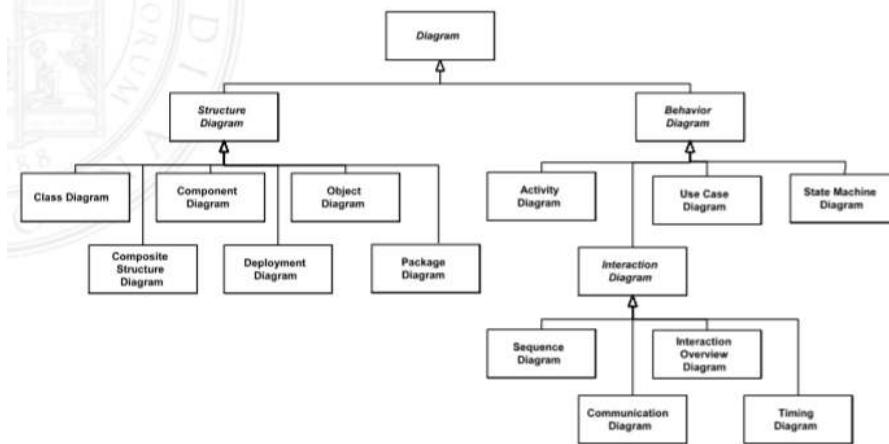
1. UML come abbozzo → diagrammi informali e incompleti, per esplorare parti difficili dello spazio del problema o della soluzione
2. UML come progetto → diagrammi di progetto relativamente dettagliati utilizzati per il reverse engineering oppure per la generazione del codice.
3. UML come linguaggio di programmazione

→ La modellazione agile enfatizza l'uso di UML come abbozzo, è il metodo più comune per applicarlo spesso con un elevato ritorno in termini di tempo.

UML è definito da uno standard di modellazione OMG chiamato MOF (Meta Object Facility) che è strutturato in 4 livelli: M0, M1, M2, M3. → I linguaggi basati su MOF, come UML, possono essere serializzati come definito dallo standard XMI.



# The 13 UML diagrams



Il modello UML include:

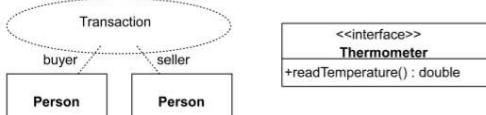
- **Classifiers** → insieme di cose
- **Events** → serie di occorrenze
- **Behaviors** → serie di esecuzioni
- + Elementi che distinguono i diversi tipi di diagrammi

Elementi strutturali:

- Classi → un classificatore che raggruppa oggetti con gli stessi attributi, operazioni, relazioni e semantica.
- Casi d'uso → un classificatore che raggruppa interazioni con risultati osservabili.



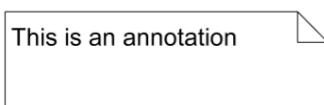
- Collaborazioni → insieme di ruoli con un comportamento cooperativo in grado di realizzare operazioni o casi d'uso.
- Interfacce → insieme di operazioni che possono essere offerte o richieste da altri elementi.



- Pacchetti → usati per raggruppare elementi e dare loro un namespace.

Entità di comportamento:

- Interazioni: unione di comportamenti basati su messaggi
- Annotazioni: migliorano la leggibilità del diagramma

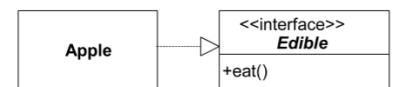
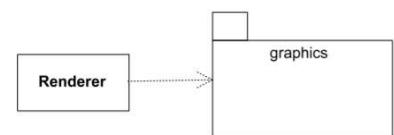
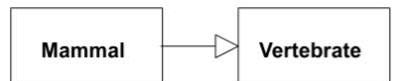


Relazioni → collegano 2 o più elementi, sono rappresentate da linee e sono di 4 tipi:

1. Associazione → dato un elemento, possiamo risalire ad un altro elemento che è legato concettualmente e possiamo risalire a delle informazioni su di esso. In genere non si usa nella modellazione concettuale.



- Linea continua (e freccia aperta nell'end point).
- 2. Generalizzazione → Indica che un elemento è la specializzazione di uno più generale, cioè un certo elemento è un tipo di un altro elemento.
- Linea continua con freccia verso l'elemento più generale.
- 3. Dipendenza → fra gli elementi c'è una relazione di tipo "utilizzatore-fornitore" cioè un elemento ha bisogno delle funzionalità di un'altra classe, quindi cambiamenti in un elemento del modello può causare cambiamenti in altri elementi.  
→ Linea tratteggiata con freccia aperta.
- 4. Realizzazione → si ha fra due elementi del modello quando uno dei due realizza, o implementa, un comportamento che l'altro specifica. Il caso più comune è quando ci sono le classi che realizzano un'interfaccia.  
→ Come una dipendenza ma con la freccia a triangolo.



## OCL

Nel metamodello di UML, un'operazione ha una firma, con nome e parametri, ed è associata ad un insieme di oggetti UML di tipo Constraint, cioè vincoli classificati come "pre-condizioni" e "post-condizioni". OCL (Object Constraint Language) è un linguaggio formale e rigoroso che può essere utilizzato in UML per esprimere i vincoli delle operazioni. OCL definisce un formato ufficiale per specificare le pre-condizioni e le post-condizioni per le operazioni.

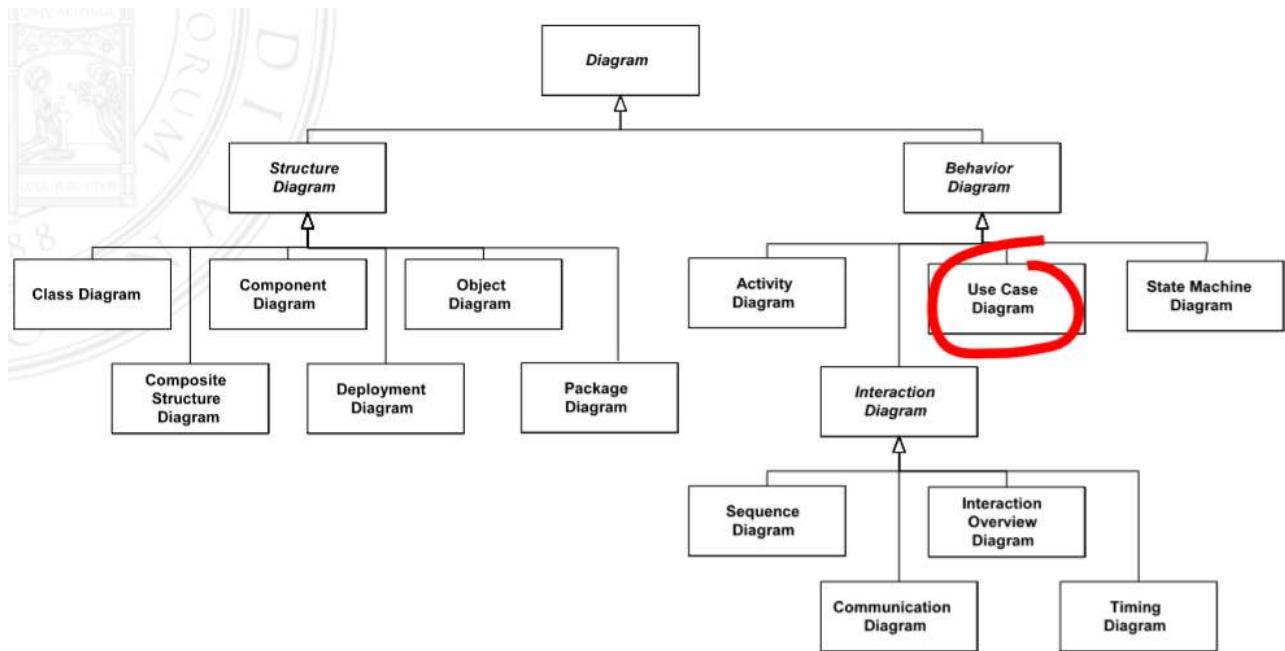
**Esempio:** System: makeNewSale()  
 pre: <dichiarazioni in OCL>  
 post: ...

OCL comprende:

- Inv → invariant
- Pre → pre-condition
- Post → post-condition
- Init → valore iniziale in un contesto
- Derive → definizione di un attributo derivato in un contesto

## 5. UML: USE CASE

I casi d'uso sono storie scritte, ampiamente utilizzati per scoprire e registrare i requisiti. Essi influenzano molti aspetti di un progetto, compresa l'analisi e la progettazione orientata agli oggetti.



I diagrammi dei casi d'uso e le relazioni tra casi d'uso sono secondari al lavoro che riguarda i casi d'uso. → Un semplice diagramma dei casi d'uso fornisce un diagramma di contesto per il sistema, visuale e conciso, che illustra gli attori esterni e il modo in cui utilizzano il sistema.

I diagrammi dei casi d'uso sono dei diagrammi di comportamento, usati per descrivere le azioni svolte da uno o più sistemi, o per descrivere collaborazioni con soggetti esterni al sistema.

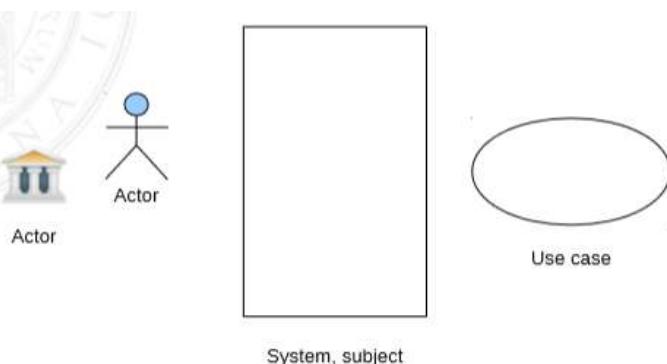
→ Ogni caso d'uso dovrebbe dare risultati osservabili e di valore per uno specifico attore.

I diagrammi dei casi d'uso sono usati per specificare:

- I confini del sistema, ciò che giace al suo esterno e come esso viene utilizzato;
- Il comportamento di un sistema esterno e i suoi attori;
- Ciò che un sistema dovrebbe fare → requisiti dei soggetti e di chi usa il sistema
- Ciò che il sistema può fare → funzionalità
- Requisiti che il soggetto pone nel suo ambiente, ma anche che il sistema pone sul suo ambiente → definendo le “regole” di interazione

→ **Un errore comune è concentrarsi sui diagrammi di UML dei casi d'uso, anziché sul testo dei casi d'uso che è molto più importante.**

### UC elements



Ci sono 3 tipi di elementi:

- Attore → qualcosa o qualcuno dotato di comportamento (una persona, un sistema informatico o un'organizzazione);
- Sistema → oggetto in analisi all'interno del quale vengono applicati i casi d'uso;
- Caso d'uso → collezione di scenari correlati, di successo e fallimento, che descrivono un attore che usa un sistema per raggiungere un obiettivo.

## Attore

In UML un attore è qualcosa o qualcuno dotato di comportamento, con un ruolo specifico come soggetto esterno al Sistema in Discussione (*SuD* → System under Discussion).

Generalmente viene rappresentato da un'omino stilizzato con sotto riportato il nome dell'attore o il ruolo ("Cassiere").

1. Attore primario: raggiunge degli obiettivi utente utilizzando i servizi del SuD (**Esempio**: Cassiere).  
→ Utile per trovare gli obiettivi degli utenti.
2. Attore di supporto: offre un servizio al SuD (**Esempio**: Servizio informatico che autorizza i pagamenti).  
→ Utile per chiarire le interfacce esterne e i loro protocolli.
3. Attore fuori scena: ha un interesse nel comportamento del caso d'uso, ma non è attore primario o di supporto (**Esempio**: Governo interessato al pagamento delle imposte).  
→ Utile per garantire che tutti gli interessi necessari vengano individuati e soddisfatti.

## Sistema in discussione (SuD)

Il sistema in discussione è l'oggetto a cui si applicano una serie di casi d'uso. Normalmente un caso d'uso descrive l'utilizzo di un sistema software (**caso d'uso di sistema**), dove quindi l'oggetto è il sistema, o il modo in cui un'azienda viene utilizzata dai suoi clienti e soci (**caso d'uso di business**), dove invece l'oggetto è un'organizzazione.

Il sistema è rappresentato con un rettangolo, con il nome del sistema in alto, all'interno del quale vengono applicati i casi d'uso, mentre gli attori rimangono disegnati esternamente nel caso d'uso di sistema, mentre li possiamo trovare disegnati internamente nel caso d'uso di business.

## Casi d'uso

In UML un caso d'uso è una collezione di scenari correlati, di successo e fallimento, che descrivono un attore che usa un sistema per raggiungere un obiettivo. I casi d'uso vengono definiti in UP nell'ambito della disciplina dei Requisiti, cioè dell'insieme di tutti i casi d'uso descritti; è un modello delle funzionalità del sistema e del suo ambiente.

I casi d'uso sono requisiti, soprattutto funzionali e comportamentali, che indicano che cosa farà il sistema. Un punto di vista correlato è che il caso d'uso definisce un contratto relativo al comportamento di un sistema.  
→ In altre parole, ogni caso d'uso descrive un'unità di funzioni complete che l'oggetto fornisce ai propri utenti.

Il caso d'uso è rappresentato come un'ellisse contenente il nome del caso d'uso.

## Scenario

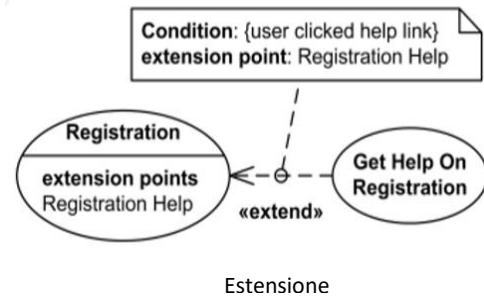
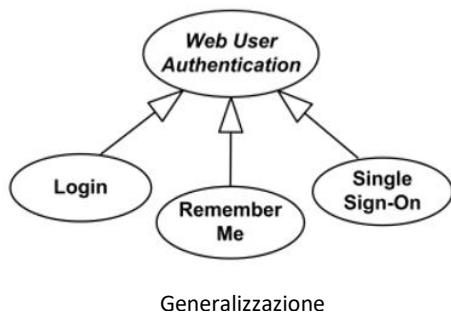
Uno scenario è una sequenza specifica di azioni e interazioni tra il sistema e alcuni attori, descrive una particolare storia nell'uso del sistema o un percorso attraverso il caso d'uso. Ci sono due tipi di scenari:

1. Scenario di successo principale → Un cliente arriva alla cassa con alcuni articoli da restituire. Il cassiere utilizza il sistema POS per registrare ciascun articolo restituito e la restituzione va a buon fine.
2. Scenari alternativi → Se il cliente aveva pagato con carta di credito e l'operazione di rimborso sul relativo conto di credito è stata respinta, allora il cliente viene informato e viene rimborsato in contanti. Oppure, se il sistema rileva un fallimento nella comunicazione con il sistema esterno di gestione della contabilità.

## Relazioni fra attori

È possibile definire attori astratti o concreti e specificare le relazioni fra loro. L'idea è che un caso d'uso rappresenti i passi di interazione nel momento in cui gli attori si relazionano, i tipi di relazione sono simili a quelle già viste nel capitolo precedente:

1. Associazione → Ogni caso d'uso specifica un'unità di funzioni utile che il sistema fornisce agli attori, queste funzioni dovrebbero essere avviati da un attore. Gli attori possono essere collegati ai casi d'uso solo per relazione di associazione binaria.  
→ Non si usa nella modellazione concettuale.
2. Generalizzazione → Dal punto di vista concettuale indica che un elemento è generalizzazione di uno più generale.  
→ `generalize` è rappresentata una linea continua e una freccia aperta.



3. Estensione → L'idea è quella di creare un caso d'uso di estensione, e al suo interno descrivere dove e sotto quali condizioni esso estende il comportamento di qualche caso d'uso base. In altre parole, usiamo `extend` quando vogliamo sottolineare che c'è un comportamento addizionale e opzionale che non sempre ha luogo.

*Tutti i passi di interazione devono sempre avere luogo quando definisco il caso generale?* No, cioè posso scrivere:

passo 1)  
passo 2)  
se avviene ... allora → passo3)  
passo 4)

oppure:  
passo 1)  
passo 2) → passo 3) come extension point  
passo 4)  
che è la stessa cosa.

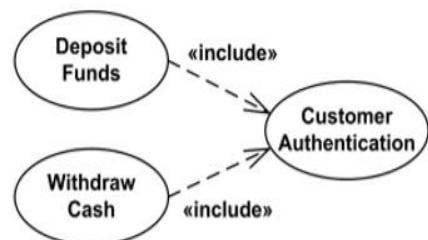
Nel primo caso è modellato come un singolo caso d'uso con un passo opzionale, uso il secondo caso se voglio sottolineare che c'è una cosa che devia dal percorso principale.

Il punto di estensione (`extension point`) è una caratteristica di un caso d'uso che identifica un punto del comportamento del caso d'uso in cui tale comportamento può essere esteso appunto; è attivato da una certa condizione (trigger), e sono etichette nel caso d'uso base di estensione, così che la numerazione in passi del caso d'uso base può cambiare senza influire sul caso d'uso d'estensione. → Nuova indirizzazione.

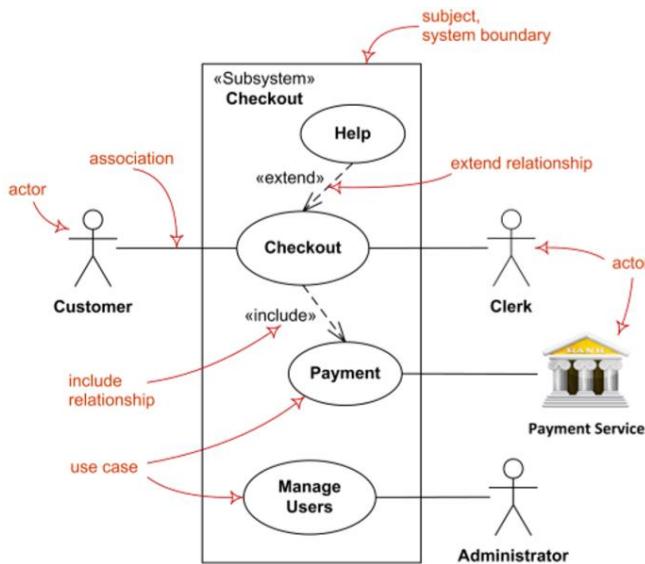
→ Ogni punto di estensione deve avere un nome, univoco, all'interno del caso d'uso.  
Può esserci l'estensione di una estensione, ma una estensione è unica per un caso d'uso.  
→ La relazione di `extend` viene rappresentata con una linea tratteggiata e una freccia che punta il caso d'uso principale, l'`extension point` viene rappresentato con un cerchio vuoto.

4. Inclusione → La relazione include è la più comune e la più importante, spesso si tratta di un refactoring e di un collegamento del testo per evitare la duplicazione. Un altro utilizzo è quello per descrivere la gestione di un evento asincrono come quando un utente è in grado di selezionare o passare ad una finestra, una funzione o una pagina web.
- La relazione di inclusione serve a specificare dei passi ulteriori che caratterizzano un certo caso d'uso e che vengono eseguiti *sempre*.

→ Viene rappresentato con una linea tratteggiata e una freccia che punta l'inclusione.



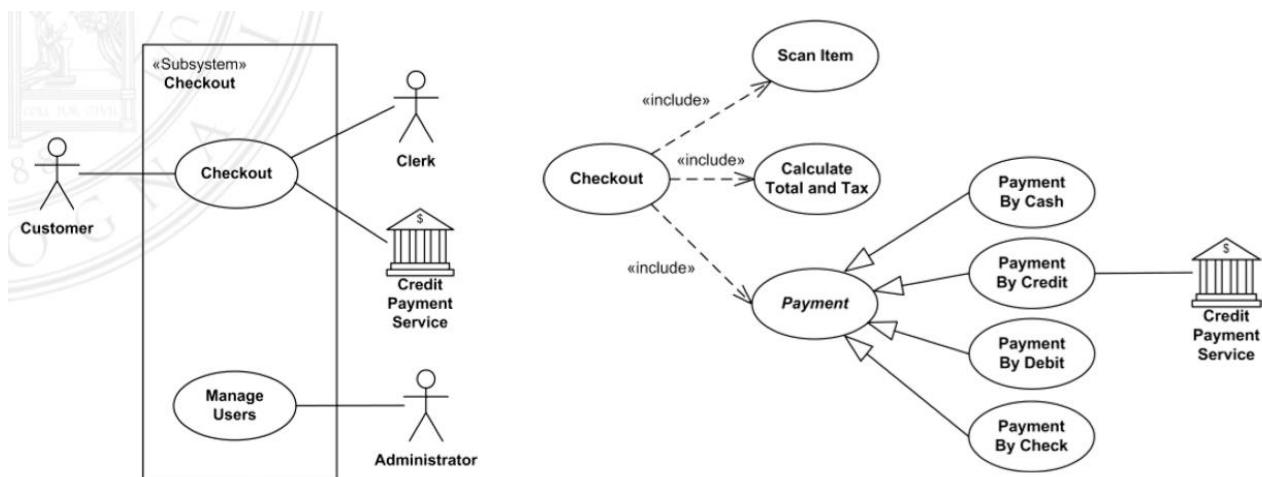
Quando si segnano le relazioni all'interno dei casi d'uso si scrive la parola chiave all'interno di <<>>.



*Cosa fare e cosa non fare con uno Use Case:*

- Non scomporlo per altri motivi se non per riusarlo
- Distinguere generalizzazione, inclusione e estensione
- Non usare “stereotipi” di relazioni fra attori
- Ricordarsi che gli attori sono esterni al sistema
- L'unica relazione significativa tra attori è la generalizzazione
- Il tempo può essere un attore nel sistema

### Templates di UC



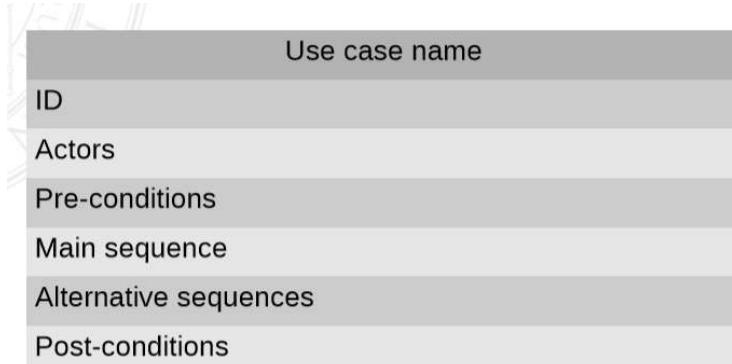
Il diagramma dei casi d'uso non specifica la natura delle interazioni fra gli elementi, possono essere usati altri tipi di diagrammi ma anche questi non catturano gli aspetti specifici. Non esiste una notazione standard per quanto riguarda i dettagli dei casi d'uso, ma ci sono diversi templete che possono essere usati.

### Cockburn's fully dressed

È un documento completo che viene usato nei contesti con progetti di media-grande dimensione, con team molto ampi.

<b>Titolo</b>	Frase che inizia con un verbo che spieghi l'obiettivo dell'attore principale.
<b>Attore primario</b>	Attore che chiede al sistema di fornirgli i suoi servizi.
<b>Portata</b>	Describe i confini del sistema che si sta progettando.
<b>Scopo</b>	
<b>Livello</b>	<u>"Obiettivo utente"</u> , descrive gli scenari con cui un attore primario può portare a termine il suo lavoro, o <u>"sottofunzione"</u> , descrive i sotto-passi in supporto al raggiungimento di un obiettivo dell'utente.
<b>Parti interessate e interessi</b>	A chi interessa questo caso d'uso e che cosa desidera (altri attori). Suggerisce e limita ciò che il sistema deve fare, infatti nel caso d'uso vanno scritti <u>tutti e solo</u> i comportamenti relativi alla soddisfazione degli interessi delle parti interessate.
<b>Pre-condizioni</b>	Che cosa deve essere <u>sempre vero prima</u> di iniziare uno scenario del caso d'uso. Le pre-condizioni non vengono verificate all'interno del caso d'uso, rappresentano cioè i presupposti significativi.
<b>Garanzie di successo (o post-condizioni)</b>	Affermano ciò che deve essere <u>vero quando è stato completato con successo</u> il caso d'uso, ovvero lo scenario principale di successo o un percorso alternativo.
<b>Trigger</b>	Condizioni che attivano il caso d'uso.
<b>Scenario principale di successo (o Happy path)</b>	Percorso di successo e incondizionato tipico, che soddisfa gli interessi delle parti interessate. Vengono specificati quali passi di interazione vanno dal sistema al cliente e viceversa Spesso non comprende alcuna diramazione o condizione, perché queste vanno nelle "Estensioni".
<b>Estensioni</b>	Tutti gli scenari alternativi, di successo e di fallimento. Di solito comprendono la maggior parte del testo, e sono diramazioni dello scenario principale di successo e vengono indicati con riferimento ai suoi passi. Costituite da condizione e gestione.
<b>Requisiti speciali</b>	Requisiti non funzionali correlati, che si riferiscono in modo specifico ad un caso d'uso.
<b>Elenco delle varianti tecnologiche e dei dati</b>	Varianti nei metodi di I/O e nel formato dei dati.
<b>Frequenza di ripetizione</b>	Frequenza prevista di esecuzione del caso d'uso.
<b>Varie</b>	Altri aspetti come i problemi aperti.

### Template di UC più semplice

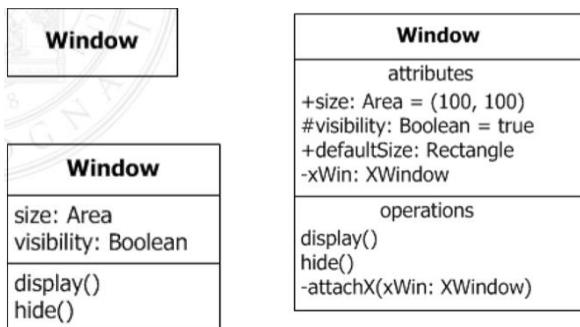


## 6. UML: CLASS DIAGRAM

Un diagramma delle classi è un diagramma che serve a rappresentare degli elementi che condividono determinati aspetti comuni.

Dal punto di vista del diagramma, una classe è rappresentata da un rettangolo, in cui in alto è presente il nome della classe. Il rettangolo può essere diviso in compartimenti (righe orizzontali) che possono contenere informazioni addizionali. → I compartimenti più frequenti sono quello delle operazioni e quello degli attributi.

**Esempio:**



Nel primo caso non ci sono compartimenti, ma solo il nome della classe.

Nel secondo caso appaiono due compartimenti: attributi e notazioni.

Nel terzo caso oltre al nome del compartimento vi sono diversi livelli di dettaglio.

Il diagramma delle classi può essere utilizzato per rappresentare elementi a livelli di astrazione diversi. In alcune classi ci può interessare solo capire gli elementi, mentre in altre abbiamo bisogno di una rappresentazione più dettagliata. → Noi useremo questi diagrammi sia nel dominio del problema che in quello della soluzione.

Un diagramma delle classi UML nasce per rappresentare le classi come elementi di un programma. → Serve a progettare soluzioni e sistemi SW, quindi viene utilizzato nel dominio della soluzione. Un diagramma delle classi UML semplificato e reso astratto al punto opportuno può anche essere utile per la modellazione nel dominio del problema.

Gli attributi e le operazioni servono a rappresentare elementi strutturali o comportamentali riferite a una classe.

Le proprietà sono gli elementi strutturali che si trovano nel compartimento degli attributi; sono descritte attraverso un breve testo, la cui sintassi semplificata è:

[+, -, #, ~] [/] <name> [ :<type> ] [<mult>]

Ovvero un nome preceduto da un simbolo/modificatore di visibilità e seguito opzionalmente da un tipo e da una molteplicità.

**Esempio:** +defaultSize: Rectangle

Le operazioni sono gli elementi comportamentali che si trovano nel compartimento delle operazioni. →

Come le proprietà sono descritte da un breve testo e la loro sintassi semplificata è:

[+, -, #, ~] <name> [ (<params>) ] [ :<type> ]

### Modificatori di visibilità

- +: pubblico → visibile a tutto. Può essere legato sia alle proprietà (in questo caso viene violato il principio di encapsulamento) che alle operazioni.
- -: private → è visibile solo all'interno del suo namespace.
- #: protected → visibile agli elementi a cui è legato con una relazione di generalizzazione
- tilde: package → visibile a tutti gli elementi presenti nel package.

### Molteplicità

Le molteplicità servono a specificare quante volte operazioni, proprietà sono presenti.

Sintassi → <multiplicity-range> ::= [<lower>..] <upper>

In generale si possono definire degli intervalli: l'upper bound deve essere sempre espresso e può essere specificato anche un lower bound, che se presente è sempre seguito da due puntini (...). L'asterisco significa che l'upper bound non è definito.

Molteplicità utilizzate frequentemente:

- 1 → proprietà obbligatoria
- 0..1 → proprietà che può esserci 0 o 1 una volta, quindi facoltativa
- 1...\* → proprietà che deve essere presente almeno 1 volta
- \* → proprietà presente 0 o più volte (\*)

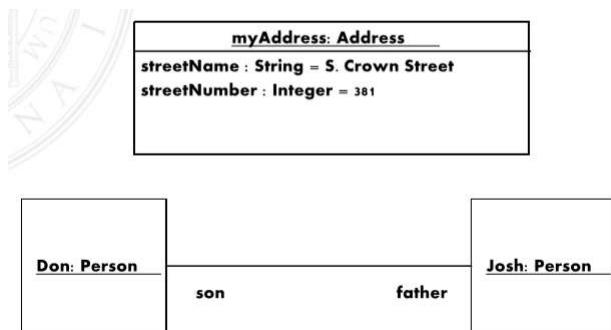
### Istanze

Esiste anche il diagramma degli oggetti, il quale definisce le istanze. Dal punto di vista tecnico non c'è differenza tra i due: il diagramma degli oggetti è un diagramma delle classi all'interno del quale si trovano anche delle istanze di classi (cioè oggetti). → Questi oggetti devono rispondere alla struttura della classe alla quale sono associati e devono contenere valori per ogni attributo della classe in accordo con le caratteristiche di tale attributo, tra cui tipo e molteplicità.

La notazione grafica dell'istanza si distingue da quella della classe perché il nome è sottolineato ed è composto da una concatenazione → nome dell'istanza : classe di cui è istanza. Il nome dell'istanza è opzionale, ma il tipo no.

L'istanza può anche avere dei valori specifiche delle proprietà.

**Esempio:**



### Relazioni

È possibile specificare relazioni diverse nel diagramma delle classi:

**NB:** ognuna viene tracciata attraverso una linea che può avere rappresentazioni grafiche diverse a seconda del tipo di relazione che si sta modellando.

1. Generalizzazione/specializzazione → Relazione che corrisponde alla frase “è un...” e dal verso opposto si può leggere come specializzazione.

Ogni generalizzazione mette in relazione una specifica classe con una più generale assumendo un meccanismo di ereditarietà.

**NOTAZIONE:** linea continua terminante con una freccia a triangolo aperto che punta verso l'elemento più generale.



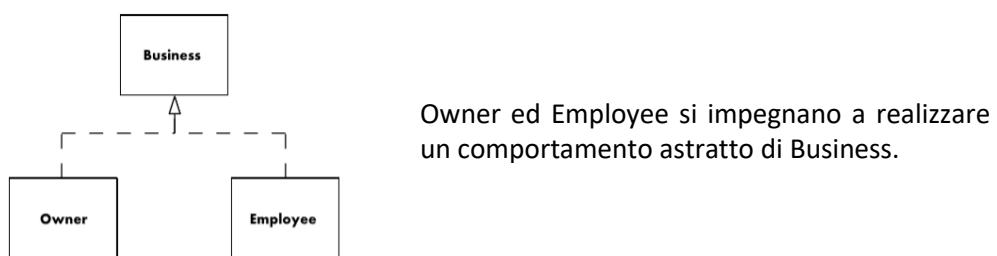
2. Dipendenza → Relazione di tipo “utilizzatore/fornitore” fra gli elementi, dove la modifica del fornitore può avere ripercussioni sull’utilizzatore, ovvero quando un elemento viene sottoposto a delle modifiche, queste si devono riflettere anche sugli elementi che a lui dipendono. Una classe ha necessità delle funzionalità dell’altra per offrire i suoi comportamenti, questo implica che la semantica degli utilizzatori non sia completa senza i fornitori.

*NOTA:* linea tratteggiata terminante con una freccia a triangolo aperto. Sulla linea possono essere inseriti una keyword e un nome, entrambi opzionali.



3. Realizzazione → È un tipo di dipendenza più specifica perché dichiara che un elemento ne realizza un altro. Il caso più comune è quando le classi realizzano l’interfaccia.

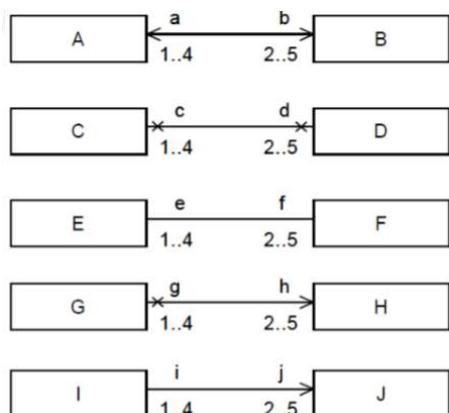
*NOTA:* linea tratteggiata terminante con una freccia a triangolo aperto.



4. Associazione → Un’associazione dichiara che ci possono essere links tra le istanze dei tipi associati, ovvero dato un elemento si è in grado di risalire ad un elemento di un altro tipo che a lui è legato in maniera concettuale. Un link è una tupla con un valore per ogni end-point dell’associazione.

*NOTA:* si rappresenta attraverso una linea continua terminante con una freccia a triangolo aperto. I nomi delle associazioni e etichette possono essere aggiunte per aiutare a leggere il verso dell’associazione.

Negli end-point è possibile inserire nomi, molteplicità, frecce di navigazione o croci.



Se è presente la freccia dato A sono capace di risalire facilmente all’elemento che gli è associato.

Se è presente una croce, allora dato C non è immediato ottenere l’elemento a cui è associato (D).

Se non è presente né freccia né croce è indeterminata la facilità con cui si può risalire all’elemento associato.

**Esempio:** si vuole rappresentare l’associazione: Person x → (1) Dog

Vi è la croce dalla parte di cane, perché dato un cane non è immediato sapere quale sia il suo padrone. Dalla parte della persona invece vi è una freccia perché con la proprietà myDog è facile risalire al suo cane.

```

Class Dog {
    Private String name;
    ...
}
  
```

```

Class Person {
    ...
    Dog mydog;
    //facile sapere quale cane perché
    //mydog dà subito il cane della persona
}
  
```



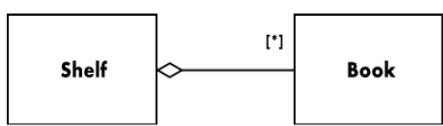
**NB:** Se si sta modellando nel dominio del problema, frecce e croci non si usano perché non sono facilmente interpretabili dal punto di vista concettuale.

→ Aggregazione e composizione sono due tipi particolari di associazione che arricchiscono l'associazione stessa.

5. Aggregazione (vero nome: shared association) → È un rafforzamento del legame dell'associazione, ma l'elemento aggregato esiste anche senza l'altro elemento.

L'elemento che viene aggregato generalmente ha una cardinalità aperta (\*).

*NOTAZIONE:* si rappresenta con una linea terminante con un rombo vuoto.

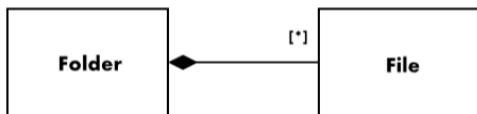


**Esempio:** Tra scaffale e libreria c'è un'aggregazione: un libro è sullo scaffale e su questo ci possono essere più libri. Poiché è un'aggregazione e non una generica associazione, non solo shelf è associata a book, ma aggrega dei libri.

6. Composizione (vero nome: composite aggregation) → è ancora più forte dell'aggregazione, perché l'elemento che è associato dipende/è parte dell'elemento aggregante.

Solo un elemento può essere aggregato compositamente, è esclusiva.

*NOTAZIONE:* si rappresenta con una linea terminante con un rombo pieno.

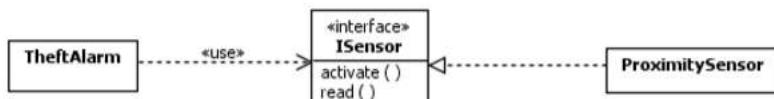


**Esempio:** all'interno di una directory ci possono essere più file, ma il file non può esistere senza la directory che lo contiene.

UML ha anche una serie di elementi per rappresentare concetti di programmazione.

Classe astratta → Una classe astratta non ha istanze dirette, ma le sue istanze sono istanze di una delle sue specializzazioni. Il nome di una classe astratta è compreso fra graffe o viene scritto in italics: *{abstract}*.

Interfaccia → le interfacce dichiarano servizi coerenti e sono implementate da classi che, a loro volta, implementano tali servizi. Il nome delle interfacce viene compreso fra ghirne <<>>.



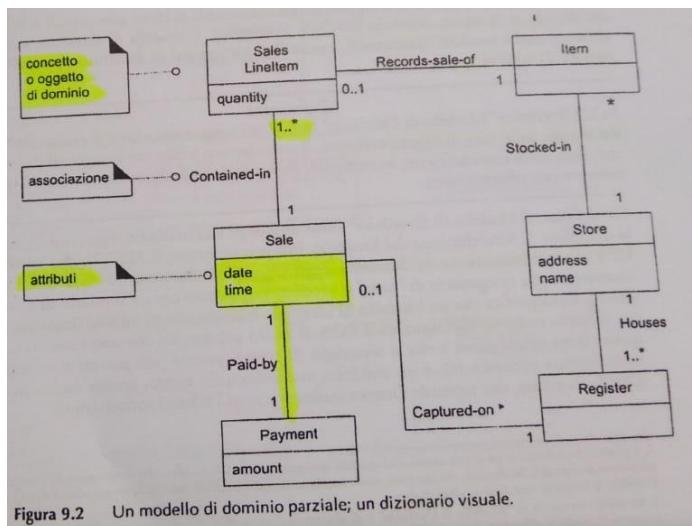
**NB:** Il diagramma delle classi può essere realizzato con diversi modelli: design model (rappresenta elementi della soluzione), domain model (rappresenta elementi del problema). Se è pensato nel dominio della soluzione, allora ha un dettaglio molto puntuale, quindi è molto ricche. Nel dominio del problema, la notazione non cambia, ma essendo ad un livello di astrazione diverso si avranno molti meno dettagli.

## 7. ANALYSIS MODEL – DOMAIN MODEL

Un modello di dominio è il modello più importante e classico dell'analisi OO poiché illustra i concetti significativi di un dominio, le caratteristiche e come si relazionano tra loro.

Un modello di dominio è una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio. → Sono chiamati anche modelli concettuali, modelli degli oggetti di dominio e modelli degli oggetti di analisi.

**Esempio:** modello di dominio parziale, disegnato con la notazione di un diagramma delle classi di UML.



**Payment** è correlato a **Sale** in modo significativo;  
**Sale** ha una data e un'ora, attributi informativi che è importante conoscere.

Modello di dominio → rappresentazione di classi concettuali del mondo reale, *non* di oggetti software. Il termine *non* indica un insieme di diagrammi che descrivono classi software o oggetti software con responsabilità.

I seguenti elementi non sono adatti a essere mostrati in un modello di dominio:

- Elementi software → come una finestra o una base di dati
- Responsabilità o metodi

Il modello di dominio di UP è una specializzazione del Modello degli Oggetti di Business (BOM) di UP, che è incentrato sulla spiegazione di "cose" e prodotti importanti per un dominio di business, ciò significa che un Modello di Dominio è incentrato su un solo dominio, come per esempio, tutto ciò che riguarda il POS.

Applicando la notazione UML, un modello di dominio è illustrato con un insieme di diagrammi delle classi in cui *non* sono definite operazioni (firme di metodi). Esso fornisce un punto di vista concettuale e può mostrare:

- Oggetti di dominio o classi concettuali
- Associazioni tra classi concettuali
- Attributi di classi concettuali

→ Pertanto, il modello di dominio è un dizionario visuale delle astrazioni significative della terminologia del dominio e del contenuto informativo del dominio.

Classe concettuale → è un'idea, una cosa o un oggetto. Può essere considerata in termini del suo simbolo, della sua intensione e della sua estensione:

- Simbolo → parole o immagini che rappresentano una classe concettuale
- Intensione → la definizione di una classe concettuale
- Estensione → l'insieme di esempi a cui la classe concettuale si applica

### Come creare un modello di dominio

Si devono affrontare i seguenti passi:

1. Trovare le classi concettuali
2. Disegnarle come classi in un diagramma delle classi di UML

### 3. Aggiungere associazioni e attributi

#### 1. Trovare classi concettuali

Ci sono 3 strategie per trovare le classi concettuali:

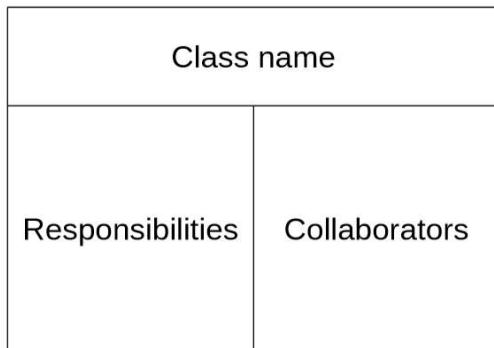
- 1) Riusare o modificare modelli esterni → usare modelli di dominio e modelli di dati pubblicati e ben congegnati per molti domini comuni.
- 2) Utilizzare un elenco di categorie → vedi **Tabella** sotto.
- 3) Identificare nomi e locuzioni nominali → è un'analisi linguistica:
  - Identificazione di nomi e frasi formate da un nome principale insieme ai suoi aggettivi e determinanti → vengono poi considerati come attributi.
  - Analisi dei verbi → per identificare responsabilità o collaborazioni.

**Tabella delle categorie**

<b>Transazioni commerciali</b> → Aspetti critici	Sale, Payment, Reservation.
<b>Elementi/righe di transazioni</b>	SalesLineItem
<b>Prodotto o servizio correlato a una transazione</b> → Le transazioni sono per qualcosa	Item Flight, Seat, Meal
<b>Dove viene registrata la transazione?</b> → Importante	Register, Ledger FlightManifest
<b>Ruoli di persone o organizzazioni correlati; attori nei casi d'uso</b> → Quali sono le parti coinvolte	Cashier, Customer, Store MonopolyPlayer Passenger, Airline
<b>Luogo della transazione; luogo del servizio</b>	Store, Airport, Plane, Seat
<b>Eventi significativi, spesso con un'ora o un luogo che è necessario ricordare</b>	Sale, Payment MonopolyGame, Flight
<b>Oggetti fisici</b> → Importante soprattutto quando si crea un software per il controllo di dispositivi o simulazioni	Item, Register Board, Piece, Die Airplane
<b>Descrizioni di oggetti</b>	ProductDescription, FightDescription
<b>Cataloghi di oggetti</b> → Fisici o informazioni	Store, Bin Board, Airplane
<b>Oggetti in un contenitore</b>	Item, Square, Passenger
<b>Altri sistemi che collaborano</b>	CreditAuthorizationSystem, AirTrafficControl
<b>Registrazioni di questioni finanziarie, di lavoro, contrattuali e legali</b>	Recepit, Ledger MaintenanceLog
<b>Strumenti finanziari</b>	Cash, Check, LineOfCredit, TicketCredit
<b>Piani, manuali, documenti cui si fa regolarmente riferimento per eseguire lavori</b>	DailyPriceChangeList RepairSchedule

## **CRC cards**

CRC cards (Class Responsibility Collaborator) sono tools che possono essere usati per la descrizione del Modello di Dominio.



**Esempio** di Cards:

Cart		User
•Knows user •Knows items •Adds items •Removes items	Item Item	•Knows name •Knows cart •Puts items in cart •Checks out

Le CRC cards si ottengono iterativamente seguendo questi passi:

1. Si identificano le classi
2. Si trovano le responsabilità
3. Si definiscono le collaborazioni
4. Si avvicinano le carte in collaborazione

→ Possono essere trasformate in diagrammi UML o utilizzate per rappresentare il Modello di Dominio.

→ Sono utili, ma il vantaggio principale è il coinvolgimento del cliente.

## 8. UML: ACTIVITY DIAGRAM

Il diagramma di attività è un diagramma UML del comportamento che mostra le attività sequenziali e parallele in un processo. Tipicamente fra queste attività esistono delle dipendenze causali di quale genere, ovvero una certa attività non può cominciare finché un'altra non è terminata. → Questo insieme coordinato di attività serve a rappresentare un meccanismo attraverso il quale raggiungere un qualche obiettivo.

I diagrammi di attività possono essere usati per modellare il comportamento di elementi diversi come: classi, casi d'uso, interfacce, componenti, operazioni delle classi, algoritmi.

I diagrammi di attività in UML 2.x sono formalizzati per essere basati liberamente sulla semantica delle reti di Petri, un importante modello della teoria computazionale dell'informatica.

La metafora o realizzazione delle reti di Petri è che ci sono dei token che scorrono attraverso il grafo dell'attività.

A differenza di quello avviene per altri diagrammi, il lato semantico è relativamente formalizzato: la semantica è descritta in termini di transizioni fra markings (distribuzioni di token nella rete). → Si è in grado di esprimere situazioni concorrenti.

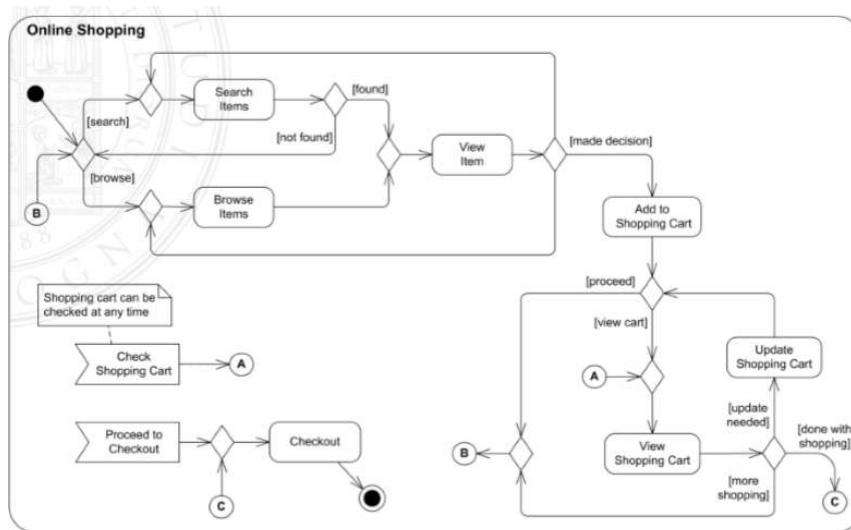
### **Principali elementi di un AD**

L'elemento principale è l'attività che si sta descrivendo.

Tale attività è strutturata in nodi (activity node) che possono essere di tipo diverso: azioni, oggetti, controllori di flusso.

Altri elementi sono gli activity edges che permettono di collegare fra loro i vari nodi.

**Esempio** di AD: sito di e-commerce quando gli utenti interagiscono con l'applicazione.

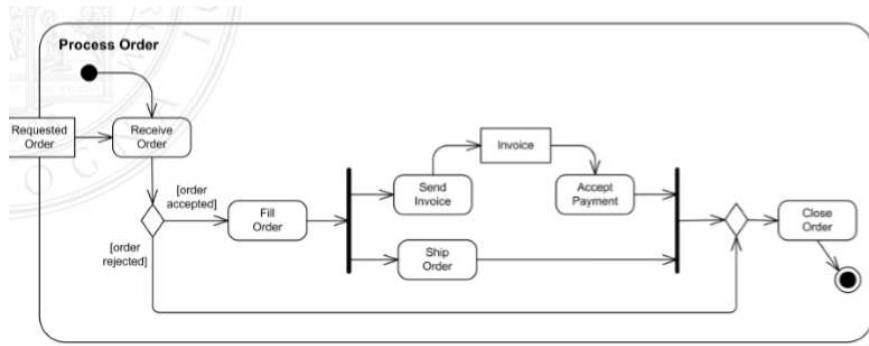


Il riquadro esterno è l'attività. I rettangoli arrotondati sono le actions, le frecce sono gli action edges e i rombi sono dei control.

L'idea dei diagrammi di attività è quella di definire un insieme di azioni e specificare come si evolvono nel corso del tempo. → Non sempre un comportamento è rigidamente caratterizzato da una stessa sequenza di azioni, in certi casi le azioni possono essere ripetute o variare.

Questa struttura di controllo permette di esprimere le alternative che il flusso di controllo può realizzare all'interno del modello.

Nel diagramma dell'online shopping c'è sempre un unico flusso attivo, ma gli AD supportano la concorrenza (più flussi contemporaneamente), come nel caso del modello per il processamento di un ordine.



La barra verticale abilita due flussi: cioè entra un flusso e ne escono due paralleli. Send invoice e ship order possono essere realizzati in maniera concorrente e due flussi rimangono concorrenti fino al loro termine. Object node → nodo che definisce un punto del flusso in cui diverse azioni si passano dei dati sotto forma di oggetto.

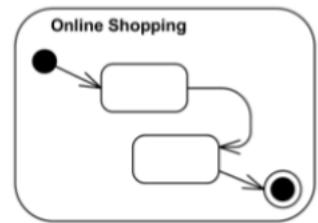
**Esempio:** Quando send invoice termina attiva accept payment specificando quali sono i dati sui quali potrà operare, ovvero invoice.

### Attività

L'attività è l'elemento complessivo che si sta modellando. → È un comportamento parametrico rappresentato come un flusso di azioni coordinato. Il flusso di esecuzione è modellato con activity node connessi attraverso activity edges.

**NOTAZIONE:** L'attività viene rappresentata attraverso un rettangolo tondeggiante, il cui nome deve essere esplicitato in alto a sinistra.

Nodi ed edges che descrivono l'attività vengono rappresentati all'interno del rettangolo.



### Azione

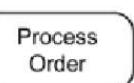
Le azioni sono elementi, a cui è associato un nome, che rappresentano un passo atomico all'interno dell'attività.

Passo atomico = cioè passo indivisibile.

Esistono diversi tipi di attività:

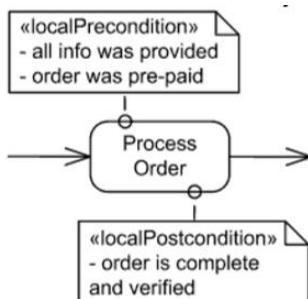
- Invocazione di una funzione o occorrenza di una funzione primitiva.
- Azioni di comunicazione come mandare o ricevere un segnale.
- Manipolazioni di oggetti come leggere o modificare attributi o associazioni.
- Invocazione di un comportamento come un'altra attività.

**NOTAZIONE:** Le azioni sono rappresentate attraverso rettangoli arrotondati, all'interno del quale si trova il nome o la descrizione dell'azione stessa.



Un'azione può avere un insieme di archi entranti e uscenti che specificano il flusso di dati da e per gli altri nodi.

Un'azione non viene eseguita finché tutte le sue condizioni di input non sono soddisfatte.



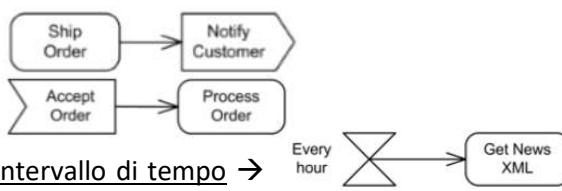
È possibile specificare pre-condizioni e post-condizioni che indicano degli invarianti che devono essere soddisfatti prima/dopo l'esecuzione dell'attività. Pre e post-condizioni sono valide solo nel punto in cui sono specificate.

## Event action

Azioni che corrispondono a eventi/segnali/messaggi che vengono generati.

NOTAZIONE:

- Generare un segnale →



- Accettare un segnale →

- Ripetere un'azione per un intervallo di tempo →

**Esempio:** Ogni ora viene generato un token che avvia get news XML.

## Call behavior action

Una call behavior action rappresenta la chiamata di un'attività, cioè l'attività si espande in un diagramma di sotto-attività.

NOTAZIONE: è indicata con un simbolo a forchetta all'interno del rettangolo arrotondato.



## Activity edges

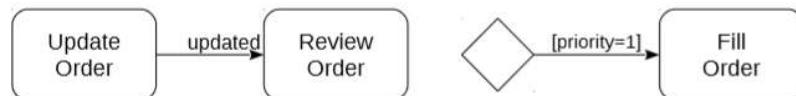
Gli edges definiscono come i flussi si muovono all'interno dell'attività. → Essi rappresentano il collegamento fra un'attività sorgente e un'attività target.

Possono essere specializzati in control flow e object flow:

- Control flow → non trasporta informazioni, ma indica solo che il flusso è arrivato fino a quel punto
- Object flow → indica il punto in cui è arrivato il flusso e trasporta dati

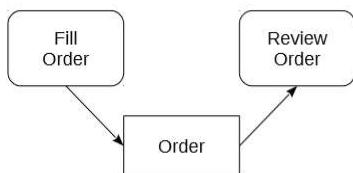
È possibile specificare delle guardie che definiscono dei predici che devono essere veri perché il token possa attraversare l'edge. → Nel caso siano falsi allora il token viene bloccato fino a quando i predici non risultano veri.

NOTAZIONE: gli edges sono rappresentati da frecce.



## Object node

L' object node è un activity node che specifica dei dati che vengono trasportati tra un'azione e l'altra. → Indica che un'istanza di un particolare classificatore potrebbe essere disponibile in un particolare punto dell'attività.

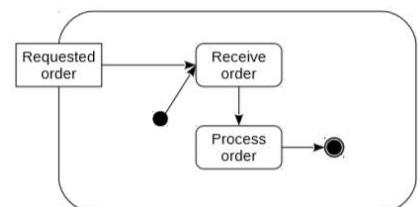


**Esempio:** Una volta che fill order è terminato viene prodotto un dato (order) che viene utilizzato da review order.

## Activity parameter node

Gli activity parameter nodes sono object nodes situati o all'inizio o alla fine del flusso, in quanto rappresentano che la prima azione per avviarsi necessita di un parametro (che loro stessi forniscono) o che l'ultima genera un output.

NOTAZIONE: si trovano sulla linea del rettangolo che rappresenta l'attività.



## Input/output pins

Input pins sono object nodes che ricevono valori da altre azioni attraverso object flows.

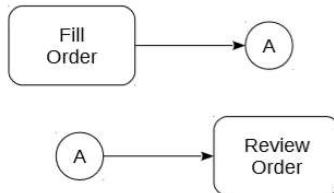
Output pins sono object nodes che ritornano valori a altre azioni attraverso object flows.



## Connectors

Notazione che permette di evitare di disegnare degli edge lunghissimi e semplificare la grafica dell'AD. Sono elementi puramente notazionali, quindi non hanno nessun effetto sul diagramma.

I connettori vengono rappresentati attraverso un piccolo cerchio con un nome all'interno.

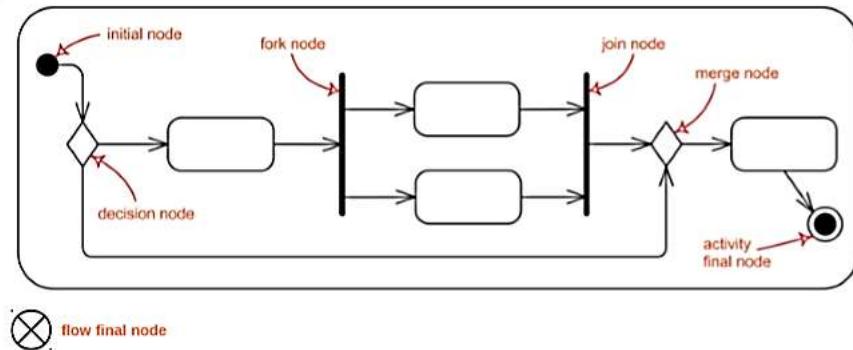


## Control node

Un control node è un activity node usato per coordinare i flussi fra altri nodi.

Sono control node:

- Initial node → specifica il punto in cui comincia il flusso
  - Flow final node → termina il flusso e distrugge tutti i token arrivati.
  - Activity final node → specifica il termine del processo che stiamo modellando.
  - Decision node → ha uno o più archi entranti e uno o più archi uscenti. Quando entra un token per uno qualunque degli archi entranti, questo uscirà su **UNO** qualunque degli archi uscenti.
  - Merge node → ha due o archi entranti e un arco uscente. Quando un token entra per un arco entrante questo viene rimandato su un arco uscente.
- È possibile avere un node che è sia decision che merge: ha più archi entranti e più archi uscenti.
- Fork node → Quando arriva un token sull'arco di ingresso viene presentato un token su ogni arco di uscita. Ovvero entra un unico flusso e si attivano più flussi concorrenti.
  - Join node → Quando si presenta un token di ingresso su ognuno degli archi entranti, il token di uscita viene generato sull'arco uscente. Viene effettuata anche una sincronizzazione temporale, solo quando tutti i token sono arrivati viene attivato l'arco di uscita.



## Structured activity nodes

Structured activity nodes sono nodi che contengono altri nodi, ovvero presentano una struttura particolare.

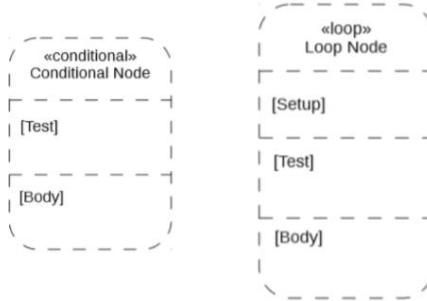
Un nodo non può essere contenuto direttamente da più structured node e structured node possono contenere altre structured node.

Conditional node → Un conditional node è un structured activity node che rappresenta una scelta esclusiva fra un certo numero di alternative (è una specie di if).

L'azione di body viene eseguita solamente se la condizione di test risulta vera.

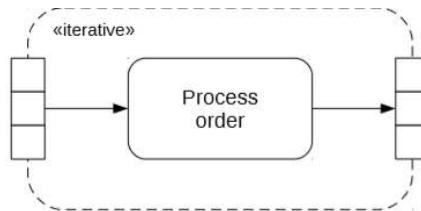
Loop node → Un loop node è un structured activity node che rappresenta un loop con setup, testo e body (è una specie di for).

Dopo il setup il corpo viene eseguito finchè la condizione di test risulta vera.



Expansion region → un expansion region è un structured node che prende in input una collezione e su ogni elemento di questa agisce individualmente producendo come output una collezione.

Tale procedimento può avvenire sequenzialmente (“iterative”), concorrentemente (“parallel”) o attraverso uno stream (“stream”). → La modalità deve essere sempre esplicitata in alto a sinistra.



## Activity partition

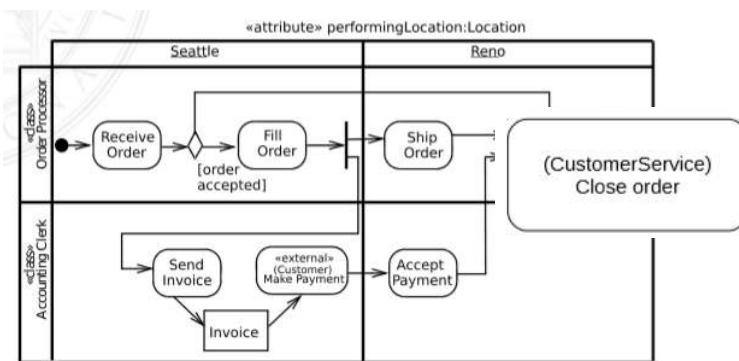
L'attività può essere partizionata per sottolineare che un determinate azioni hanno alcune caratteristiche in comune. → **Esempio:** azioni eseguite dallo stesso attore.

Per creare una partizione è sufficiente dividere l'attività con una linea (orizzontale o verticale).

Alle partizioni vengono applicati dei vincoli (**Esempio:** Azioni eseguite dal customer), quelli normativi in UML sono: classificatori, specificazione dell'istanza e proprietà.

Il vincolo ruolo è spesso usato, ma non è definito nelle specifiche.

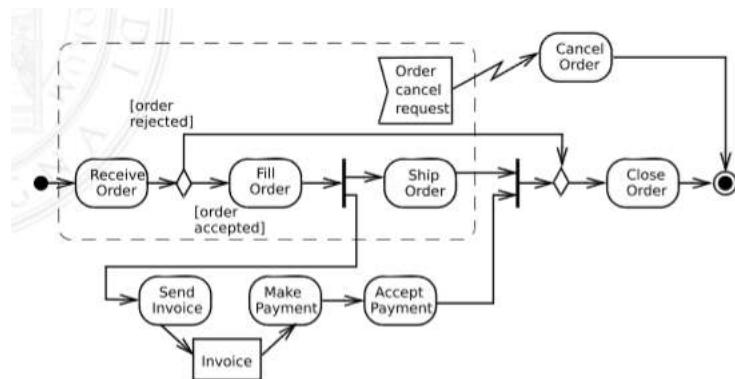
Invece del partizionamento, è possibile scrivere prima del nome dell'azione il nome dell'attore/ classificatore fra parentesi.



## Interruptible regions and interrupting edges

Interruptible regions (regione interrompibile) è un tipo di activity region che definisce un meccanismo per il quale se avviene un certo evento, mentre gli elementi all'interno di tale regione sono attivi, allora queste attività devono essere interrotte.

Interrupting edges serve a specificare l'evento che fa interrompere la regione ed eventualmente che azione intraprendere dopo l'interruzione. → È rappresentato da un arco a forma di fulmine.



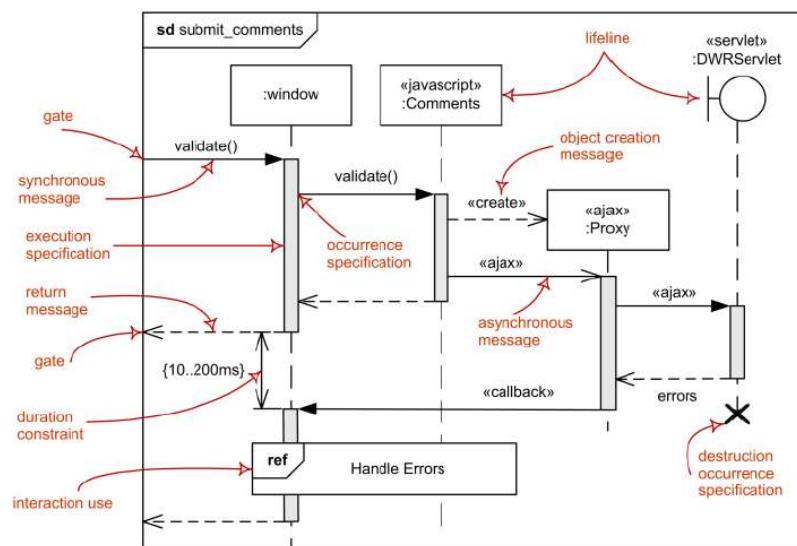
Considereremo gli acitvity diagram solo per modellare processi e casi d'uso, ma possono essere utilizzati nel dominio della soluzione.

## UML: integration diagram

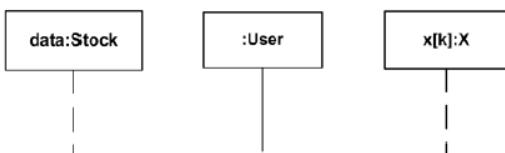
### Sequence diagram

Questo diagramma fa parte della più comune categoria dei diagrammi di interazione, esso si concentra sullo scambio del messaggio tra un numero di lifeline. Il diagramma di sequenza descrive un'interazione concentrandosi sulla sequenza dei messaggi che sono scambiati e le loro occorrenze specifiche sulle lifeline. Esso è un elaborato che può essere creato rapidamente e facilmente che illustra eventi input e output relativi ai sistemi in discussione. Costituisce, inoltre, un input per i contratti delle operazioni e soprattutto per la progettazione degli oggetti. UML fornisce la notazione dei diagrammi di sequenza per illustrare eventi generati dagli attori esterni al sistema.

### Elementi del sequence diagram:



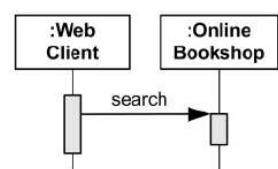
La **lifeline** è un elemento che rappresenta un singolo partecipante nell'interazione. Mentre le caratteristiche strutturali ed altre parti possono avere molteplicità maggiore di 1, la lifeline rappresenta solo un'entità dell'interazione. Essa rappresenta principalmente delle istanze.



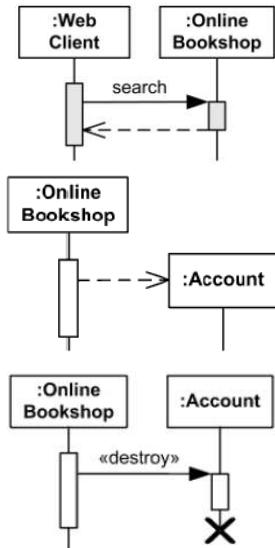
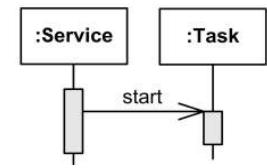
Un **messaggio** è un elemento che definisce uno specifico tipo di comunicazione tra lifelines di un'interazione ed è rappresentato da una freccia. Il messaggio specifica non solo il tipo di comunicazione, ma anche il mittente e destinatario. Essi sono normalmente due specifiche dell'occorrenza. Un message rispecchia anche un'operazione call e start di un'esecuzione o invio/ricezione di un signal.

In base al tipo di azione che il messaggio genera, il messaggio può essere:

- Una chiamata sincrona: tipicamente rappresenta operazioni di chiamata/invio (call/send) di messaggi e si sospendono le esecuzioni in attesa della risposta. Nelle chiamate sincrone i messaggi sincroni sono mostrati con una freccia continua con "testa" colorata in nero. È dunque una richiesta per attivare una funzionalità, chi invia rimane poi in attesa di risposta.

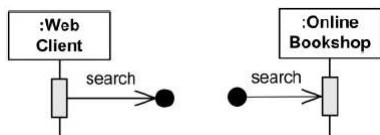


- Una chiamata asincrona: si manda il messaggio e si continua immediatamente senza aspettare la risposta. I messaggi asincroni sono rappresentati da una freccia con la “testa” aperta. In sostanza specifica che una lifeline corrisponde ad un elemento, genera un altro elemento con una nuova lifeline e poi potrà interagire con quello. È un messaggio che serve per modellare la istanziazione di nuovi elementi.
- Un segnale asincrono: diverso da messaggio asincrono, il messaggio è una richiesta, non un’invocazione di una specifica richiesta.
- Un reply message: il messaggio di risposta ad una operazione call è mostrato come una linea tratteggiata e freccia con la “testa” aperta (simile alla creazione del messaggio).
- Un create message: il messaggio di creazione è mandato alla lifeline in modo che lo crei lei stessa. È pratica comune mandare il create message ad un oggetto non esistente per farlo creare a lui. In sostanza è uno pseudomessaggio per dire che un elemento è generato da una lifeline.
- Un delete message: il messaggio di eliminazione è mandato per terminare un’altra lifeline. La lifeline di solito finisce con una croce a forma di X in basso che denota la distruzione dell’occorrenza. È dunque simile a create ma elimino il messaggio con un meta messaggio <destroy>.



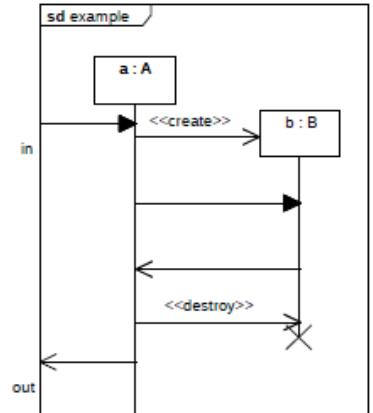
### Lost & founds

Si spedisce il messaggio a qualcuno che non si vuole modellare oppure si ricevono messaggi da qualcuno di non modellato. C’è un punto con una lifeline.



Alcuni messaggi provengono dal bordo del rettangolo cioè il **gate** che è la fine di un messaggio, è un punto di connessione/comunicazione per relazionare un messaggio fuori da un frangimento dell’interazione con un messaggio all’interno.

Non tutti gli elementi dell’iterazione esistono dall’inizio.



Un **interaction fragment** è un elemento che rappresenta la più generale unità di interazione, esso descrive cosa avviene in un sistema. Ogni interaction fragment è concettualmente uguale ad un’interazione. Non c’è una notazione generale per un interaction fragment. Le sue sottoclassi definiscono la loro stessa notazione.

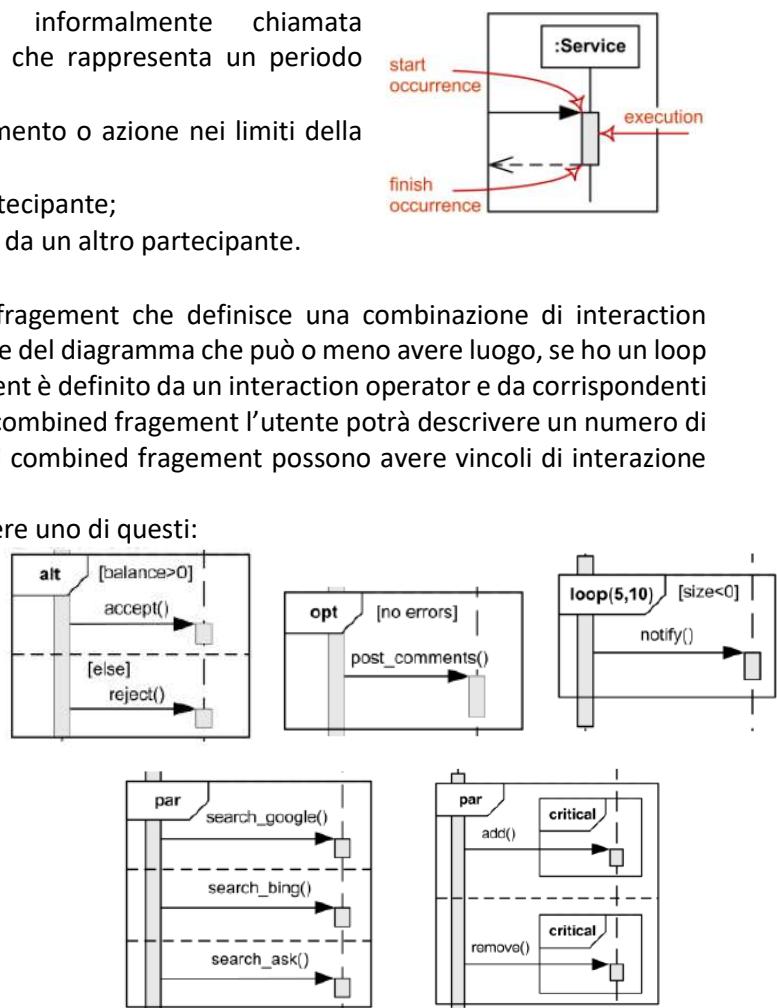
**Esempio:** di interaction fragment sono: occurrence, execution, state invariant, combined fragment, interaction use.

- Occurrence (occurrence specification): è un’interaction fragment che rappresenta un momento nel tempo (evento) all’inizio o alla fine di un messaggio o all’inizio/fine di una execution. È dunque un periodo dove è eseguita un’azione o spedito un segnale o se ne aspetta uno ed è rappresentato da rettangoli nella lifeline. Una occurrence specification è una delle unità semantiche di base delle interazioni. Il significato di interazione è specificato da sequenze di occurrences descritte da occurrence specification. In sostanza è un evento che avviene in uno specifico momento.

- Execution (execution specification, informalmente chiamata activation): è un'interaction fragment che rappresenta un periodo nella lifetime del partecipante in cui:
  - Si esegue una unità di comportamento o azione nei limiti della lifetime;
  - Si manda un signal ad un altro partecipante;
  - Si aspetta una risposta al message da un altro partecipante.
- Combined fragment è un'interaction fragment che definisce una combinazione di interaction fragment. Serve per definire una regione del diagramma che può o meno avere luogo, se ho un loop ha luogo più volte. Un combined fragment è definito da un interaction operator e da corrispondenti interaction operands. Durante l'uso dei combined fragment l'utente potrà descrivere un numero di tracce in maniera compatta e concisa: i combined fragment possono avere vincoli di interazione (guards).

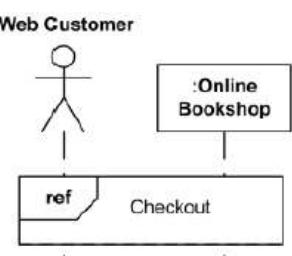
Gli operatori di interazione possono essere uno di questi:

- Alt – alternativa
- Opt – opzione
- Loop – iterazione
- Break – break
- Par – parallelo
- Strict – strict sequencing
- Seq – weak sequencing
- Critical – critical region
- Ignore – ignora
- Consider – considera
- Assert – asserzione
- Neg – negativo



Una **interaction use** è un interaction fragment che consente di usare (o chiamare) un'altra interazione. Larghe e complesse sequenze di diagrammi possono essere semplificate grazie ad essi. È inoltre comune il riutilizzo di alcune interazioni tra parecchie altre interazioni.

Nei diagrammi di sequenza **non** bisogna generalizzare troppo le sequenze. Inoltre un use case può essere descritto da più di un solo sequence diagram. Si deve modellare solo quello a cui si è interessati. Infine il tipo di azione del messaggio può essere decisa al momento del design.



In conclusione il diagramma di sequenza serve per modellare il dominio della soluzione, il livello di dettaglio è molto ricco nel descrivere ciò che avviene. Con esso si vuole specificare una possibile traccia di esecuzione, non tutte perché altrimenti si complicherebbe. Se ad es. si mettessero 2 segmenti nidificandoli diventerebbe troppo complicato. Nel dominio del problema, per modellare i casi d'uso più o meno complessi, questa soluzione si può usare per rappresentare l'happy path o le altre, anche per lo use case, ma se è complesso sarà difficile da leggere. Se ne esce fuori un use case semplice si può usare, altrimenti se ne usa uno in più dividendo in sequenze alternative. Uno use case può essere modellato da più SD.

### Diagrammi di sequenza di sistema

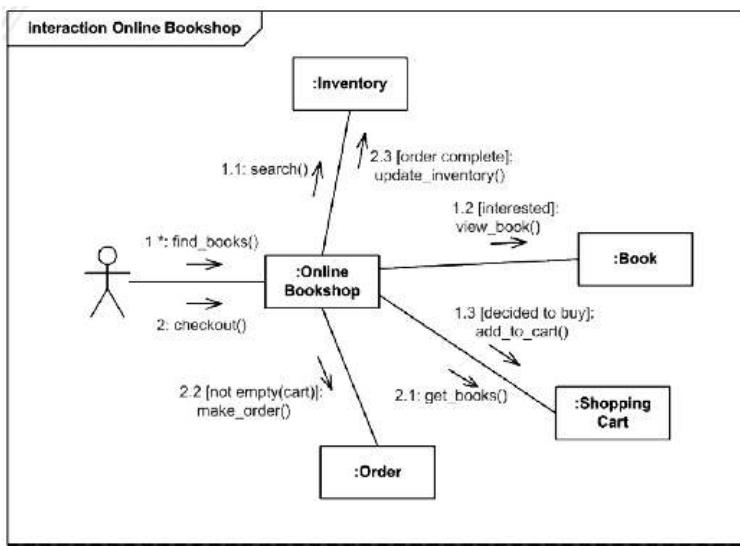
I casi d'uso descrivono il modo in cui gli attori esterni interagiscono con il sistema sw che interessa creare. Durante questa interazione un attore genera eventi di sistema a un sistema, solitamente per richiedere l'esecuzione di alcune operazioni di sistema definite per gestire l'evento. Quell'evento dà inizio a un'operazione sul sistema. UML include i **diagrammi di sequenza** come notazione in grado di illustrare interrogazioni tra attori e le operazioni iniziate da essi. Un **diagramma di sequenza di sistema** è una figura

che mostra, per un particolare scenario di un caso d'uso, gli eventi generati da attori esterni, il loro ordine e eventi inter-sistema. Tutti i sistemi sono considerati a scatola nera; il diagramma enfatizza gli eventi che superano i confini dei sistemi, dagli attori ai sistemi.

Fondamentalmente un sistema sw reagisce a 3 cose: eventi esterni da parte di attori umani o sistemi informatici, eventi di timer e guasti/eccezioni. È pertanto utile sapere con precisioni quali siano gli eventi esterni di input (ovvero eventi di sistema). Essi rappresentano una parte importante dell'analisi del comportamento del sistema. Il comportamento del sistema è una descrizione di cosa fa il sistema, senza spiegare come lo fa. Un diagramma di sequenza è una parte di questa descrizione.

### Communication diagram

Questo diagramma mostra le interazioni tra le lifeline che usano un aggiustamento in forma libera (free-form arrangement). Quindi non si hanno lifeline con corrispondenza spazio/tempo ma sono free-form. I diagrammi di comunicazione possono essere convertiti in diagrammi di sequenza, ma non il contrario. I communication diagram quindi servono per fare la stessa cosa dei sequence diagram ma sono meno potenti. In questo caso si usa una numerazione per rappresentare l'orizzonte temporale. Si presume che i messaggi sono ricevuti nello stesso ordine in cui sono stati generati. Un communication diagram può contenere frames, lifeline e messaggi. Si usa questo diagramma se si hanno iterazioni più semplici, è più immediato dell'altro se si conoscono le notazioni.



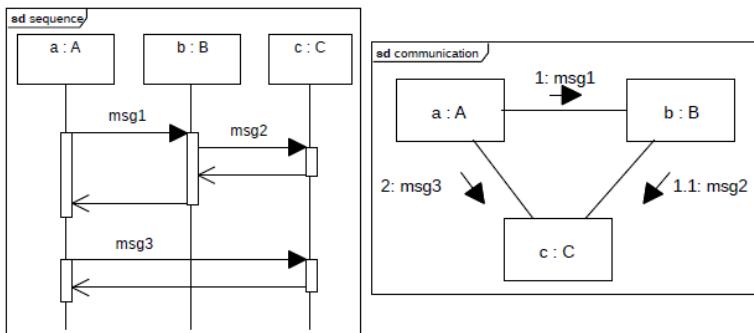
La notazione dei **messaggi** nei communication diagram segue le stesse regole dei sequence diagram. L'evoluzione dei messaggi di sistema ha una **sequence expression**: essa serve per capire l'ordine temporale dei messaggi.

Sequence-expression ::=

Sequence-term '.' . . . ':' message-name

Sequence-term ::= [integer [name]] [recurrence]

Le sequence terms sono usato per rappresentare l'annidamento dei messaggi fra le interazioni, essi sono formati da un numero integer + una ricorrenza.



### Concorrenza e ricorrenza

Sequence-term ::= [integer[name]] [recurrence]

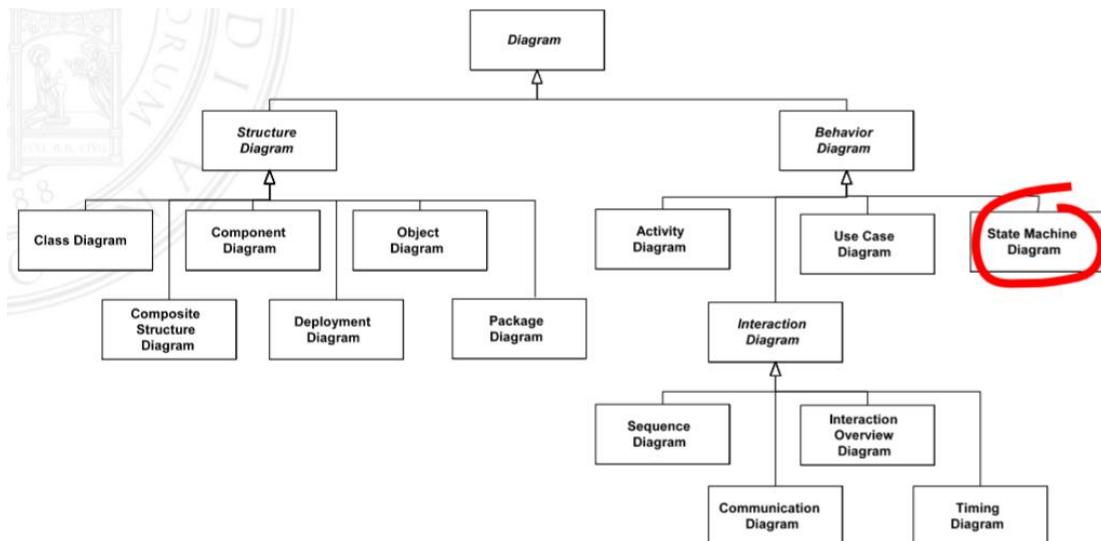
→ Messaggi che differiscono solo per il nome sono considerati concorrenti.

Recurrence ::= branch | loop

branch ::= '[' guard ']'

→ Le guards specificano condizioni che devono accadere al messaggio.

## 10. UML: STATE MACHINE DIAGRAMS



Come i diagrammi di attività, i diagrammi a stati di UML mostrano una vista dinamica, fanno parte dei diagrammi comportamentali e possono essere usati per l'analisi e per la progettazione, ma sono più usati per la progettazione.

Un diagramma di macchina degli stati di UML illustra gli eventi e gli stati interessanti per un oggetto, e il suo comportamento in reazione a un evento, mostra cioè il ciclo di vita di un oggetto. Se si verifica un evento che non è rappresentato nel diagramma, l'evento viene ignorato per quanto riguarda il diagramma di macchina a stati → è possibile creare un diagramma di macchina a stati che descrive il ciclo di vita di un oggetto a livelli che possono essere semplici o complessi.

In senso ampio, le macchine a stati sono applicate in due modi:

1. Per modellare il comportamento di un oggetto reattivo complesso, in risposta agli eventi → Avremo quindi la Behavioral state machine, che rappresentano in modo dinamico il sistema, o parte di esso, attraverso un modello descritto in termini di un grafo, contenente dei vertici di varia natura che possono essere degli stati connessi tra di loro attraverso delle transizioni. Ci illustra cioè qual è il comportamento di un sistema o sottosistema.
2. Per modellare le sequenze valide delle operazioni, ovvero specifiche di protocollo o di linguaggio → Avremo la Protocol state machine, usa la stessa notazione della precedente ma definisce quali sono le interazioni valide che si possono determinare fra un elemento e altri elementi all'interno della situazione. Ci mostra quindi come gli altri interagiscono con un elemento.

### Stati

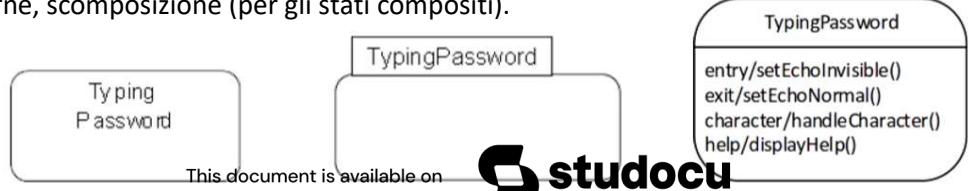
Stato → Condizione di un oggetto in un certo intervallo di tempo, il tempo tra due eventi. È caratterizzato da un insieme di reazioni e risposte di un certo tipo.

→ Dire che un sistema è in un certo stato vuol dire che offre le stesse risposte di altri nello stesso stato, altrimenti sono in stati differenti.

Se un oggetto risponde sempre nello stesso modo ad un evento o per tutti gli eventi di interesse, viene considerato indipendente dallo stato.

Al contrario, gli oggetti dipendenti dallo stato reagiscono in modo diverso agli eventi a seconda del loro stato o modo.

**NOTAZIONE:** Gli stati sono rappresentati con rettangoli arrotondati, ma possono essere internamente suddivisi in più scomparti con linee orizzontali e, in tal caso, nella zona più in alto (oppure sopra al rettangolo) va riportato il nome dello stato, mentre gli altri compartimenti sono per: nome, comportamento interno, transizioni interne, scomposizione (per gli stati composti).



Gli stati possono essere:

- Semplici → senza vertici interni o transazioni
- Compositi → contengono una o più regioni, gli stati in queste regioni sono chiamati sottostati
- Descritti da macchine

Gli stati possono essere associati ai seguenti comportamenti:

- enter → si attiva quando il sistema entra in uno stato
- exit → si attiva quando il sistema esce da uno stato
- doActivity → si attiva quando il sistema è nello stato, esegue in modo concorrente ad altri stati e se il sistema esce dallo stato prima che completi, allora si interrompe.

## **Transizioni**

In un diagramma gli stati sono collegati da transizioni.

Transizione → è una relazione tra due stati che descrive in modo atomico che quando si verifica un evento, l'oggetto passa da uno stato all'altro.

→ Normalmente hanno luogo in funzione di eventi, quando l'evento si verifica la transizione abilita il passaggio del sistema da uno stato all'altro.

Una transizione può causare l'avvio di un'azione, in un'implementazione software questo può rappresentare l'invocazione di un metodo della classe a cui è associato il diagramma di macchina a stati.

Una transizione può essere caratterizzata anche da una guardia condizionale, ovvero un test booleano.

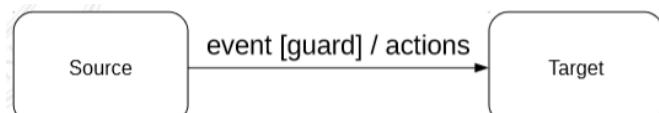
Quindi, se l'evento ha luogo:

- se la guardia è vera → la transizione è abilitata
- se la guardia è falsa → la transizione non avviene

Può anche succedere che in una transizione sia specificata la guardia ma non l'evento, questo non vuol dire che non c'è ma si assume l'evento di terminazione dei comportamenti che caratterizzano lo stato.

**NOTAZIONE:** Le transizioni sono mostrate come frecce, etichettate con il rispettivo evento.

Legano uno stato sorgente a uno destinazione, specificano (o meno) l'evento, una eventuale guardia (che deve essere vera) e si possono specificare delle azioni che hanno luogo durante il cambiamento di stato.



Le transizioni, dal punto di vista della semantica del sistema, sono atomiche, cioè il token si trova allo stato sorgente o allo stato destinazione, ma come abbiamo visto, si possono associare dei comportamenti, quindi attraversano anche esse degli stati. → Cioè una transizione può essere raggiunta, attraversata o completata. Triggers → determinano il fatto che la transizione venga raggiunta, sono associati ai tipi di eventi diversi che si possono verificare e si abilitano successivamente a questi.

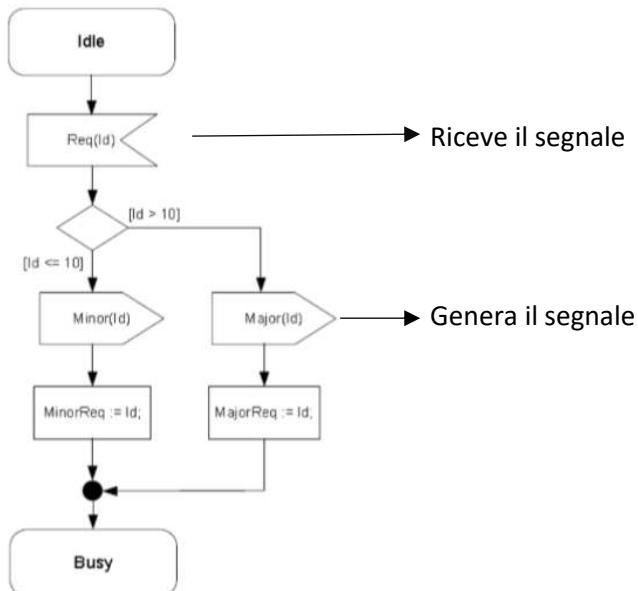
Le guardie sono valutate prima che una transizione abbia luogo, a meno che non partano da uno pseudostato. Infatti, se alla partenza ci sono degli pseudostati in catena, abbiamo solo il momento subito prima e il momento subito dopo allo pseudostato, appunto perché il token non viene trattenuto. → Quindi avremmo una catena di transizioni che non si possono fermare. Perciò:

- Guardie fra stati → Possono bloccare il token
- Guardie fra pseudostati → Non possono bloccare il token

`{<trigger>}* ['['<guard>']] [<behavior-expr>]`

È possibile anche specificare degli elementi che corrispondono alla generazione e alla ricezione di segnali.

→ Action signal e send signal



In generale è meglio non usare questi elementi perché rischiano di farci confusione!

## Eventi

Evento → è un avvenimento significativo o degno di nota, cioè un'occorrenza osservabile che avviene in un momento specifico. → Possono avere dei parametri.

**Esempio 1:** Premere un tasto.

**Esempio 2:** Premere il tasto 3. → 3 è il parametro.

Possono essere:

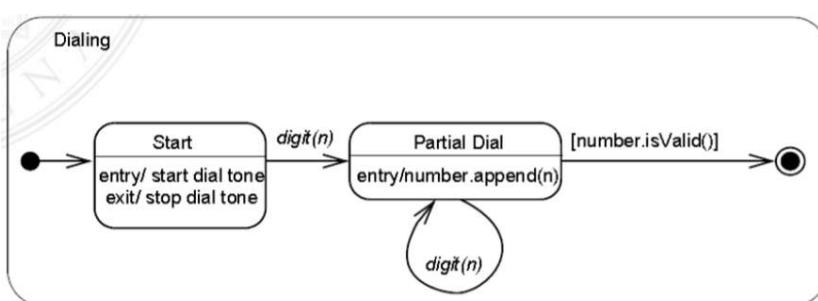
- MessageEvent (CallEvent, SignalEvent) → legati a dei messaggi che vengono scambiati
- ChangeEvent → legati al cambiamento di stato
- TimeEvent → legati al tempo, quindi periodici o con scadenza ecc.

## Stato finale e Pseudostati

Oltre a stati, transizioni ed eventi, in un diagramma di macchina a stati ci sono anche:

- Pseudostati → sono simili ai nodi del controllo del flusso, ma hanno una semantica diversa. La differenza fondamentale con gli stati è che non sono in grado di trattenere il token, che quindi "scivola" subito fuori. → initial, join, fork, junction, choice, entryPoint, exitPoint, terminate.
- Stato finale → definisce il completamento del comportamento modellato attraverso il diagramma.

**Esempio:** State Machine Diagram che modella un telefono (cornetta).



Ci sono due stati: Start State e Partial Dial State, con in mezzo una transizione.

C'è uno pseudostato iniziale, che non è in grado di trattenere il token e quindi permette di passare subito, attraverso la transizione, allo stato Start.

Start State: viene descritto un comportamento **entry**, quando arriva il token (cioè appena si solleva la cornetta) parte il **dial tone**, e un comportamento **exit**, quando il token esce dallo stato avviene **stop dial tone**. **Digit(n)**: è un evento parametrico, lo stato del sistema evolve quando si digita il numero. Prende il token da Start e lo mette in Partial Dial State.

Partial Dial State: c'è una transizione che non va da nessuna parte, ma parte e torna allo stesso stato ed è associata all'evento **digit(n)**. → Mentre siamo in questo stato se viene premuto un tasto di rimane nello stesso stato, e aggiunge il numero premuto al parametro con cui sono entrato nello stato.

**NB**: Stati differenti si caratterizzano per rispondere in modo differente allo stesso stimolo → L'evento **exit** di Start e di Partial Dial è lo stesso, ma nel primo caso poi il token passa allo stato successivo, mentre nel secondo caso il token resta nello stesso stato.

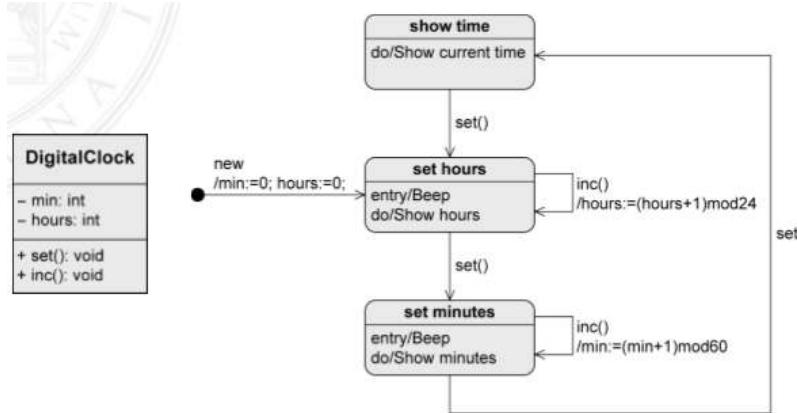
[**number.isValid()**]: transizione che permette di uscire da Partial Dial State, viene specificata la guardia ma non l'evento. → Quindi si intende per evento, quello che ha luogo quando sono terminati i comportamenti dello stato di partenza.

Quando termina **number.append(n)** avviene l'evento, se [**number.isValid()**] è falso l'evento va perso e la transizione non ha luogo; se [**number.isValid()**] è vero l'evento va perso e la transizione ha luogo.

Stato finale: il token entra e non va via.

→ Questo potrebbe essere un design model non ben raffinato, o un analysis model.

**Esempio: Modellazione di un orologio digitale.**



Ci sono 3 stati differenti, perché con l'evento **set()** si hanno tre reazioni differenti.

Pseudostato iniziale: vengono specificate le azioni che si determinano quando si mette il token allo stato iniziale, che setta gli attributi a 0.

set hours: il comportamento di all'ingresso è **Beep**, il comportamento che si ha mentre il token rimane nello stato è **Show hours**. Quando si preme nuovamente **set()** il doActions viene interrotto e si passa allo stato successivo. Se si preme **inc()** viene aggiornata l'ora in modulo 24, quindi entra ed esce dallo stato.

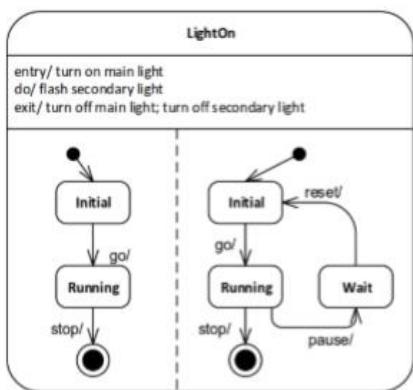
set minutes: il comportamento di all'ingresso è **Beep**, il comportamento che si ha mentre il token rimane nello stato è **Show minutes**. Quando si preme nuovamente **set()** il doActions viene interrotto e si passa allo stato successivo.

show time: mostra l'ora finchè si rimane in questo stato.

**NB**: il comportamento do è continuativo fintanto che si rimane in quello stato.

→ Questo è un design model, cioè descrive come funziona il sistema nel dominio della soluzione.

Questi diagrammi possono essere usati per descrivere in maniera molto dettagliata un problema nel dominio della soluzione, ma serve un formalismo molto puntuale.



Per questo lo stato può essere diviso in regioni, in maniera gerarchica o ortogonale, che definiscono due sottomacchine, cioè due comportamenti che hanno luogo in modo concorrente mentre il sistema si trova in uno stato.

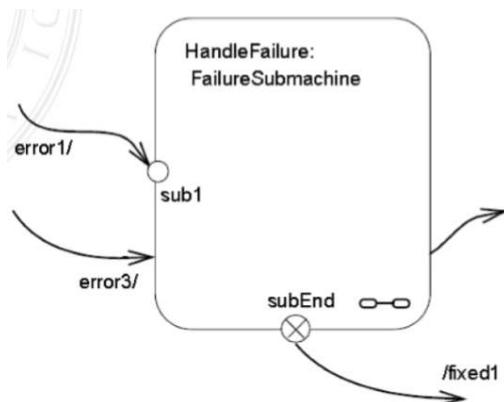
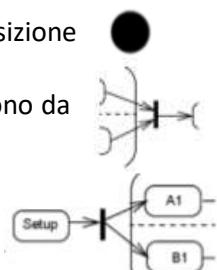
Quindi LightOn è uno stato, quando il sistema si trova in questo stato il comportamento interno viene descritto da due macchine stati combinate, in maniera concorrente.

→ Questo non è l'unico modo per specificare comportamenti concorrenti.

Sono notazioni che si possono applicare anche al dominio del problema.

## Pseudostati

- Initial → Quando parte il sistema viene messo un token nello pseudostato iniziale, la transizione all'uscita da questo pseudostato non può essere associata né un evento né una guardia.
  - Join → È il duale del fork, serve per rimettere insieme due comportamenti che provengono da regioni ortogonali differenti.
  - Fork → Divide una transizione in entrata in una o più transizioni, siccome non ci possono essere due token attivi in una state machine, quindi per modellare la concorrenza ci devono essere due stati ortogonali che vengono puntati dal fork.
- È meglio non usare questi ultimi due pseudostati, perché sono molto potenti e hanno una notazione semantica complessa.
- Choice → Rombo; il token entra ed esce in una delle due parti, alternativamente. È attivata quando hanno termine i comportamenti dello stato precedente e si ha la scelta. È equivalente all'uso delle guardie.
  - Anche questo può creare confusione, quindi è meglio non usarlo.
  - EntryPoint → È per sottomacchine o stati composti che possono avere un punto di ingresso.
  - ExitPoint → È per sottomacchine o stati composti che possono avere un punto di uscita.
  - Terminate → Per far terminare TUTTO il comportamento della macchina anche se ci sono stati concorrenti attivi nello stesso momento.



→ Quando c'è una sottomacchina, queste possono avere più EntryPoint o ExitPoint che servono a gestire casi particolare, eccezioni o altro.

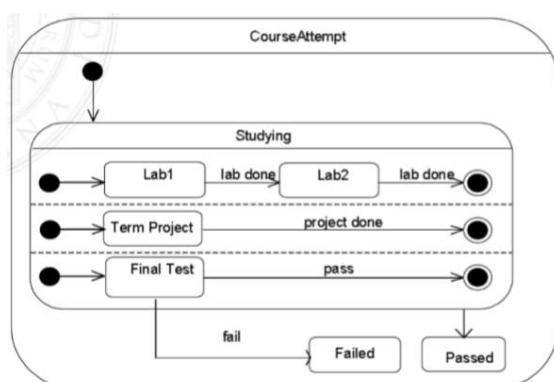
History pseudostate → è un altro tipo di pseudostato usato per tener traccia dello stato di una regione. Permettono quindi di uscire da una regione, per poi rientrarvi con lo stesso stato che si aveva al momento dell'uscita.

- deepHistory pseudostates → consente di ripristinare lo stato della regione in si era, e in tutte le sottoregioni delle sottomacchine che possono descrivere la regione.
- shallowHistory pseudostates → ripristina solo lo stato principale, gli altri vengono persi.



## Alcuni esempi di State Machine più dettagliate

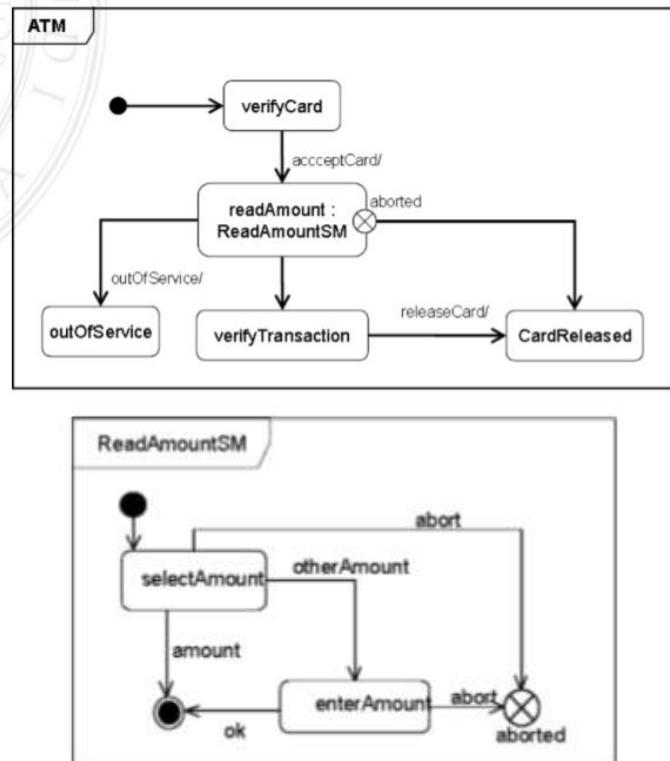
Esempio: Studio



Questo State Machine descrive il comportamento di uno studente che deve superare il corso.

Mentre il sistema è nello stato **Studying**, il comportamento è quello descritto dalle macchine di stati. Quindi, mentre lo studente studia fa **Lab1**, è in **Team Project** e faccio **Final Test**, oppure sono in **Lab2** sono in **Team Project** e fa **Final Test**. Se passa **Final Test** ha finito, la transizione ha luogo senza eventi specificati, altrimenti va in uno stato **Failed** che non è un suo sottostato ma è un sottostato di tutto **CourseAttempt**, perché se non passa test finale fallisce tutto il corso.

Esempio: funzionamento bancomat.



Non stiamo modellando che cosa avviene se la carta non è accettata ma il nostro obiettivo è il funzionamento del bancomat.

Metto la tessera, quando viene accettata il sistema è in **readAmountSM** (cioè State Machine), se arriva l'evento **outOfService** qualunque cosa stavo facendo il sistema esce dallo stato e va in **outOfService**.

Alternativamente, quando ha finito l'input da tastiera da parte dell'utente, da **readAmount** passo in **verifyTransaction**.

Viene eseguito **verifyTransaction**, quando finisce viene rilasciata la carta e si passa in stato **CardReleased**, e si suppone che lasci i soldi.

Se da **readAmount** viene attivato **aborted** che è un exitState, si passa direttamente a **CardReleased**.

### Compound transition e run to completion (RTC)

L'elaborazione di un singolo evento può innescare diverse transizioni che attraversano diversi stati o pseudostati. In generale le State Machine lavorano facendo delle transizioni, se esse passano per pseudostati in realtà avremo delle transizioni che sono composite, ossia compound transition.

### Compound transition

Per arrivare dallo stato di partenza a quello di destinazione, le transizioni passano attraverso gli pseudostati, che però non tengono il token quindi avviene tutto in maniera atomica. → Si dice che la semantica avviene per passi di stati compositi e prende il nome di run to completion (RTC).

### RTC

Al momento della creazione, una macchina a stati esegue la sua inizializzazione eseguendo la transizione composta iniziale, entra quindi in un wait point, in cui il token si trova all'interno di uno stato, e grazie a degli step si passa ad un altro wait point. → Gli step sono transizioni semplici o composite.

Quando gli eventi vengono inviati, i trigger vengono valutati e, se è possibile attivare almeno una transizione, viene eseguita una nuova transizione composta (un passaggio) e viene raggiunto un nuovo punto di attesa.

Tutto viene ciclato finché i token non si arriva al Final State o qualcuno non blocca in modo asincrono il funzionamento.

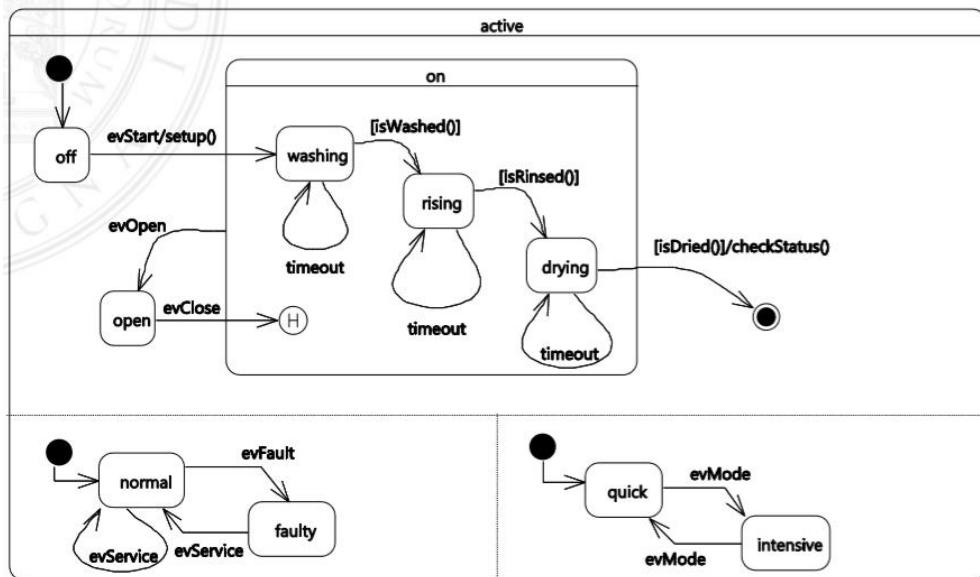
**NB:** Un singolo evento può attivare più transizioni, se queste transizioni hanno un'intersezione di stati di uscita non vuoti, sono in conflitto. Le priorità delle transizioni in conflitto sono basate sulla loro posizione relativa nella gerarchia di stati.

Il modello RTC semplifica la gestione della concorrenza nelle macchine a stati: quando la macchina si trova in uno stato non ben definito, l'invio dell'evento non avviene e viene memorizzato in una cosa di eventi.

→ Questo modello è stato ereditato dalla notazione di Harel che ha definito tutta questa complessa semantica di State Machine. È la cosa più complicata che c'è in tutto UML, ma è anche una notazione molto potente.

→ È alla base del funzionamento delle automobili, di tutta l'automotiva, del mondo militare, ecc..

Esempio: Dishwasher → Questo esempio è il minimo e il massimo della complessità che viene richiesto.



Questa State Machine descrive il funzionamento di una lavastoviglie in termini di tre macchine a stati concorrenti, cioè in ogni istante di tempo la lavastoviglie potrebbe essere in stato **washing**, **normal** e **quick**, oppure in stato **rising**, **normal** e **quick**, oppure in **drying**, **normal** e **quick**. → Lo stato attivo della lavastoviglie è composto dallo stato delle tre macchine a stati.

- La prima macchina descrive il funzionamento principale della macchina. All'inizio la macchina è in stato **off**, quando viene premuto il tasto di accensione (**evStart**) esegue il comportamento **setup()**, si ha la transizione ed entra in stato **on**. Alla fine dello stato **on** arriva al Final State ed esegue l'ultima operazione.

Stato on: descritto da una state machine, si entra nello stato **washing** per passare allo stato successivo la macchina deve eseguire l'operazione **[isWashed()]**, che non è un evento.

Per passare da uno stato ad un altro abbiamo bisogno di un trigger, qui c'è solo la guardia, allora si passa a **rising** quando **washing** ha terminato la sua esecuzione, che però non sappiamo qual è, quindi vuol dire che appena il token entra in quello stato esce subito, la guardia effettua il controllo e se non è valida il token torna allo stato precedente. → Per questo c'è **timeout**, che fa ripartire **washing** ogni tot tempo.

Stessa cosa avviene per il passaggio da **rising** a **drying** ed arriva poi allo stato finale.

Mentre avviene lo stato **on**, può essere attivato **open** con l'evento **evOpen**, questo stato ruba il token e quando lo sportello viene richiuso (**evClose**), grazie all'history state (**H**), viene ripristinato lo stato della macchina in **on**.

- La seconda macchina dice che di solito la macchina è in stato **normal**, se avviene un evento **evFault**, attraverso il quale la macchina si accorge che c'è un problema, allora passa in stato **faulty** finché non avviene un evento **evService**. → **evService** può avvenire anche senza che si verifichi **evFault**.
- La terza macchina descrive la modalità di lavaggio: **quick** o **intensive**, il passaggio da uno stato all'altro avviene con **evMode**, si preme il bottone "mode" e ogni volta cambia stato, e può essere premuto in qualsiasi momento.

→ Avvengono in modo l'una indipendente dall'altra.

## **11. INTO OO DESIGN**

### **Nella soluzione del dominio**

Ora che sappiamo che cosa i sistemi dovrebbero fare, dobbiamo progettare un sistema che svolga bene questo lavoro. Ci sono altri metodi di modellamento: vediamo ora il cosiddetto metodo semplificato.

### **Metodi di ispirazione UP super semplificati**

Essi valgono per ogni caso d'uso e per ogni diagramma di sequenza di casi d'uso presente nel sistema. Bisogna seguire certi passi:

- Dividere il sistema in interface layer (parte dell'interfaccia) e business logic/modello di dominio. L'interface layer riceve le interazioni dagli utenti.
- Il sistema per rispondere ad un utente dovrà trovare/creare elementi nel business logic/domain layer che sono responsabili di supportare tutte le interazioni dell'interface layer.
- Si aggiungono man mano nuovi elementi/nuove classi/nuove operazioni alla design class diagram (alla soluzione).

Questi passaggi garantiscono la funzionalità del sistema, il sistema sarà quindi funzionante se non vogliamo interessarci alla sua qualità. Il metodo semplificato, quindi, fa sì che il sistema funzioni ma che non sia di buona qualità.

### **Lista dei modelli da presentare**

Se un committente ci commissiona una progettazione di qualche tipo, devono essere presentati vari modelli/artefatti nella fase di elicazione (tirare fuori la conoscenza dal committente):

- Modello dei requisiti;
- Modello dei casi d'uso;
- Modello di dominio;
- Diagramma di sequenza.

## 12. SOFTWARE QUALITY E OBJECT ORIENTED PRINCIPLES

Lo scopo delle attività legate alla progettazione è creare un software funzionante e con un elevato livello di qualità.

*Cosa si intende qualità di un software?*

Distinguiamo due categorie di qualità:

- Qualità esterne → direttamente percepibili da chi utilizza il sistema.
  - Aspetti funzionali → elementi del sistema che sono percepibili dall'utente e che hanno a che fare con cosa fa il sistema.  
**Esempio:** correttezza, ma non tutte le qualità esterne sono funzionali.
  - Aspetti non funzionali → elementi percepibili dall'utente che riguardano come il sistema fa qualcosa.
- Si possono avere due sistemi identici dal punto di vista funzionale, ma molto diversi rispetto a quello non funzionale.
- Qualità interne → riguardano come il sistema è organizzato internamente e non appaiono direttamente all'utilizzatore.

Qualità esterne:

- Correttezza → funzionale
- Usabilità → quale è lo sforzo richiesto all'utente per raggiungere i suoi obiettivi; è non funzionale. Più il sistema è usabile minore è lo sforzo che l'utente deve compiere per raggiungere l'obiettivo che si prefigge.
- Efficienza → come sono usate dal sistema le risorse; non funzionale.
- Integrità → solitamente è riferita ai dati; indica quanto i dati mantengono la loro integrità (eventuali vincoli rispettati ecc...).
- Affidabilità → valutata in termini di lap-time (tempo del giro). Più l'applicazione è critica più questa qualità risulterà critica. Non funzionale.
- Adattabilità → poter utilizzare il sistema in contesti diversi rispetto a quelli che si erano pensati.
- Accuratezza → ha molte sfaccettature es. precisione dei dati
- Robustezza → quanto il sistema è in grado di fornire funzionalità anche degradate in caso di guasti anche parziali.

In certi casi misurare queste qualità è difficile.

Qualità interne:

- Manutenibilità → costo di effettuare le piccole modifiche necessarie in un SW (es. bug fixing).
- Flessibilità → quale è lo sforzo necessario per introdurre nuove funzionalità nel sistema o cambiare in maniera rilevante funzionalità esistenti. Un sistema flessibile richiede poco sforzo.
- Portabilità → quanto costa il porting
- Ri-usabilità → quali opportunità si hanno di utilizzare il codice in un contesto diverso.
- Leggibilità → quanto è comprensibile ogni singola riga di codice. Un programma altamente leggibile richiede il minimo sforzo cognitivo per comprendere cosa si sta facendo.
- Testabilità → quale è lo sforzo per sottoporre il sistema a testing, che è un aspetto molto importante della validazione dei sistemi software. Minore è lo sforzo necessario, più test si possono fare.
- Comprensibilità → riguarda le scelte relative alla realizzazione della soluzione nel suo complesso. Consiste nel comprendere l'organizzazione del sistema e come i diversi elementi della soluzione interagiscono fra loro.

Il codice non deve essere scritto una volta sola. → I costi associati all'evoluzione del software sono alti e rappresentano circa il 50%-90% del costo totale di produzione del software.

Il fatto che esistano delle qualità del software e si possano definire, in alcuni casi, dei metodi per stimare tali qualità ha portato alla nascita di una serie di standard, tra cui ISO, che rientrano nella categoria "software product Quality Requirements and Evaluation" (requisiti di qualità e valutazione dei prodotti software).

Lo standard di riferimento utilizza un framework chiamato square ed è definito all'interno di ISO 25010 (evoluzione di ISO 9126 e ISO 14598) e definisce 3 modelli di qualità del software: software product quality mode, data quality model, quality in use model.

La software product quality serve a definire le caratteristiche che possono essere misurate internamente o esternamente: compatibilità funzionale, efficienza di performance, operabilità, sicurezza, compatibilità, mantenibilità e affidabilità.

Quality in use → qualità non funzionali e legate alla percezione dell'utente: soddisfazione, sicurezza, usabilità, efficienza e efficacia.

Un altro framework è "Consortium for IT software quality (SEI + OMG) che definisce software functional quality e software structural quality.

Software structural quality → comprende 5 principali caratteristiche desiderabili e necessarie per creare valore: efficienza, sicurezza, mantenibilità, dimensione e affidabilità.

Software functional quality → conformità a esplicati requisiti funzionali e usabilità (livello di soddisfazione sperimentato dall'utente finale).

*Abbiamo bisogno di un framework per stimare la qualità?*

A volte, ovvero nel caso di progetti di un certo livello di complessità può essere richiesta una valutazione delle qualità rispetto alle metriche definite in questi standard. → Per progetti più piccoli non si utilizza un intero modello di qualità, ma solo una serie di pratiche in modo da garantire che il progetto software abbia un buon livello qualitativo: sia riutilizzabili e progettato per il cambiamento.

## Principi OO

Nel progettare un sistema OO è necessario identificare correttamente gli oggetti e trovare il giusto mix dei principi base del paradigma OO (incapsulamento, ereditarietà e polimorfismo) per ottenere un sistema software con un alto livello qualitativo.

Principi di base Object Oriented:

- Incapsulamento → i dettagli non sono visibili all'esterno; l'interazione con un altro elemento della soluzione avviene attraverso le interfacce, che devono essere quanto più semplici possibili per lo scopo per sono state create.
- Ereditarietà → comportamenti simili (parti di stato o comportamenti) non è opportuno si ripetano, si possono condividere utilizzando l'ereditarietà.
- Polimorfismo → permette di trattare più tipi correlati come se fossero uno solo (sono trattati in maniera uniforme).

Lo strumento fondamentale per lo sviluppo software è una mente ben educata nei principi di progettazione. I design smells → nella maggior parte dei casi (ma non sempre), sono indicatori di un qualche problema nella progettazione.

Design smells:

- Rigidità → è la tendenza di un software a richiedere un elevato sforzo per essere cambiato, anche in caso di cambiamenti semplici. Un design è rigido se un singolo cambiamento causa una cascata di cambiamenti conseguenti in moduli dipendenti. → Più moduli devono essere cambiati, più il sistema è rigido.
- Fragilità → tendenza di un programma a "rompersi" in diversi punti quando viene effettuata una singola modifica. Spesso i nuovi problemi sono in aree concettualmente sciolte con l'area interessata dalla modifica. Risolvere i nuovi problemi generati porta ancora più problemi.
- Immobilità → un design è immobile quando contiene elementi che potrebbero essere utili in altri sistemi, ma il riuso è problematico, sia dal punto di vista del rischio che dello sforzo per separarli dal sistema originario. → Problema molto comune.
- Viscosità → può essere presente a due livelli:

1. Viscosità dell'ambiente → quando l'ambiente è lento e inefficiente, per cui il tempo per raggiungere gli obiettivi aumenta.
  2. Viscosità del software → tipicamente il codice marcisce, soprattutto quando si continua ad apportare modifiche senza cambiare la struttura del codice. Più sono i cambiamenti che introducono questo problema più il sistema è viscoso e quindi nel tempo degrada sempre più facilmente.
- Needless complexity → quando il design contiene più elementi di quelli che sono attualmente utilizzati, ovvero si crea una soluzione più complicata del necessario. Questo accade spesso quando gli sviluppatori anticipano i potenziali cambiamenti e rendono la struttura del software più complessa, ma in grado di supportarli.
  - Needless repetition → non si programma con il copia e incolla del proprio codice perché può essere disastroso nel lungo periodo a causa di operazioni di code-editing. Quando lo stesso codice appare molte volte in diverse forme, gli sviluppatori si sono persi un'astrazione.
  - Opacità → tendenza di un modulo ad essere difficile da comprendere. Un codice può essere scritto in modo chiaro e comprensibile oppure in modo opaco e contorto. L'opacità deve essere minima sia durante la scrittura del codice, sia durante le successive modifiche.

## **SOLID**

Sistema di principi **SOLID**, acronimo di:

- Single responsibility principle
- Open-closed principle
- Liskow substitution principle
- Interface segregation principle
- Dependency inversion principle

### Single Responsibility Principle

Una classe dovrebbe avere, e solo una, ragione (intesa come un singolo tipo di ragione) per cambiare.

Ogni tipo di responsabilità è un asse di cambiamento. Se una classe ha più di una responsabilità queste diventano accoppiate e la classe è stata sovraccaricata, quindi non è più in grado di rispondere alle singole responsabilità. I cambiamenti ad una responsabilità possono danneggiare o inibire l'abilità della classe di collegarsi alle altre. → Questo porta a un design fragile.

Le responsabilità associate ad una classe devono essere coesive tra loro.

→ Questo principio è facilmente applicabile

### OCPI: OPEN-CLOSED PRINCIPLE

Una classe dovrebbe essere aperta alle estensioni, ma chiusa per le modifiche.

→ Questo principio può essere generalizzato a diversi tipi di entità software: classi, moduli, funzioni ecc..

Il software dovrebbe crescere per aggiunte e tali aggiunte non devono avvenire attraverso la modifica del codice già scritto, perché, se non viene rispettato il principio, un cambiamento può generare a cascata altri cambiamenti che porta a rigidità. → La probabilità di malfunzionamenti cresce.

OCPI consiglia di applicare refactoring al design per evitare questi problemi

→ Il refactoring è una tecnica disciplinata per la ristrutturazione di un sistema software (quindi di una soluzione) in modo che la sua struttura esterna sia modificata senza cambiare il suo comportamento esterno.

Dovrebbe garantire che, in caso di cambiamenti del codice, il comportamento dell'intero sistema non cambi.

→ Il refactoring permette di non abusare di gerarchie di ereditarietà, ma di poterle comunque introdurre in un secondo momento ristrutturando il sistema internamente senza modificarne il comportamento.

**Esempio:** Javascript non permette di fare refactoring ed è viscoso rispetto alle migliori pratiche di progettazione del sistema.

Esistono molti strumenti moderni che permettono di fare refactoring al minimo costo possibile. → Il refactoring comprende aspetti minimi (es. cambiare il nome di una proprietà),

### LSP: Liskow Substitution Principle

La relazione fra le classi in una gerarchia dovrebbe essere sub-typing (deve rispettare il principio di sottotipazione). → Idea di sostituibilità.

Si vuole qualcosa del tipo: se per ogni oggetto O1 di tipo S c'è un oggetto O2 di tipo T in tutti i programmi P definiti in termini di T, allora il comportamento di P non cambia quando O1 è sostituito da O2. → S è un sottotipo di T.

Se un metodo f, che accetta come argomento un riferimento a B, si comporta male quando gli viene passato un riferimento ad un'istanza di D, sottoclasse di B, allora D è fragile in presenza di f.

Il LSP è uno dei primi di facilitatori di OCP: se è rispettato il primo principio allora la gerarchia di ereditarietà funzionerà in maniera corretta e non si rivelerà fragile, per cui il codice che sfrutta tale gerarchia potrà essere esteso più facilmente (viene quindi rispettato anche il secondo principio).

Ci sono dei casi in cui sembra perfettamente ragionevole che una sottoclasse possa essere descritta nei termini di "è un" di una superclasse, invece viola il LIP.

#### **Esempio:**

Immaginiamo di voler creare un programma che rispetti i principi SOLID e che riguardi figure geometriche (cerchi, ellissi e altri poligoni di varia natura). → È necessario quindi il più possibile utilizzare un codice polimorfico (e non molti if) e questo si ottiene attraverso una gerarchia che può essere più o meno complessa.

Classe base: figura geometrica + tante specializzazioni per le varie figure.

Problema quadrato e rettangolo: *Sono tutti e due sottoclassi di figura geometrica o esistono relazioni a più livelli?* Quadrato è in effetti un sottotipo di rettangolo e questo gli permette di ereditare tutti i metodi di rettangolo (quindi favorisce il riuso).

#### *Può essere fragile?*

La classe rettangolo supponiamo abbia diversi metodi: cambia lato lungo, cambia lato corto, calcolo perimetro, calcolo area. Quando specializzo Rettangolo in Quadrato alcuni metodi, come cambia lato lungo o cambia lato corto, non vanno più bene, ma devono essere riscritti: quando si cambia il lato lungo o il lato corto anche l'altro lato deve essere cambiato, perché altrimenti non sarebbe più un quadrato.

*Quale operazione a cui viene passato un quadrato non funziona?*

Set.latocorto(5);

Set.latolungo(6);

Area() → dovrebbe dare come risultato 36 e non 30, perché l'ultima operazione ridefinisce entrambi i lati.

→ Quindi è fragile rispetto a Quadrato.

### ISP: Interface Segregation Principle

Dipendenza → L'elemento A dipende dall'elemento B se un cambiamento di B si trasforma in una modifica che subisce anche A.

**Esempio:** dipendenza d'uso.

Le dipendenze fra le classi dovrebbero dipendere dalla più piccola interfaccia possibile. Detto in altri termini: gli utilizzatori non dovrebbero essere forzati a dipendere da metodi che non usano.

Se questo principio non viene rispettato porta a dipendenze non necessarie e degenera nell'implementazioni delle interfacce, causando inutile complessità e potenziale violazione del principio LSP.

Quando una classe ha un elevato numero di operazioni, ma utilizzatori diversi ne utilizzano sottoinsiemi differenti, ma relativamente piccoli, allora è opportuno estrapolare in altre interfacce solo i comportamenti necessari. → Per ogni categoria di utilizzatori sarebbe opportuno creare un'interfaccia comprendente solo i metodi utilizzati.

### DIP: Dependency Inversion Principle

Le dipendenze dovrebbero essere associate alle astrazioni, ovvero:

- I moduli di alto livello non dovrebbero dipendere da moduli di basso livello, perché entrambi dovrebbero dipendere dalle astrazioni. Gli stati di alto livello comprendono le classi che offrono

funzionalità all'utente, mentre quelli di basso/medio livello comprendono le classi che supportano le funzionalità di alto livello. → Ciò deve avvenire SENZA che si crei una dipendenza diretta. Le dipendenze devono esistere fra elementi dello stesso livello.

Gli elementi di alto livello sono per loro natura stabili, mentre quelli di basso livello sono più volatili. Le astrazioni devono diventare elementi concreti della soluzione, ovvero pezzi di codice.

- Le astrazioni non dovrebbero dipendere dai dettagli, ma sono i dettagli che dovrebbero dalle astrazioni.

Un' architettura object-oriented ben strutturata, dovrebbe avere degli strati chiari e ben definiti, in cui ognuno espone un set di servizi coerenti attraverso un'interfaccia ben definita e controllata.

Un'ingenua applicazione dello stile a livelli può portare a facili violazioni del DIP. → Una soluzione comune consiste nel rendere i livelli più bassi dipendenti da un'interfaccia di servizi dichiarata nei livelli di più alti.

I principi SOLID sono validi sempre, ma è sempre facile convertirli in meccanismi operativi. → Per garantire l'applicazione dei principi si possono utilizzare delle pratiche che dipendono dal modo in cui si realizza il software: i pattern.

## 13. GRASP

### **Progettazione guidata dalle responsabilità (Responsibility Driven Development)**

RDD è un metodo per progettare sistemi software sulla base di responsabilità.

Responsabilità → Un contratto o un obbligo di un classificatore (definizione data da UML).

Nella RDD, gli oggetti software sono considerati come dotati di responsabilità, intendendo per responsabilità un'astrazione di ciò che fanno. Le responsabilità sono correlate agli obblighi, o al comportamento di un oggetto in relazione al suo ruolo.

- ***Responsabilità di fare*** di un oggetto:
  - Fare qualcosa con esso
  - Dare inizio ad un'azione in altri oggetti
  - Controllare e coordinare le attività di altri oggetti
- ***Responsabilità di conoscere*** di un oggetto:
  - Conoscere i propri dati privati encapsulati
  - Conoscere gli oggetti correlati
  - Conoscere cose che può derivare o calcolare

→ Le responsabilità vengono assegnate alle classi di oggi durante la progettazione ad oggetti: quelle più grandi coinvolgono centinaia di classi e metodi, mentre quelle minori possono coinvolgere un solo metodo.

La RDD comprende anche l'idea di collaborazione.

### **GRASP (General Responsibility Assignment Software Patterns)**

GRASP può essere un aiuto per l'apprendimento della progettazione OO e per l'applicazione dei ragionamenti di progettazione in un modo metodico, razionale e spiegabile.

#### Connessione tra responsabilità, GRASP e diagrammi UML

Le decisioni sull'assegnazione delle responsabilità agli oggetti possono essere prese mentre si esegue la codifica, oppure durante la modellazione. → In UML, il momento migliore per considerare le responsabilità è il disegnare i diagrammi di interazione.

→ Quando si disegna un diagramma di interazione di UML, vengono prese delle decisioni riguardo all'assegnazione di responsabilità.

#### Pattern

Pattern → Descrizione di un problema e una soluzione, con un nome e che può essere applicata in nuovi contesti, con consigli su come applicarla in circostanze diverse.

Dare un nome ad un pattern serve perché:

- Permette la nostra segmentazione e assimilazione del concetto
- Facilita la comunicazione.

Nel 1994 venne pubblicato il libro “*Design Patterns*” che descrive 23 pattern per la progettazione OO, creati da quattro autori e per questo tali pattern vengono chiamati Design Patterns della Gang of Four (GoF).

I GRASP sono un insieme di pattern o di principi?

I GRASP definiscono 9 principi di progettazione OO di base (o blocchi di costruzione base), che possono essere anche chiamati pattern.

## **GRASP Patterns**

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

### Creator

**Problema:** Chi è responsabile della creazione di nuove istanze in una classe A?

**Soluzione:** Assegnare alla classe B la responsabilità di creare un'istanza della classe A, se almeno una delle seguenti condizioni è vera:

- B "contiene" o aggrega una composizione oggetti di tipo A
- B registra A
- B utilizza strettamente A
- B possiede i dati per l'inizializzazione di A

→ Si tratta di una responsabilità di fare.

→ B e A fanno riferimento a oggetti software, non ad oggetti del modello di dominio: prima si cerca di applicare Creator cercando degli oggetti software ma, se ancora non è stata definita alcuna classe software, per LRG (Low Representational Gap) va guardato il modello di dominio.

### Information Expert (o Expert)

**Problema:** Qual è il principio di base per assegnare responsabilità agli oggetti, in modo che i sistemi tendano ad essere più semplici da comprendere, mantenere ed estendere, e in modo che le nostre scelte offrano maggiori opportunità di riutilizzare componenti nelle applicazioni future?

**Soluzione:** Assegnare una responsabilità alla classe che possiede le informazioni necessarie per soddisfarla.

→ Una responsabilità necessita di determinate informazioni per essere soddisfatta, come informazioni su altri oggetti, sullo stato di un oggetto, su ciò che circonda l'oggetto, sulle informazioni che si possono ricavare ecc.

### Low Coupling

**Problema:** Come ridurre l'impatto dei cambiamenti, supportare la bassa dipendenza e un maggiore riutilizzo?

**Soluzione:** Assegnare le responsabilità in modo tale che l'accoppiamento rimanga basso. Usa questo principio per valutare le alternative.

→ È un pattern usato per valutare progetti esistenti o la scelta tra nuove alternative: se tutte le cose sono uguali, va preferito il progetto in cui l'accoppiamento è più basso.

Lo scopo è quello di ridurre il tempo, i costi e i difetti nella modifica del software.

**NB:** Information Expert guida verso una scelta che sostiene Low Coupling.

### Controller

**Problema:** Qual è il primo oggetto, oltre lo strato UI, a ricevere e coordinare un'operazione di sistema?

**Soluzione:** Assegnare la responsabilità a un oggetto che rappresenta una delle seguenti scelte:

- Rappresenta il "sistema" complessivo, un "oggetto radice", un dispositivo all'interno del quale viene eseguito il software o un sottosistema principale.
- Rappresenta uno scenario di un caso d'uso all'interno del quale si verifica l'operazione di sistema, in genere chiamata: <UseCaseName> Handler, <UseCaseName> Coordinator, o <UseCaseName> Session.

**NB:** Utilizzare la stessa classe di controller per tutti gli eventi di sistema nello stesso scenario di utilizzo.

**Sessione** → Informalmente, è un'istanza di una conversazione con un attore. Le sessioni possono essere di qualsiasi lunghezza ma sono spesso organizzate in termini di casi d'uso (utilizzare le sessioni del caso).

→ In base al principio di separazione Modello-Vista, si sa che gli oggetti della UI non devono contenere logica applicativa o di "business", come per esempio calcolare una mossa di un giocatore. Quindi, una volta che gli oggetti della UI rilevano l'evento del mouse, devono delegare la richiesta agli oggetti di dominio nello strato del dominio ( $\neq$  UI). → *Qual è il primo oggetto a ricevere il messaggio dallo strato UI?*

### High Cohesion

**Problema:** Come mantenere gli oggetti focalizzati, comprensibili e gestibili, e come effetto sostenere Low Coupling?

**Soluzione:** Assegnare le responsabilità in modo tale che la coesione rimanga alta. Usa questo principio per valutare le alternative.

Coesione → Misura quanto siano correlate le operazioni di un elemento software da un punto di vista funzionale, e anche quanto lavoro sta eseguendo un elemento software.

Una classe con bassa coesione fa molte cose non correlate o fa troppo lavoro. → Tali classi sono indesiderabili, perché soffrono dei seguenti problemi:

- Difficile da comprendere
- Difficile da riutilizzare
- Difficile da mantenere
- Delicato e costantemente influenzati dal cambiamento

Le classi con bassa coesione hanno assunto responsabilità che avrebbero dovuto essere delegate ad altri oggetti.

Coesione funzionale alta → Quando gli elementi di un componente lavorano tutti insieme per fornire un comportamento ben circoscritto.

**NB:** Questo è un principio di valutazione che un progettista deve sempre applicare quando deve valutare tutte le decisioni di progettazione.

### Polymorphism

**Problema:** Come gestire le alternative basate sul tipo? Come creare componenti software inseribili?  
→ Cioè: La variazione delle condizioni utilizzando le istruzioni del flusso di controllo produce codice che è difficile da estendere.

**Soluzione:** Quando le alternative o i comportamenti correlati variano con il tipo (classe), assegnare le responsabilità del comportamento ai tipi per i quali il comportamento varia, utilizzando operazioni polimorfiche.

→ Un progetto basato sull'assegnazione di responsabilità in base a Polymorphism può essere facilmente esteso per gestire nuove variazioni.

→ Prima di investire sull'elasticità di questo pattern valutare la vera probabilità che si verifichino tali variazioni.

**NB:** Non testare il tipo di un oggetto, o usare la logica condizionale per eseguire le alternative che variano in base al tipo.

### Pure Fabrication

**Problema:** Quale oggetto deve avere la responsabilità quando non si vogliono violare High Cohesion e Low Coupling, o altri obiettivi, ma le soluzioni offerte da Expert (per esempio) non sono appropriate?

**Soluzione:** Assegnare un insieme di responsabilità altamente coeso a una classe artificiale o di convenienza che non rappresenta un concetto del dominio del problema, ma è qualcosa di inventato per sostenere High Cohesion, Low Coupling e il riuso.

→ High Cohesion è sostenuta, poiché le responsabilità sono divise in una classe a grana fine, incentrata solo su un insieme molto specifico di compiti correlati. Il potenziale riuso può aumentare grazie alla presenza di classi Pure Fabrication, le cui responsabilità possono essere usate in altre applicazioni.

**NB:** Se usata in modo eccessivo, questo pattern può portare ad un eccesso di oggetti comportamentali che hanno responsabilità che non sono collocate insieme alle informazioni richieste per la loro soddisfazione.

### Indirection

**Problema:** Dove assegnare una responsabilità, per evitare l'accoppiamento diretto tra due (o più) elementi? Come disaccoppiare degli oggetti in modo da sostenere il Low Coupling e mantenere alto il potenziale riuso?

**Soluzione:** Assegnare le responsabilità ad un oggetto intermedio, per mediare tra componenti o servizi in modo che non ci sia accoppiamento diretto tra essi.

### Protected Variations

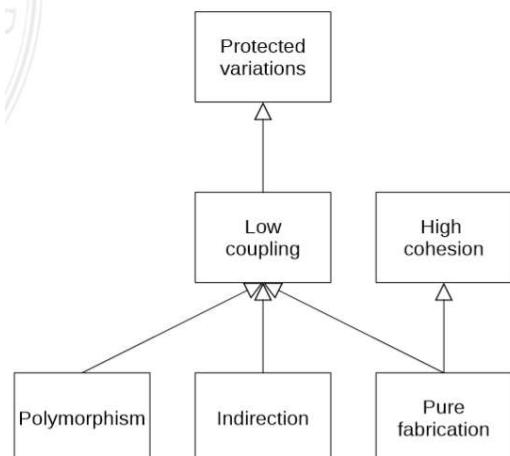
**Problema:** Come progettare oggetti, sistemi e sottosistemi in modo tale che le variazioni o l'instabilità in questi elementi non abbiano effetti indesiderati su altri elementi?

**Soluzione:** Individuare i punti in cui sono previste variazione o instabilità, assegnare delle responsabilità in modo da creare delle interfacce stabili intorno a questi punti.

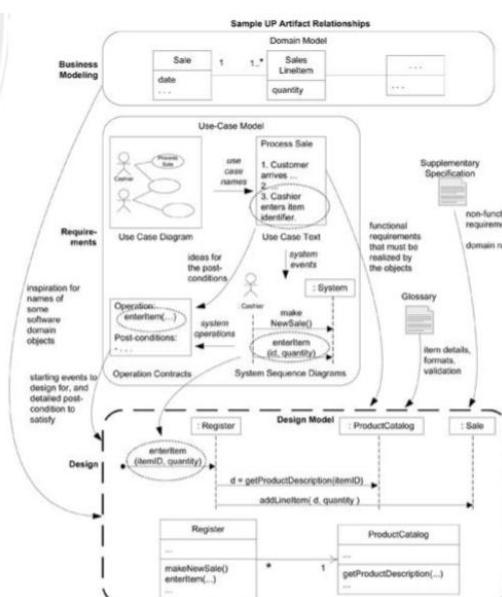
Interfaccia → Usato nel senso più ampio di vista per l'accesso, non significa solo letteralmente qualcosa come l'interfaccia di Java.

→ Quasi tutti gli artefatti di progettazione software sono una specializzazione di Protected Variations.

Low Coupling formalizza il principio di protezione dalle variazioni nelle diverse implementazioni di un'interfaccia o estensioni di sottoclassi di una superclasse.



### Relazione fra gli artefatti



## 14. DESIGN PATTERNS PART 1

Ogni **pattern** descrive un problema ricorrente più volte nell'ambiente, descrive inoltre la soluzione ad esso in modo tale da poterla usare tante altre volte. I pattern forniscono, dunque, soluzioni funzionanti a tutti i problemi conosciuti che avvengono durante il design. Forniscono una descrizione nominale del problema, una soluzione, dicono quando e come applicarla a quel determinato contesto. La soluzione è riutilizzabile per tutti i problemi comuni che incorrono nel contesto del design del software.

I pattern sono solitamente raggruppati in una struttura coerente con il suo proprio vocabolario sintattico e grammaticale, sono quindi raggruppati in linguaggi cioè cataloghi di elementi correlati. Esistono vari cataloghi di patterns nell'ingegneria del software. Ci contreremo nei design patterns i quali hanno vari cataloghi.

### Documentazione

- Classificazione e nome del pattern: un nome unico e descrittivo che aiuti l'identificazione ed il riferimento al pattern.
- Intento: una descrizione dell'obiettivo del pattern e le ragioni per le quali usarlo.
- Conosciuto anche come: altri nomi per il pattern.
- Motivazioni: uno scenario che consiste in un problema e ad un contesto nel quali questo pattern può essere usato.
- Applicabilità: situazioni nelle quali questo pattern è utilizzabile; il contesto per il pattern.
- Struttura: una rappresentazione grafica del pattern. Il diagramma delle classi e il diagramma delle interazioni possono essere usati per questo scopo.
- Partecipanti: una lista delle classi e degli oggetti usati nel pattern e nei suoi ruoli nel design.
- Collaborazioni: una descrizione di come le classi e gli oggetti usati nel pattern interagiscono tra di loro.
- Conseguenze: una descrizione dei risultati, effetti collaterali e trade off causati dall'utilizzo del pattern.
- Sample code: una illustrazione di come il pattern può essere usato in un linguaggio di programmazione.
- Usi conosciuti: esempi di usi reali del pattern.
- Pattern correlati: altri pattern che hanno qualche relazione con quel pattern; discussione delle differenze tra il pattern e i suoi simili.

### Obiettivi dei pattern della Gang of Four

- Creazionale → astrazione del processo di istanziazione.
- Strutturale → concentrati su come le classi e gli oggetti sono composti per formare strutture più grandi.
- Comportamentale → concentrati su algoritmi e sull'assegnamento delle responsabilità tra gli oggetti.

### I 23 patterns della Gang of Four

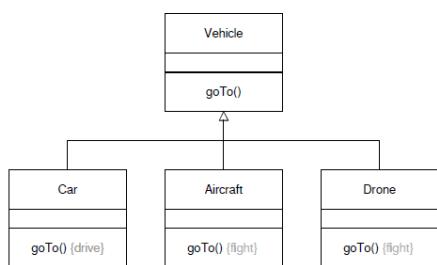
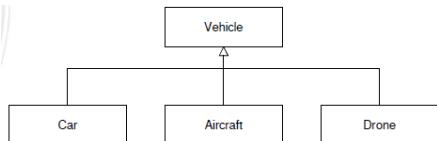
- Creational → Abstract Factory, Builder, Factory Method, Prototype, Singleton.
- Structural → Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
- Behavioral → Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor.

		Purpose		
Scope	Class	Creational	Structural	Behavioral
		Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

## Composizione dell'ereditarietà

Le 2 più comuni tecniche per il riutilizzo funzionale nei sistemi object-oriented sono le classi di ereditarietà e la composizione ad oggetti. Favorire la object composition rispetto all'ereditarietà delle classi aiuta a mantenere ogni classe incapsulata e concentrata su un solo compito. La delega è un modo di rendere la composizione efficace per il riuso tanto quanto l'ereditarietà.

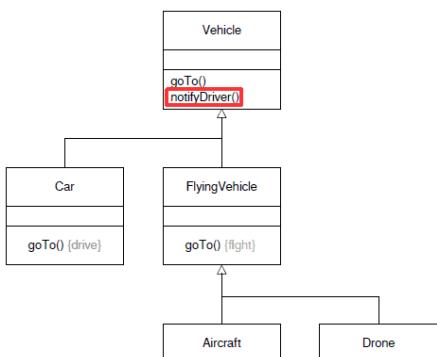
Cosa c'è di sbagliato nell'ereditarietà?



**goTo()** in Aircraft ed in Drone si comportano nella stessa maniera. Secondo i design smells ci sono delle ripetizioni che non sono necessarie. Copiare ed incollare può essere una operazione text-editing utile ma può allo stesso tempo essere una code-editing operazione disastrosa.

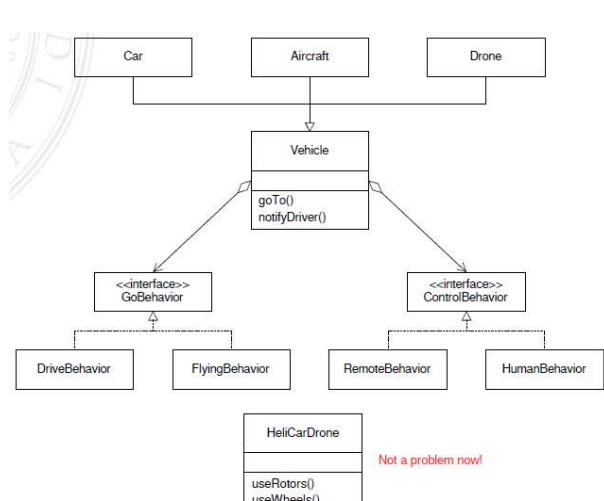
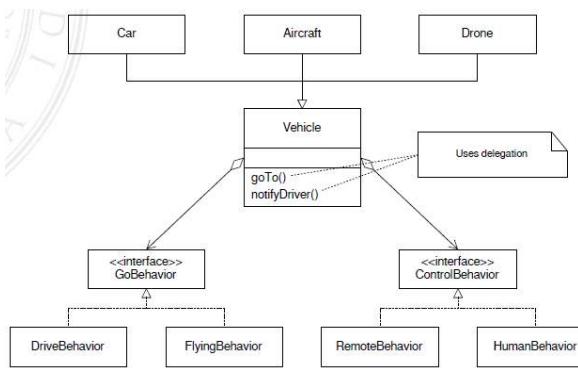
Quando lo stesso codice appare più e più volte in varie forme, gli sviluppatori stanno mancando un'astrazione.

Cos'è sbagliato nel nostro uso di ereditarietà? Non stiamo indirizzando OCP e PV correttamente.

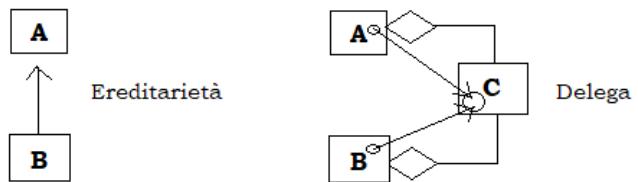


### Soluzioni:

- Le superclassi non sono le sole possibili astrazioni.
- Usare le composizioni: sono più flessibili e possono essere cambiate a run-time.

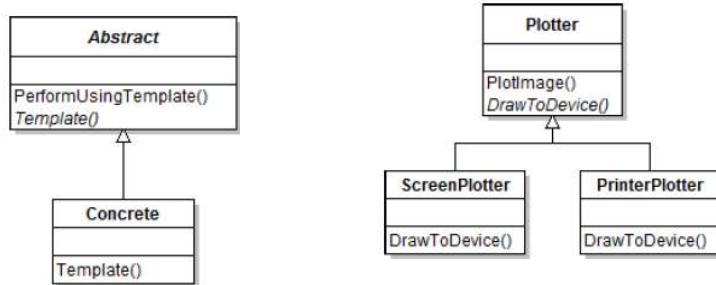


Invece che far ereditare B da A, il testo della Gang of Four spiega il **meccanismo di delega**: A e B chiamano un metodo riferito a C, così che delegano a C l'esecuzione della funzionalità. La delega è un meccanismo esplicito: dentro al codice lo scrivo, mentre nell'ereditarietà è automatico. Con la delega posso effettuare modifiche durante l'esecuzione, mentre con l'ereditarietà non è possibile. L'ereditarietà può, inoltre, dare più problemi.



### Gang of Four: Template Method

È un pattern comportamentale per facilitare degli aspetti algoritmici, usa l'ereditarietà. Il problema è che spesso posso avere algoritmi di cui conosco il comportamento ma alcuni aspetto non li conosco e dipenderanno dalle gerarchie di ereditarietà. Esso definisce lo scheletro di un algoritmo in un'operazione, rimandando certi step alle sottoclassi. Questo metodo lascia ridefinire alle sottoclassi certi step di un algoritmo senza cambiare la sua struttura.



### Template Method nella libreria standard di Java

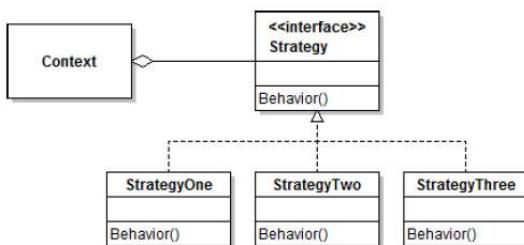
```

Java.util.AbstractList<E> {
    int indexOf(Object o)
    abstract E get(int index)
    ...
}
  
```

Questo metodo muove, dunque, comportamenti comuni in alto nell'albero dell'ereditarietà anche quando parzialmente specificati.

### Gang of Four: Strategy

Questo pattern definisce una famiglia di algoritmi, incapsulando ognuno di loro e rendendoli intercambiabili. Strategy lascia variare gli algoritmi indipendentemente dai clients che li usano. Esso aiuta ad implementare OCP. Favorisce, inoltre, la composizione (che è dinamica) rispetto all'ereditarietà (che è statica); quindi il “ha un” prevale sopra al “è un”.



## 15. DESIGN PATTERNS PART 2

Allocata memoria ogni volta che il client ha bisogno di una referenza. → Di conseguenza non vi è controllo sul ciclo di vita delle istanze.

Inevitabilmente, quando è direttamente l'utilizzatore a creare le istanze delle classi concrete che si vogliono utilizzare, è molto probabile che si violi il DIP.

Inoltre, cambiamenti nel costruttore delle classi concrete rompono i clients, violando l'OCP.

**Esempio:** In Java la classe `ArrayList` che implementa l'interfaccia `List`.

`ArrayList mylist = ...` → Questo crea una dipendenza, perché se in un secondo momento si fa `mylist.ad()`; si crea una dipendenza d'uso: la classe di alto livello crea una dipendenza verso la classe di basso livello.

Per non violare il DIP quando si vuole creare una lista deve essere scritto:

`List mylist = newArrayList();` → Adesso non è più direttamente dipendente da `ArrayList`, quindi ho spostato la dipendenza all'interfaccia.

Nella classe `ArrayList` potrei voler cambiare qualcosa, quindi volatile, mentre l'interfaccia `List` è molto più stabile. → La classe di basso livello lavora nei termini dell'interfaccia, quindi si abbassa la probabilità che la modifica si propaghi.

Con questa seconda scrittura, se qualcuno cambia un metodo della classe `ArrayList` non si ha nessun effetto perché `mylist` fa riferimento ai metodi di `List`.

**Problema:** Nel momento in cui la classe di alto livello crea la classe di basso livello di cui ha bisogno si crea comunque una dipendenza (**Esempio:** Vengono cambiati i parametri del costruttore di `ArrayList`).

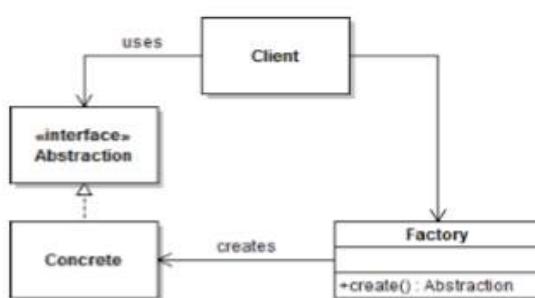
→ Non sono completamente slegati dalla classe `ArrayList`.

→ L'instanziazione diretta rischia di violare il DIP.

**Soluzione:** La soluzione è fare in modo che non sia direttamente il client a generare le istanze: si utilizza un intermediario che, essendo coinvolto nella creazione degli oggetti, può essere utilizzato per tenere traccia delle istanze che vengono create e del loro ciclo di vita, oppure un intermediario che porrebbe verificare la necessità di creare nuovi oggetti piuttosto di riciclarne altri esistenti.

Questo intermediario è detto factory. → Serve per disaccoppiare l'utilizzatore di un oggetto dal creatore dell'oggetto e permette di referenziare l'oggetto creato attraverso una common interface.

Quindi quando client ha bisogno di un'istanza di un'abstraction la chiede alla factory.



**Esempio:**

Per una lista `List mylist = FactoryList.create();`

`FactoryList` è l'intermediario che consente di contenere una reference ad un oggetto di tipo `List`, ma poiché `List` è un'interfaccia l'oggetto non sarà di tipo `List`, ma un'istanza della classe concreta che implementa l'interfaccia.

La factory anche se non crea effettivamente un'istanza, ne restituisce una su cui lavorare che non è più utilizzata.

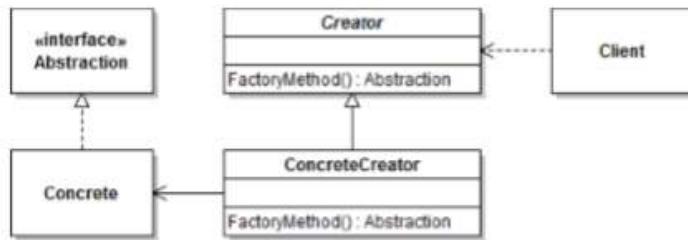
Soluzione che aggiunge un elemento che non è collegato al dominio, quindi si sta aumentando il livello di complessità della soluzione, ma dall'altro lato è rispettato il DIP e quindi sarà più difficile che nel software una modifica si propaghi, ovvero effettuare una modifica avrà un costo più basso.

## GoF: Factory method

Nel testo della gang of 4 vengono presentate due factory che partono da questa idea di base aggiungendo complessità.

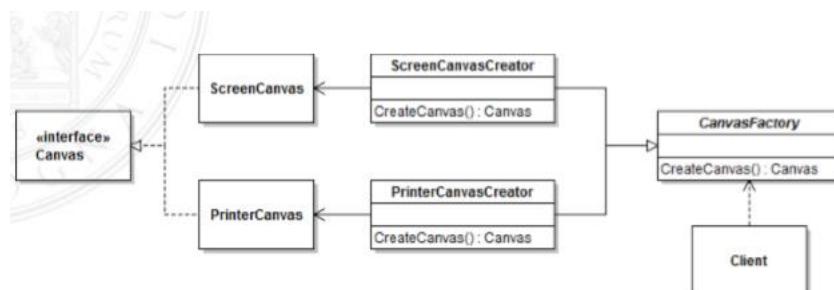
La factory introduce un ulteriore livello di invenzione.

Si definisce un'interfaccia per creare un oggetto, ma si lascia alla sottoclasse decidere quale classe istanziare. Anche la factory è suddivisa in un livello di astrazione e in un livello concreto (differenza rispetto alla soluzione precedente). → A partire dalla class di base astratta Creator ci possono essere più ConcreteCreator. Questo perché si potrebbero volere classi concrete di tipo diverso che utilizzano internamente logiche diverse per determinare quale classe concreta utilizzare.



**Esempio:** Si vuole un client che voglia disegnare oggetti su un canvas.

Per ottenere un Canvas (astrazione) si deve utilizzare un CanvasFactory, che in base alla logica seconda la quale viene implementata può essere di due tipi: ScreenCanvasCreator (disegno a video) e PrinterCanvasCreator (disegno stampato).



L'idioma factory nella pratica è più molto usato e quindi è molto facile trovare implementazioni di factory come idioma piuttosto che nella versione di factory method.

In alcuni casi la forma è ancora più semplificata di quella dell'idioma: viene fatto collassare l'astrazione e il creatore inserendo FactoryMethod dentro l'astrazione (che diventa la classe astratta).

**Esempio:**

Java API in cui sono state fatte collassare l'astrazione e la factor:

- `java.util.Calendar#getInstance()` → Calendar ha sia il ruolo di astrazione (sulla base della quale sono poi definiti diversi tipi di calendari completi) sia di factory mettendo a disposizione un factoryMethod attraverso il quale vengono create le istanze.  
Mettendo a disposizione un solo metodo concreto non può essere un'interfaccia, ma deve essere una classe astratta. I parametri possono servire alla logica interna della factory per decidere quale logica interna utilizzare.
- `java.text.NumberFormat#getInstance()`
- `java.nio.charset.Charset#forName()`

**Esempio:** Quando si ha una lista e interessa solamente manipolare degli elementi non interessa il tipo di lista ritornato dalla Factory (ArrayList, LinkedList ecc...). → Dal punto di vista funzionale le funzionalità sono le stesse, ma dal punto di vista computazionale no: per certe operazioni è più efficiente un ArrayList, mentre per altre una LinkedList.

Questo si può risolvere passando dei parametri al metodo create per influenzare il comportamento della factory. → In questo modo non si lega il client che utilizza la lista alla classe concreta, ma il client è in grado di indicare quale classe di tipo lista gli farebbe più comodo.

Non è l'unico tipo di factory presente nel libro della GoF; l'altro parla di famiglie di factory per generare famiglie di oggetti. → L'uso non è molto frequente.

### **The notification problem**

I pattern risolvono problemi ricorrenti in progettazione: sia relativi al mantenimento di un certo livello qualitativo interno del software, ma anche relativi ad aspetti funzionali.

Factory permette di istanziare gli oggetti senza perdita di qualità.

### **Problema di notifica**

La qualità non c'entra, ma la soluzione proposta dal pattern sarà di buona qualità.

All'interno della soluzione vi sono diversi elementi che sono interessati ai cambiamenti di stato di un terzo elemento.

Possibili soluzioni:

- **Polling:** i vari elementi periodicamente vanno a controllare se lo stato dell'oggetto è cambiato.  
→ Meccanismo non molto ottimale: parte delle risorse del sistema sono indirizzate verso un'attività che non ha utilità pratica, ma è necessaria.
- L'elemento interessato, ogni volta che cambia stato, informa gli altri elementi.  
→ Questo meccanismo suppone che l'oggetto conosca a priori tutti quelli che sono interessati al suo cambiamento di stato. Vengono create molte dipendenze e non è logico, perché sono gli altri elementi a dover dipendere dall'oggetto che cambia stato.

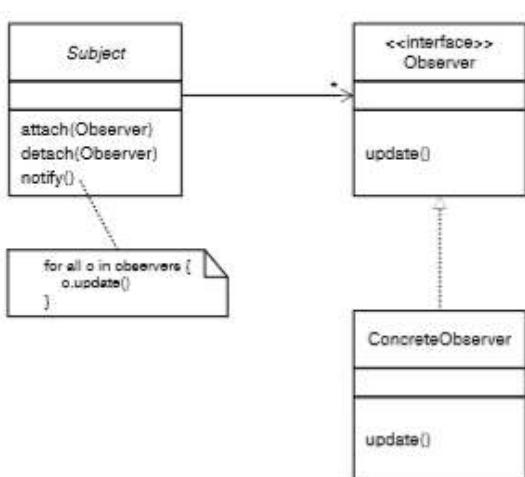
Si vuole che gli osservatori non siano conosciuti a priori dal soggetto e questi oggetti possono anche cambiare nel tempo (aspetto funzionale). Quando una classe notifica un'altra, è esposta alla sua interfaccia.

Si vuole che la soluzione rispettati:

- **ISP:** la dipendenza di una classe all'altra dovrebbe dipendere dall'interfaccia più piccola possibile (solo all'unico metodo con cui l'oggetto notifica che il suo stato è cambiato).
- **DIP:** dipendenze verso le astrazioni e non verso le classi concrete.
- **PV**

### **GoF: Observer**

Definisce una dipendenza uno a molti, così quando un oggetto cambia stato, tutti gli oggetti che dipendono da esso sono identificati e aggiornati automaticamente.



C'è un Subject (oggetto a cui si è interessati) e dei ConcreteObserver (oggetti concreti).

→ L'interazione fra queste due strutture è mediata dall'interfaccia Observer: ogni elemento interessato a essere notificato sul cambiamento dello stato di Subject, dovrà implementare l'interfaccia, quindi il metodo update().

Nel soggetto ci sono i metodi di registrazione e de-registrazione, ovvero attach() e detach() in cui come parametro viene passato l'Observer. → Il Subject mantiene una lista degli osservatori registrati.

Il metodo notify() consiste in un ciclo che scorre tutti gli osservatori che sono registrati e per ognuno richiama il metodo update().

### Vantaggi:

- È possibile aggiungere in modo dinamico osservatori
- Il soggetto non deve conoscere in anticipo il tipo degli osservatori
- La dipendenza tra il soggetto e l'interfaccia observer (è una sola!!) e viene rispettato DIP
- È rispettato PV

Questa soluzione è pensata per essere di buona qualità.

### **Esempio: Observer in Java**

```
java.util.Observer  
    public interface Observer {  
        void update(Observable o, Object arg)  
    }  
  
java.util.Observable  
    public class Observable {  
        public void addobserver(Observer o);  
        public void deleteobserver(Observer o);  
        public void notifyobservers();  
        protected void setchanged();  
        ...  
    }
```

### **GoF: Façade (pattern strutturale)**

Quando si progetta una soluzione di una certa complessità questa si scomponete in funzionalità che sono risolte da elementi diversi che costituiscono sottosistemi.

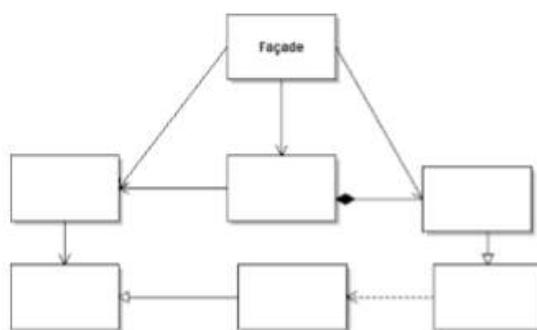
→ In questi casi è utile definire un'interfaccia comune: Façade (è un elemento esclusivamente della soluzione).

Façade perché è una facciata, è una classe concreta e ha un certo numero di metodi. In realtà ogni volta che si richiama una funzionalità su oggetti di questa classe rimanda a funzionalità di qualcun altro. → Non c'è profondità nell'oggetto, perché non realizza internamente nessun tipo di operazione.

Il Façade si pone a schermo tra l'utilizzatore esterno e le classi all'interno della soluzione, provvede un'interfaccia unificata di una serie di interfacce in un sottosistema.

La facciata riceve le richieste del client esterno e le delega all'oggetto corretto e definisce un'interfaccia di alto livello che rende il sottosistema più facile da utilizzare.

Se non si utilizza Façade la soluzione funziona uguale, perchè non riguarda un problema funzionale, ma viene introdotta solo per miglioria interna della soluzione.



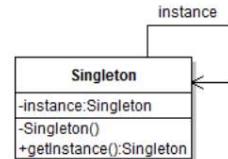
*Quali sono i problemi della soluzione senza Façade (e quindi le migliorie introdotte da Façade )?*

- Potrebbe non essere rispettato il DIP
- Se si fa dipendere direttamente le classi dai singoli elementi della soluzione si una dipendenza da tanti elementi, quindi ogni elemento che viene modificato può comportare modifiche anche in altri punti. Se si utilizza Façade questo aspetto viene mitigato.

## 16. DESIGN PATTERNS PART 3

### Pattern Singleton

Assicura che una classe abbia una sola istanza e fornisca un punto di accesso globale e singolo ad essa. In java si implementa con un metodo statico della classe singleton, questo metodo deve ritornare l'unico metodo esistente della classe stessa. Il costruttore deve diventare privato e questo impedisce al codice di altre classi di creare istanze a causa di questo costruttore privato. Quando devo accedere all'unica istanza della classe singleton uso il metodo statico `getInstance()` che è un metodo della classe che restituisce il singleton (precedentemente definito).



Con questo approccio, uno sviluppatore ha una visibilità globale nei confronti di questa singola istanza, attraverso il metodo statico `getInstance()` della classe. Poiché la visibilità verso le classi pubbliche ha un campo d'azione globale, in qualsiasi punto del codice, in qualsiasi metodo è possibile scrivere `SingletonClass.getInstance()` per ottenere la visibilità verso l'istanza singleton. Bisogna scrivere questa operazione in maniera opportuna, può essere fatta in varie maniere: appena parte il programma può venire creata quell'unica istanza oppure l'istanza non viene creata fino alla prima volta in cui viene richiesta. Il singleton è un pattern problematico perché è spesso usato in modo improprio, infatti i membri della Gang of Four se dovessero riscrivere il libro lo rimuoverebbero perché spesso gli usi sono errati.

L'uso del singleton è ritenuto un code smell (non necessariamente dietro uno smell c'è un errore ma è un'indicazione in un potenziale problema). Da un lato esso deve garantire che da una certa classe venga fornita un'unica istanza, dall'altro lato deve fornire un meccanismo di accesso globale a questo oggetto creato. Spesso però è utilizzato per la seconda ragione e non per la prima (la seconda sarebbe la maniera, non lo scopo per cui dovrebbe venire creato).

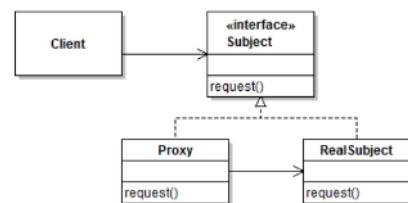
I candidati per creare singleton sono:

- Classi factories: classi a cui mi rivolgo per avere delle istanze. Queste classi non hanno uno stato proprio. Se voglio factories diverse che condividono la stessa object pool sono ottimi candidati per fare dei singleton.
- Classi loggers: classi che servono a fare logging. Voglio ad es. fare in modo che durante l'esecuzione il codice possa memorizzare delle informazioni in un file unico. Non vale la pena avere 100 istanze diverse con lo stesso stato quindi io scrivo un logger unico per risparmiare tante istanze di logger diverse che puntano allo stesso file, evito anche di avere 100 file descriptor aperti che puntano allo stesso file.
- Classi di configurazione: si usa una classe per leggere un file di configurazione e si interroga questa classe per capire quali operazioni sono abilitate e quali disabilitate. Uso il singleton così uso sempre la stessa classe senza stare a rileggere i dati.
- Classi di che non hanno attributi non statici: astrazioni che si basano su quell'unico file di configurazione, anche se ci costruisco sopra 5 astrazioni il file è sempre uno. È ragionevole rappresentarli come singleton perché in queste classi se creo più istanze diverse sarebbero uguali perché sono sempre di una stessa risorsa/file.

Il fatto che l'istanza sia singola non è intrinseco nel dominio del problema.

### Pattern Proxy

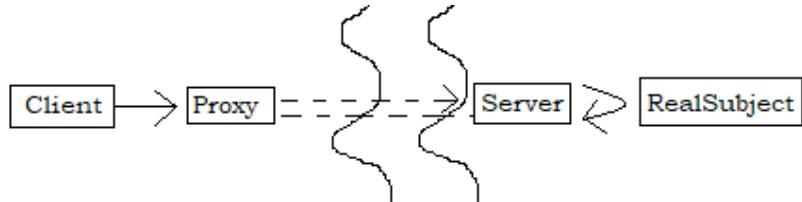
Fornisce un surrogato ad un oggetto per poterne controllare l'accesso. L'idea è quella di mediare l'accesso ad un oggetto (RealSubject) da parte di un client, in maniera che sia trasparente al client. Il client deve essere convinto di parlare direttamente sull'oggetto sul quale vuole richiamare certe operazioni, ma in realtà questa iterazione avviene per mezzo di un intermediario detto Proxy. Esso deve esporre verso l'esterno la stessa identica interfaccia di RealSubject.



Quando client vuole parlare con RealSubject, qualcuno deve dargli l'istanza che gli serve. Invece di passargli direttamente RealSubject gli passo il proxy. Il client ragiona in termini di interfaccia quindi lui sa che gli

viene passato un oggetto di tipo subject, quindi è come se si aspettasse una lista ma posso passargli una qualunque classe che implementi la lista.

→ Invece di passargli l'oggetto che direttamente implementa le funzionalità gli passo il proxy. Esso ha l'impressione di parlare esattamente con quell'oggetto, invece dialoga con un proxy che rimanda la richiesta a RealSubject e ritornare al client, a volte però effettua ad es. controlli di accesso e cose di questo tipo. Quindi si mette un intermediario trasparente rispetto al client.



Accessi remoti: il proxy permette di costruire programmi distribuiti (su più nodi della rete) facendo in modo che la presenza degli oggetti remoti diventi trasparente agli utilizzatori.

#### Esempio: Uso del Proxy.

- Access control;
- Access counter;
- Access logger;
- Accesso ad oggetti remoti;
- Smart references.

#### Esempio: Di più dimensioni – “caso della pizza”

Si possono avere 2 dimensioni di variabilità ad esempio il tipo di pizza (alici, margherita o marinara) e la presenza o meno degli optional sulla pizza.

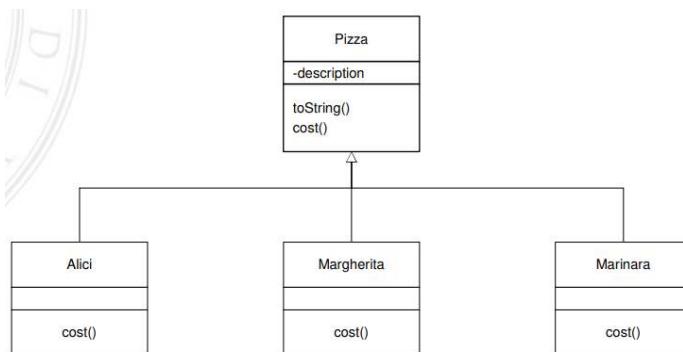
→ È problematica dal punto di vista qualitativo, devo vedere se viola un principio di SOLID.

Se devo aggiungere nuove opzioni devo modificare la struttura di Pizza violando l'open-closed principle.

Classe cost ():

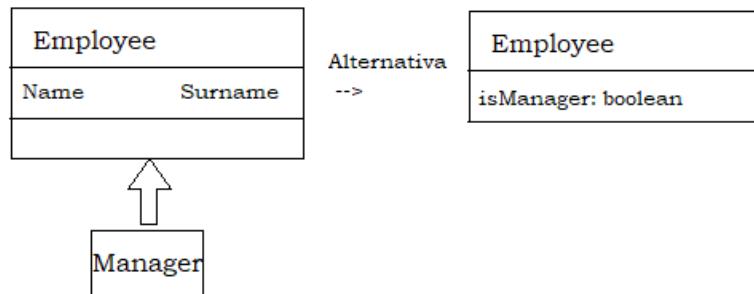
```

cost = 0;
if (this.bufala)
    cost += 1;
if (this.wholegrain)
    cost += 1.5;
  
```



Polimorphic e open-closed principle sono collegati e possono essere entrambi violati. Questi if si referenziano a variazioni di comportamento legati alla natura degli oggetti su cui opero, questo è legato alla definizione di polymorphic strettamente legato ad open-closed.

Esempio:



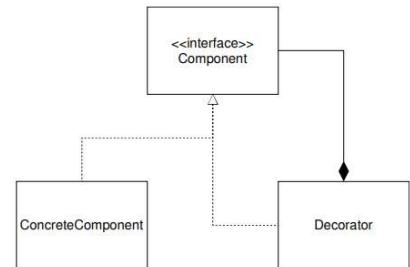
### **Pattern decorator**

Serve per aggiungere dinamicamente un comportamento ad un oggetto che corrisponde ad una responsabilità.

C'è un'interfaccia che viene realizzata sia da un concrete component sia da un decoratore.

Ogni decoratore che aggiunge può realizzare una responsabilità diversa.

Uso l'esempio di prima della pizza senza violare l'open-closed principle.



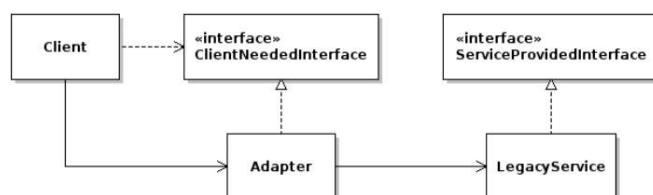
I decoratori sono oggetti con la stessa interfaccia del metodo (in questo caso di Pizza). Quindi non parlo con l'oggetto Pizza ma con un oggetto più esterno ad es. può essere Bufala. È un pattern comportamentale.

### **Pattern adapter**

È un pattern strutturale, ci serve per adattare gli elementi all'interno di una soluzione. Converte, quindi l'interfaccia di una classe in un'altra interfaccia che il cliente si aspetta.

L'adattatore è un interprete, il client parla il linguaggio dell'interfaccia di cui ha bisogno. → Per rendere i linguaggi compatibili perché magari client ed oggetto parlano lingue diverse ci metto in mezzo un adapter.

Adapter fa lavorare insieme classi che non potrebbero per colpa di interfacce incompatibili, il problema consiste nel gestire interfacce incompatibili e la soluzione è la conversione dell'interfaccia originale di un componente in un'altra interfaccia, attraverso un oggetto adattatore intermedio.



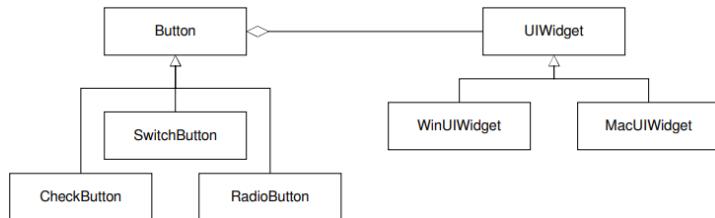
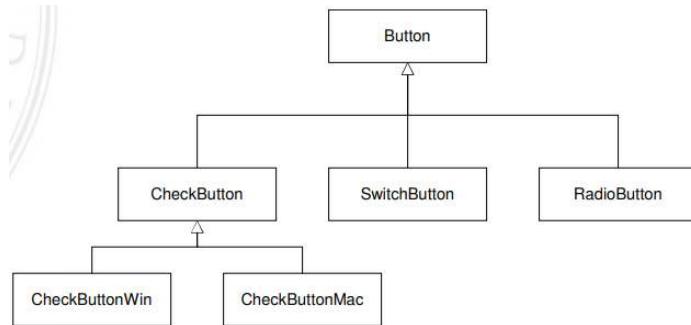
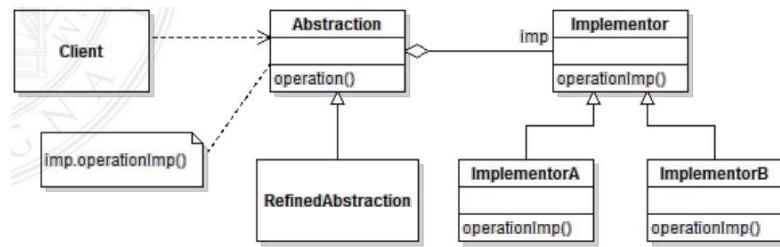
### **Pattern Bridge**

La struttura è simile all'adapter, il problema da affrontare però è quello di permettere che tra un'astrazione (intesa come gerarchia di classi) possa evolvere indipendentemente da un altro insieme di classi che invece svolge delle funzionalità.

L'idea è avere 2 gerarchie di ereditarietà diverse, una concreta e una basata sulle astrazioni, è stato chiamato bridge perché l'idea è avere un ponte, da un lato una parte concreta e dall'altro una parte astratta.

Il client parla con l'astrazione, l'astrazione delega la funzionalità concreta alla parte concreta attraverso la delega (il bridge); le due gambe del ponte, dunque, sono l'abstraction e l'implementor.

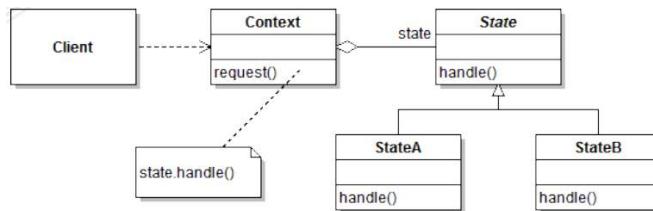
Il client dunque non parla direttamente con la classe di alto livello ma parla con il raffer che delega all'abstraction la realizzazione della funzionalità vera e propria.



## 17. DESIGN PATTERNS PART 4

### Pattern State

Pattern State → Consente ad un oggetto di alterare il suo comportamento quando il suo stato interno cambia. L'oggetto apparirà per cambiare la sua classe.



### Pattern Mediator

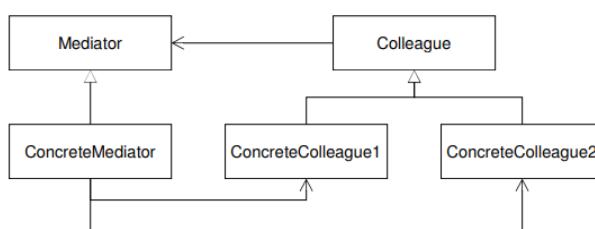
Pattern Mediator → Definisce un oggetto che incapsula come un gruppo di oggetti interagisce.

Se ho un gruppo di oggetti nella soluzione (non per forza correlati) che dialogano tra di loro, posso metterli in mediator facendogli implementare un'unica interfaccia per implementare le funzionalità dall'esterno costruendo una soluzione come quella mostrata nello schema.

Ognuno di questi colleghi può dialogare con un altro collega solo con metodi comuni a tutti quanti. Tutti i metodi che possono servire dai vari colleghi possono essere esposti da questa interfaccia. Il vantaggio è minimizzare il numero di dipendenze e aver spostato tutto in un unico oggetto che fa da mediatore della situazione, ha dunque meno problemi.

Gli oggetti che usiamo devono essere figli di una classe padre o usino una stessa interfaccia.

Mediator al suo interno ha una collezione che vuole poter richiamare, non vuole quindi avere tanti tipi di references diversi ma vuole referenziarli in maniera informe (devono quindi essere figli di una unica classe o che implementino tutti una stessa interfaccia anche se spesso è difficile applicare questo pattern perché la situazione non sempre si verifica).



### Pattern Composite

Pattern Composite → Serve per creare delle strutture che sono comuni in soluzioni legate a strutture ricorsive.

L'idea è quella di raccogliere una collezione di oggetti sotto forma di struttura ad albero che rappresentano delle gerarchie, dentro questa collezione andiamo a definire ogni elemento come un Component. Questo oggetto Component può essere un oggetto foglia o un oggetto composite.

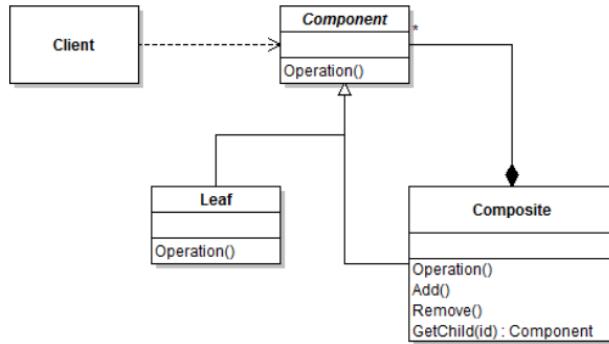
Voglio fare in modo che ogni oggetto di questa struttura appaia all'utilizzatore in modo uniforme, devono apparire come dei Component ma poi in realtà questi oggetti hanno natura diversa: quelli in fondo non referenziano altri oggetti della struttura mentre quelli più su sono collegati ad altri oggetti.

Quelli in fondo (in basso) saranno quindi le foglie, quelli in mezzo saranno i nostri Composite perché compongono altri oggetti. Tutti questi elementi presentano delle operazioni comuni, in più i Composite intermedi hanno operazioni ulteriori che sono tipicamente quelle per gestire la struttura cioè per aggiungere e togliere altri elementi.

Il vantaggio è che il client può parlare indipendentemente con qualunque oggetto della struttura in modo uniforme. Il problema è, quindi, capire come trattare un gruppo o una struttura composta di oggetti nello stesso modo (polimorficamente) di un oggetto non composto (atomico).

→ La soluzione è definire classi per gli oggetti compositi e atomici in modo che implementino la stessa interfaccia.

Si compongono gli oggetti in strutture ad albero per rappresentare le gerarchie:



### Pattern Memento

Pattern memento → È un pattern che ha a che fare con la gestione dello stato.

Se un oggetto vuole essere salvato e poi ripristinato deve presentare un metodo che serve per offrire una rappresentazione in termini dello stato dell'oggetto che sia opaca, ad esempio un metodo che ritorna una stringa (**Esempio**: ritorna lo stato e lui ritorna una stringa).

Questo oggetto mi espone un metodo per conoscere lo stato per poterlo salvare e uno per poterlo ripristinare ed espone dati opachi, non necessariamente gli altri dati sono in grado di leggerli. Elementi in gioco: ripristinatore, caretaker (riprende la stringa dal disco) e memento (oggetto opaco, al suo interno c'è lo stato codificato e non necessariamente gli altri elementi della soluzione sanno interpretarlo).

Questi mementos devono essere facilmente salvabili su disco es. stringhe. Aggiungiamo una responsabilità che ha a che fare con la rappresentazione dello stato quindi che non ci lega ad altri oggetti esterni. Non sempre, tuttavia, si riesce a rispettare il single responsibility principle. Aggiungo quindi 2 funzionalità che sono interamente svolte dentro all'originator.

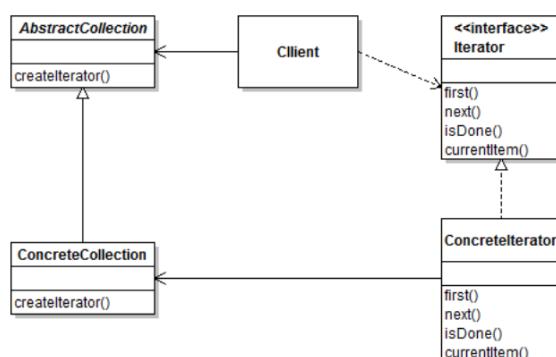
### Pattern Iterator

Pattern Iterator → È usato da tantissimi linguaggi di programmazione, l'idea è offrire un meccanismo per passare in rassegna gli elementi che compongono la collezione senza necessariamente conoscere la struttura e il tipo di collezione con il quale sto dialogando (quindi che sia lista, coda eccetera).

Vado dalla collezione e mi rivolgo ad essa chiedendole di darmi un iteratore. Qualunque sia il tipo di collezione so che avrà il metodo `createIterator()`, quando lo richiamo mi ritorna un oggetto che è istanza di questa interfaccia.

L'oggetto `Iterator` avrà metodi per passare in rassegna gli elementi della collezione, il modo in cui si effettua la rassegna dipende dal tipo di collezione.

Avrò tanti `ConcreteIterator` diversi perché ci sono vari tipi di collezioni, ogni collezione quindi ha un iteratore differente, cambia pure il metodo per passare in rassegna ma tutti presenteranno la stessa interfaccia.



## Pattern Visitor

Avrò una `ObjectStructure`, ogni elemento ha il metodo `Asset()` e noi siamo il visitor. Noi dovremo implementare dei metodi `visit`, dopo ci passiamo all'`ObjectStructure` come metodo `asset` e ci passiamo come parametro.

Pattern Visitor → Pattern che lascia definire una nuova operazione senza cambiare la classe degli elementi sulla quale opera.

L'`ObjectStructure` opera su noi come parametro richiamando i metodi di visita. Noi quindi vogliamo visitare tutti gli elementi di una struttura quindi ci passiamo come parametro ad un metodo `Asset()`, questo richiama il nostro metodo di visita per ogni elemento, noi non controlliamo niente ma specifichiamo un metodo (**Esempio:** se voglio stampare specifico che `visit` deve stampare, poi ci passiamo alla struttura dati che sarà quella che compone la `ObjectStructure`).

Non facciamo noi il ciclo e non abbiamo il controllo di ciò che succede ma specifichiamo il metodo che deve essere richiamato e ci affidiamo all'`ObjectStructure`. → Questo è un inversion of control perché non abbiamo più noi il controllo ma ci affidiamo a qualcun altro che prende il controllo della situazione.

## Pattern Command

Pattern Command → Pattern che permette di estendere dinamicamente il comportamento di alcuni elementi.

L'idea è incapsulare una richiesta sotto forma di oggetto che poi può essere interrogato dal client dall'altra parte. In questo modo possiamo parametrizzare queste richieste inizializzandole in modo opportuno. Quindi invece di passare parametri/oggetti passivi ci passiamo le istanze di un oggetto che verrà poi dall'altra parte interrogato. Questa è una generalizzazione per quanto riguarda quello che abbiamo visto con visitor. Questo pattern fa sì che, attraverso un parametro che passo, riesco a passare anche dei comportamenti oltre ai dati. Chi lo riceve attiva dei comportamenti che decidiamo noi (perché decidiamo come sono realizzati i metodi dell'oggetto che passiamo).

## Pattern Abstract Factory

Pattern Abstract Factory → Fornisce un'interfaccia per creare famiglie di oggetti correlati o dipendenti senza specificare la loro concreta classe. L'idea è non solo rivolgersi alla factory per ottenere un'istanza che ci serve, ma anche che l'istanza possa essere inclusa in una gerarchia di classi che corrisponde ad una gerarchia di costruttori diversi.

## Pattern Builder

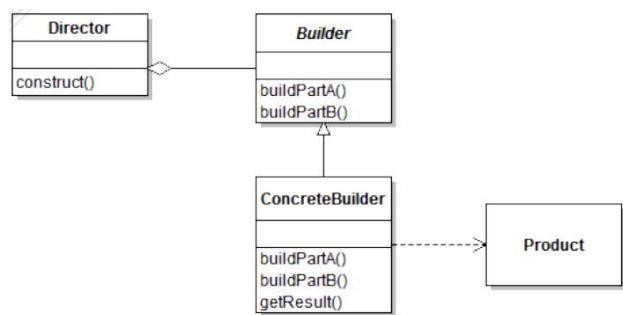
Pattern Builder → Pattern che si occupa dell'istanziazione, è un pattern creazionale e ha a che fare con quegli oggetti che hanno una istanziazione complessa.

**Esempio:** Quando li creo creano anche altri oggetti.

La soluzione è definire un `Director` che si occupa di coordinare la creazione di tutti questi elementi, che usando uno o più `ConcreteBuilder` (specializzazioni del generico `Builder`) ricrea la creazione delle diverse parti.

Questo ci permette di fare in modo che un'eventuale modifica nella struttura degli elementi che compongono l'elemento complessivo possa essere tradotta nella realizzazione di un `ConcreteBuilder` diverso senza cambiare tutto il codice, si rispetta così l'open-closed principle.

→ Separo, quindi, la costruzione di un oggetto complesso dalla sua rappresentazione così che lo stesso processo di costruzione possa creare rappresentazioni differenti.



## **Pattern Prototype**

Pattern Prototype → È un pattern basato sull'idea di clonazione. Specifica, quindi, il tipo di oggetto da creare usando un'istanza prototipo, e crea nuovi oggetti copiando questo prototipo. Si creano nuove istanze clonando le esistenti.

Se mi servono tanti oggetti inizializzati in una certa maniera, questo può non essere elementare. Chiedo di creare un'altra copia dello stato in cui si trova, in questo modo se mi serve inizializzare un gruppo di oggetti in maniera complicata io ne prendo uno e lo porto nello stato in cui mi serve.

## **Pattern Flyweight**

Pattern Flyweight → Permette di istanziare un grande numero di oggetti in modo efficiente.

È una soluzione ragionevole se c'è un problema di memoria che affronteremo solo se abbiamo tanti oggetti da creare o se abbiamo pochissima memoria. Invece di ripetere elementi comuni io esternalizzo la parte comune dello stato e la faccio condividere. Lo stato intrinseco è quello che non posso tirare fuori e quello estrinseco lo posso tirare fuori e metterlo in comune.

## **Pattern Chain of Responsibilities**

Pattern Chain of Responsibilities → È un pattern che serve quando devo gestire un oggetto in qualche maniera, quindi ho un client fuori che mi passa come parametro un oggetto che mi dice di gestirlo, di farci qualcosa, come per esempio stamparlo. Il principio è che non è detto che questo oggetto lo sappia fare, quindi lo passo ad un altro e se non lo sa gestire lo passa a quello dopo, si crea quindi una catena di oggetti correlati che si delegano l'uno con l'altro finché non si arriva ad un oggetto che compie l'operazione.

## **Pattern Interpreter**

Pattern Interpreter → Serve da interprete di un linguaggio; vogliamo permettere di modificare dinamicamente il comportamento di un programma attraverso le istruzioni fornite di volta in volta, esse sono fornite seguendo un linguaggio.

Il programma legge le istruzioni (ne fa il parsing) e in funzione di quello che ci trova attiverà comportamenti diversi, questo è un interprete vero e proprio.

## **18. MODERN PATTERNS**

Esistono altri cataloghi di pattern generali che non hanno avuto particolare successo o comunque non hanno tante occasioni di essere utilizzati.

### **Pattern Null Object**

C'è il problema dell'inizializzazione delle references (inizializzate a null). Per risolvere il problema, se vogliamo dire che questa references non è ancora stata inizializzata invece di metterla a null, la facciamo puntare ad un'istanza specifica che serve a rappresentare l'oggetto non inizializzato.

Se voglio sapere se l'oggetto è inizializzato non guardo se è = null ma lo interrogo e chiedo se è lui il NullObject, se risponde di sì non è stato ancora inizializzato.

→ È da utilizzare in tutte le parti del codice, ci può essere il problema che se uso librerie da terzi possono non utilizzare questo metodo. È quindi di difficile applicazione perché non è sempre applicabile a tutto quanto il codice. Se è imposto dal linguaggio questo problema non c'è.

### **Pattern Type Object**

Ho un oggetto principale che delega ad altri oggetti secondari l'esecuzione di certi comportamenti e di recuperare certi dati che sono esterni, ed è come se estendessero dinamicamente l'oggetto principale.

→ Creo tanti oggetti esterni per la gestione ed un'unica classe base.

### **Pattern Extension Object**

Ho una famiglia di oggetti che sono o estensioni di una classe base comune o di un'interfaccia comune, posso andare dall'oggetto e chiedergli quali sono le sue interfacce. → Questi oggetti hanno interfacce diverse e a run-time posso interrogarli per sapere se supportano o meno quell'interfaccia.

### **The Hollywood Principle**

Questo pattern richiama l'ordine degli eventi che avviene quando si fa un'audizione, ed è legato all'inversion of control menzionato prima.

→ Nel controllo normale è il chiamante che ha in mano il comando però in certi casi può far comodo usare l'inversion of control. Abbiamo un framework quando le classi proposte usano questo meccanismo.

Un framework permette di specificare un comportamento e sarà poi lui ad attivare il comportamento quando lo ritiene opportuno, perciò, noi realizziamo le classi del programma definendo certi comportamenti e poi ci sarà un momento in cui dirò al framework di richiamare lui le classi che ho progettato quando lui ritiene opportuno farlo, ci mettiamo nelle mani di qualcun altro.

### **Dependency Injection**

Dependency Injection → Meccanismo per migliorare la qualità dei sistemi che progettiamo.

Ogni volta che c'è una dipendenza rischiamo di modificare una parte ma di dover modificare anche l'altra, quindi il costo della modifica sale. → Il problema è l'istanziazione.

Dependency Injection serve per definire dei meccanismi di istanziazione degli oggetti che mi permettono di non usare new e di non usare le factory. → New è sbagliato sempre, le factory possono diventare critiche quindi si usano bene solo nei casi più semplici (funzionalità elementari, un'unica classe concreta).

Questo metodo riconduce i problemi a 2 casi elementari:

1. Quello in cui l'oggetto lo conosco già
2. Quello in cui non conosco l'oggetto e lo devo creare al momento.

Se devo operare su un oggetto che creo al momento è difficile che sia un oggetto di cui non voglio tener traccia, quindi è frequente che io debba dialogare con un oggetto di cui ho già il riferimento piuttosto che doverlo creare al volo (non succede quasi mai).

Dependency Injection si occupa di inizializzare opportunamente le variabili di istanza in modo che referenzino degli oggetti che mi servono.

Si chiama così perché usa inversion of control per inizializzare in maniera opportuna le variabili di istanza:

- Scrivo la classe Persona che ha una reference ad un'animale di compagnia di tipo animale che può essere cane, gatto ecc
- All'interno di Persona scrivo una funzionalità ad esempio nutri l'animale.
- Utilizzerò la references animale da compagnia per dare da mangiare all'animale.
- *La reference "animale da compagnia" da chi è stata associata all'animale specifico (cane, gatto ecc.)?* Risolvo questo problema con l'inversion control: dico a qualcun altro di inizializzare il mio animale da compagnia sulla base delle mie istruzioni. → Ho quindi un metodo setAnimaleDaCompagnia(). → Questo lavoro di referenziare lo fa il framework del Dependence Injection.

### **DI in Java**

Esiste un framework chiamato Spring basato sul Dependence Injection: l'idea è che non voglio avere a che fare con le new e con le factories, specifico qual è la mia reference all'animale da compagnia, quando dovrò operarci sopra sarà già inizializzata opportunamente da qualcun altro.

## **19. AGILE SOFTWARE DEVELOPMENT**

I metodi per lo sviluppo Agile di solito applicano lo sviluppo iterativo ed evolutivo timeboxed, fanno uso della pianificazione adattiva, promuovono le consegne incrementali e comprendono altri valori e pratiche che incoraggiano agilità, ovvero una risposta rapida e flessibile ai cambiamenti.

Non è possibile dare una definizione precisa di metodo Agile poiché le pratiche adottate variano da metodo a metodo.

Tuttavia, una pratica base condivisa dai vari metodi è quella che prevede iterazioni brevi e timeboxed, con un raffinamento evolutivo dei piani, dei requisiti e del progetto. → Promuovono pratiche e principi che riflettono una sensibilità agile per la semplicità, la leggerezza, la comunicazione, i gruppi di lavoro che si organizzano in modo autonomo e altro.

→ Qualsiasi metodo iterativo, compreso UP, può essere applicato in uno spirito Agile. UP stesso è flessibile, e incoraggia un atteggiamento che favorisce l'inclusione di pratiche da Scrum, XP e da altri metodi.

### **Problemi**

Problema di efficienza: viene impiegato troppo sforzo nell'overhead (tutte quelle attività non direttamente collegate alla costruzione del sistema software).

→ Migliorare il processo per poter fornire il prodotto non funziona bene come nelle altre branche dell'ingegneria.

Viene, inoltre, spesa molta energia per garantire l'aderenza al piano: contratto di negoziazione, documentazione comprensiva (collegata al processo), assistenza ai rischi ecc. Questi sforzi possono essere sprecati se il piano non è del tutto chiaro dall'inizio, ed è spesso il caso degli sviluppi software.

→ Non abbiamo bisogno solo di un software "flessibile" ma anche di un nostro intero sistema di sviluppo che si adatti al cambiamento.

### **Manifesto per gli sviluppi software Agile**

Si stanno scoprendo modi migliori per lo sviluppo software attuandoli ed aiutando gli altri ad effettuarli.

Attraverso questo lavoro si è arrivati a valutare:

- Individui ed interazioni su processi e strumenti;
- Software di lavoro su documentazione completa;
- Collaborazione con il cliente per la negoziazione del contratto;
- Rispondere ad un cambiamento dopo l'esecuzione di un piano.

### **Agile software development**

Lo sviluppo software Agile non è:

- Un processo
- Un metodo di progettazione

Lo sviluppo software Agile, dunque, è una raccolta di pratiche guidate da un insieme di principi.

### **I principi Agile**

1. La nostra maggiore priorità è di soddisfare il cliente attraverso consegne anticipate e continue di software di valore fin dall'inizio.
2. Sono benvenuti i cambiamenti nei requisiti, anche durante fasi avanzate dello sviluppo. I processi Agile favoriscono il cambiamento per il vantaggio competitivo del cliente.
3. Distribuire frequentemente software funzionante, in intervalli che vanno da un paio di settimane fino ad un paio di mesi. Sono preferiti i periodi di tempo più corti.
4. I responsabili dell'organizzazione e sviluppatori devono lavorare insieme, giornalmente, per tutta la durata del progetto.
5. Costruzione del progetto attorno a degli individui motivati. Fornitura dell'ambiente e del supporto di cui hanno bisogno, e confidare nel fatto che faranno il loro lavoro.
6. Il più efficiente ed efficace metodo per trasmettere informazioni tra un team è attraverso la conversazione faccia a faccia.

7. Un software funzionante è la misura principale del progresso.
8. I processi Agile promuovono uno sviluppo sostenibile.
9. Gli sponsor, gli utenti e gli sviluppatori dovrebbero essere in grado di mantenere un'andatura costante all'infinito.
10. Una continua attenzione all'eccellenza tecnica e ad una buona progettazione migliora l'agilità.
11. La semplicità (l'arte di massimizzare l'ammontare di lavoro non fatto) è essenziale.
12. Le migliori architetture, requisiti e progetti emergono da team di lavoro auto-organizzativi.
13. Ad intervalli regolari, il team deve riflettere su come diventare più efficace, quindi si accorda e sistema il suo comportamento di conseguenza.

## **Metodi Agile**

Negli scorsi anni sono stati proposti una serie crescente di metodi ispirati all'Agile:

- Agile Modeling
- Agile Unified Process
- Crystal Clear
- Extreme Programming
- Scrum e altri

## **Le pratiche**

- Refactoring
- Piccoli cicli inversi
- Integrazioni continue
- Standard coding
- Ownership collettive
- Planning game
- Whole team
- Meeting giornalieri
- Design test-driven → Stile di programmazione dove 3 attività sono strettamente intrecciate: Coding, Testing e Design.  
Può essere descritto dalle seguenti 3 set di regole:
  - Scrivere una singola unità di test descrivendo un aspetto del programma;
  - Fare il run del test, che dovrebbe fallire perché al programma manca la funzionalità;
  - Scrivere abbastanza codice nel modo più semplice possibile per far sì che il test passi;
  - Fare refactor del codice finché non è conforme al più semplice criterio;
  - Ripetere accumulando unità di testing nel tempo.
- Revisioni di codice e design → Con il code review solitamente intendiamo le pratiche con le quali del codice nuovo/rivisto deve passare una revisione prima di essere consegnato.  
→ Servono per fornire benefici addizionali come il trasferimento della conoscenza, aumento della consapevolezza dei team e creazione di soluzioni alternative al problema.
- Pair programming (extreme code review) → Consiste in 2 programmatore che condividono una singola stazione di lavoro (uno schermo, una tastiera e un mouse).  
→ Il programmatore alla tastiera è solitamente chiamato guidatore, l'altro è coinvolto attivamente nel compito di programmazione ma si focalizza maggiormente sulle direzioni generali, è chiamato navigatore. Ci si aspetta che essi si scambino di ruolo ogni serie di minuti.
- Document late
- Uso dei design patterns

## **User stories**

Sono frasi brevi e concise scritte in linguaggio di dominio, che catturano le aspettative degli utenti. Le user stories NON sono casi d'uso. Uno dei soliti modelli per un user story è:

As a <role>, I want <goal> so that <benefit>

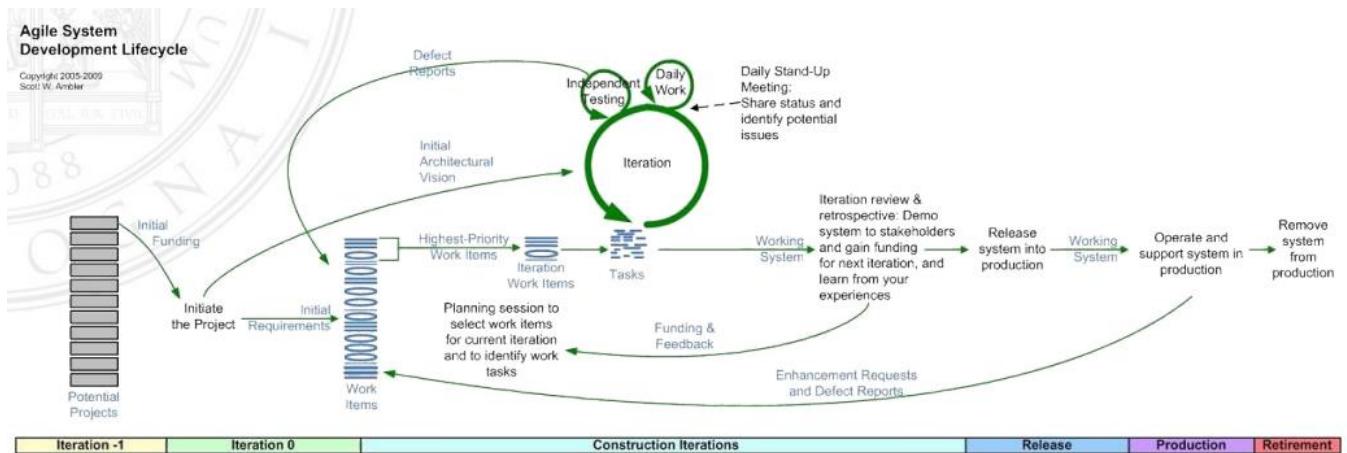
**Esempio:** As a employee I want to purchase a parking pass so that I can drive to work.

## Invest

Un gruppo accettato di criteri, o lista di controlli, per valutare la qualità di una user story:

- Independent
- Negotiable
- Valuable
- Estimable
- Small
- Testable

## Ciclo di vita dell'Agile



## Agile ed evoluzione

Lo sviluppo è solo una parte del ciclo di vita del software, l'evoluzione ha lo stesso livello di importanza. Negli approcci Agile la conoscenza è condivisa implicitamente e vengono prodotte piccole documentazioni: cosa succede se la manutenzione è affidata ad un team diverso? Questo dovrebbe essere documentato più tardi anche se molte volte questa cosa non viene fatta.

## Extreme programming (XP)

- 4 attività: Coding, Testing, Listening e Designing.
- 5 valori: Comunicazione, Semplicità, Feedback, Coraggio e Rispetto.
- 3 principi: feedback, si assume semplicità e propensione al cambiamento.

## Pratiche XP

Le pratiche nell'extreme programming sono divise in 4 gruppi:

- Feedback fine scale → Comprende il pair programming, il planning game, lo sviluppo test driven e coinvolge tutto il team.
- Processi continui → Consiste in integrazioni continue, miglioramenti nel design (progettazione) ed è caratterizzato da piccole release.
- Comprensione condivisa → Il coding è standard, c'è una proprietà collettiva del codice, la progettazione è semplice e si usano le system metaphor.
- Benessere del programmatore → Ritmo sostenibile.

Planning game → È il processo di pianificazione in XP ed è basato sulle user stories. → È tenuto prima di ogni iterazione.

Ci sono 2 parti:

- Pianificazione della release (include i clienti)
- Pianificazione delle iterazioni (coinvolge solo gli sviluppatori).

I clienti ordinano le stories in base al valore (critico, significativo, buono), mentre i programmatori in base al rischio. → Le user stories che saranno finite nella successiva release sono quelle scelte.

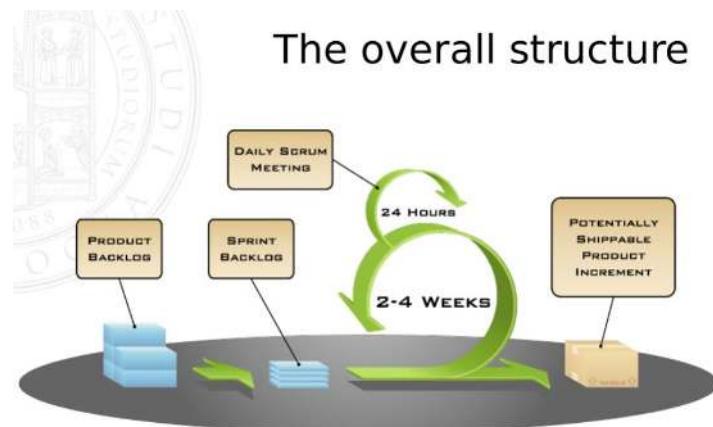
## 20. Scrum

### **Che cos'è Scrum**

Scrum è un metodo agile che consente di sviluppare e rilasciare prodotti software con il più alto valore per i clienti, nel breve tempo possibile. Si occupa principalmente dell'organizzazione del lavoro e della gestione dei progetti. È invece meno interessato agli aspetti tecnici dello sviluppo software.

Si tratta di un approccio iterativo e incrementale allo sviluppo del software.

La caratteristica distintiva di Scrum, tra i metodi agili, è l'enfasi sull'adozione di Team auto-organizzati e auto-gestiti. Inoltre, Scrum è basato su un insieme di elaborati ed eventi che hanno lo scopo di rendere visibili gli obiettivi e il progresso delle iterazioni e di favorire un adattamento evolutivo del processo di sviluppo.



### **La teoria di Scrum**

Scrum è basato su un modello di controllo del processo di sviluppo di tipo empirico, sorretto da tre pilastri:

- Trasparenza → Scrum è trasparente nel senso che gli aspetti significativi del processo di sviluppo devono essere visibili a tutte le parti interessate.
- Ispezione → Il Team Scrum deve ispezionare frequentemente il lavoro fatto (con i relativi elaborati) per verificare che si stia procedendo verso gli obiettivi posti.
- Adattamento → Nel caso in cui il processo di sviluppo stia deviando dai suoi obiettivi, allora è necessario un adattamento per minimizzare ulteriori deviazioni.

### **Gli elaborati di Scrum**

In Scrum, lo sviluppo iterativo viene gestito con riferimento a due elaborati principali: il Product Backlog e lo Sprint Backlog.

#### Product Backlog

Il Product Backlog è un elenco completo di tutto ciò che è necessario nel prodotto da realizzare. È l'elaborato che riporta tutti i requisiti del prodotto. Si tratta di un elenco ordinato sulla base delle priorità tra le voci. Le priorità sono di solito assegnate sulla base del valore di business delle voci oppure sul rischio.

Il Product Backlog è un elaborato dinamico, che viene gestito in modo iterativo. All'inizio del progetto, contiene delle caratteristiche desiderate per il prodotto e dei problemi che devono essere risolti.

Per trasparenza, il Product Backlog è visibile all'intero Team Scrum, ma anche a tutte le parti interessate alla realizzazione del prodotto, come i committenti e rappresentanti degli utenti.

#### Sprint Backlog

Lo Sprint Backlog è l'insieme delle attività (task) necessarie per le voci del Product Backlog selezionate per lo sviluppo di uno Sprint. Questo elaborato definisce lo Spirit Goal per il Team, e contiene anche un piano per la realizzazione dell'incremento del prodotto.

Per trasparenza, lo Sprint Backlog è visibile all'interno del Team di sviluppo.

## **I ruoli di Scrum**

Un progetto Scrum viene svolto da un gruppo di lavoro, chiamato complessivamente Team Scrum, in cui sono previsti tre ruoli:

- Product owner → È responsabile della definizione delle caratteristiche del prodotto da sviluppare, e dunque della massimizzazione del suo valore. Questo ruolo è svolto da una singola persona, e non da un comitato.

Durante lo sviluppo il Product Owner ha la responsabilità di gestire il Product Backlog: di scegliere le voci iniziali, ma anche di aggiungere, eliminare o raffinare le voci; è anche responsabile di assegnare priorità alle voci. Deve anche assicurarsi che il Team di sviluppo comprenda le voci del Product Backlog e la loro importanza.

- Development team → Chiamato anche solo Team, è formato da professionisti che hanno la responsabilità di svolgere il lavoro necessario per sviluppare e rilasciare un prodotto in incrementi successivi. Sono solo i membri del Team a realizzare questi incrementi. Il Team è *cross-funzionale*, ovvero i suoi membri possiedono tutte le competenze necessarie per sviluppare gli incrementi del software. Il Team è anche *auto-organizzato* e *auto-gestito*.

Un Team deve essere grande abbastanza per poter completare in ciascuno Sprint un incremento di prodotto significativo, ma piccolo abbastanza da poter lavorare in modo agile. Per questo il Team è composto in genere da 5 a 9 persone.

Scrum può essere utilizzato anche da Team di sviluppo più grandi, composti anche da diverse centinaia di sviluppatori. In questo caso gli sviluppatori sono suddivisi in più Team e sono previste delle attività di controllo fra i diversi Team.

- Scrum master → È responsabile di assicurare che Scrum venga compreso e applicato correttamente dal Team di sviluppo, dal Product Owner e dalle altre parti interessate al prodotto. Si tratta di un ruolo molto importante perché, anche se Scrum è semplice da capire, non è altrettanto semplice da padroneggiare. Lo Scrum Master sostiene il Team, affinché aderisca alle pratiche e alle regole di questo metodo. Si tratta di un istruttore e di una guida, non è il manager.

## **Gli eventi di Scrum**

Scrum prevede un certo numero di eventi in coordinamento, da svolgere regolarmente. L'evento principale è lo Sprint, che può essere considerato come un contenitore di tutti gli altri eventi. Tutti gli eventi sono timeboxed, ovvero hanno una durata fissata. Gli eventi contenuti nello Sprint sostengono trasparenza e forniscono l'opportunità di ispezionare e adattare.

- Sprint → Il cuore di Scrum è lo Sprint, un'interazione di sviluppo che ha lo scopo di realizzare un incremento di prodotto potenzialmente rilasciabile.

Gli Sprint di un progetto hanno in genere tutta la stessa durata, e vengono svolti uno dopo l'altro, senza interruzioni. La durata degli Sprint deve consentire la realizzazione di incrementi in prodotti significativi, ma anche la possibilità di avere feedback e adattamenti tempestivi.

→ La maggior parte del tempo di uno Sprint è dedicata al lavoro di sviluppo.

- Sprint planning → È una riunione di pianificazione che viene svolta all'inizio di ciascun Sprint. La durata massima è di otto ore per uno Sprint di quattro settimane. L'obiettivo dello Sprint Planning è selezionare le voci del Product Backlog da sviluppare.

La prima parte dello Sprint Planning ha lo scopo di definire l'obiettivo di sviluppo dello Sprint. Il Product Owner illustra al Team di sviluppo le voci che, se realizzate, potrebbero massimizzare il valore dell'incremento di prodotto.

Nella seconda parte dello Sprint Planning, il Team di sviluppo si concentra su "come" realizzare lo Sprint Goal. Il Team stima lo sforzo necessario per sviluppare le voci proposte dal Product Owner.

Alla fine, il Team di sviluppo dovrebbe essere in grado di spiegare al Product Owner e allo Scrum Master come esso intende lavorare come squadra *auto-organizzata* per realizzare l'obiettivo dello Sprint e creare l'incremento previsto.

- Daily scrum → È una riunione giornaliera, della durata massima di 15 minuti, che si tiene sempre alla stessa ora (solitamente la mattina), e nello stesso posto. I membri del Team di sviluppo ispezionano il lavoro fatto nelle ultime 24 ore e definiscono un piano di lavoro per le prossime 24 ore.

In particolare, per ciascun membro del Team a turno si risponde brevemente a queste tre domande:

- 1) *Che cosa ho fatto dall'ultimo Daily Scrum (per aiutare il Team a raggiungere l'obiettivo dello Sprint)?*
- 2) *Che cosa farò fino al prossimo Daily Sprint (per aiutare il Team a raggiungere l'obiettivo dello Sprint)?*
- 3) *Che ostacoli ci sono nello svolgimento del mio lavoro (che impediscono al Team di raggiungere l'obiettivo dello Sprint)?*

Nel Daily Scrum oltre alle tre domande non è prevista nessuna ulteriore discussione. Se invece, è necessaria una discussione, questa può avvenire dopo la fine del Daily Scrum.

- Sprint review → Si tiene alla fine di ciascun Sprint, per ispezionare l'incremento di prodotto realizzato per adattare il Product Backlog. La durata è di 4 ore per uno Sprint di 4 settimane.

Il risultato della Sprint Review è un Product Backlog rivisto, che definisce quelle informazioni utili per pianificare il prossimo Sprint. Tuttavia, questa pianificazione verrà effettuata in un momento separato, ovvero nello Sprint Planning all'inizio dello Sprint successivo.

- Sprint retrospective → È un'opportunità per l'intero Team Scrum di ispezionarsi e di adattarsi rispetto al modo in cui si sta applicando Scrum. In generale, infatti i metodi iterativi e agili si imparano ad applicare in modo iterativo e agile. Si tratta di una riunione di massimo 3 ore, tra la Sprint Review e il prossimo Sprint Planning. Lo scopo è ispezionare com'è andato lo Sprint appena concluso in termini di persone, relazioni, processo e strumenti.

## **Altre pratiche con Scrum**

Scrum è un metodo agile, basato su sviluppo iterativo, trasparenza, ispezione e adattamento. Il lavoro viene gestito dal Team di sviluppo in modo *auto-organizzato* e *auto-gestito*.

### Linee guida: definizione di "fatto"

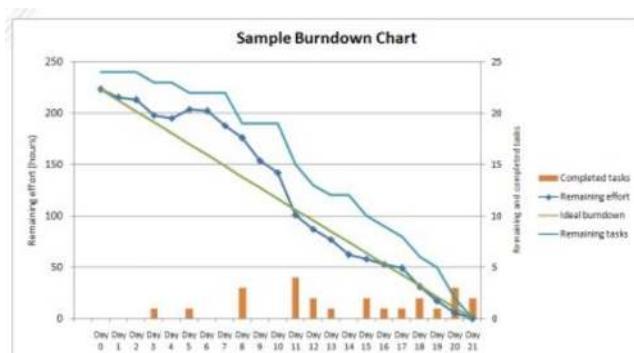
Nello sviluppo iterativo, quando una voce del backlog viene completata, essa viene segnalata come “Fatta”. Per trasparenza, è importante che ognuno capisca che cosa si intende per “Fatto” (*Done*). La definizione di “fatto” è “potenzialmente rilasciabile”: per la voce è stata effettuata analisi, progettazione, implementazione, verifica, integrazione con il resto del sistema e inoltre è stata predisposta anche la documentazione per gli utenti finali.

### Linee guida: tracciare il progresso

Il Team di sviluppo, per potersi *auto-organizzare*, deve sapere come sta progredendo lo sviluppo. Per questo, ogni giorno i membri del Team aggiornano le loro stime dello sforzo rimanente per completare (“fare”) la lavorazione di ciascuna voce dello Sprint Backlog.

Queste stime vengono poi riportate nello Sprint Burndown Chart (grafico sull'andamento del lavoro dello Sprint), che consente a tutto il Team di tracciare e ispezionare il progresso e lo sviluppo.

L'analisi del grafico dell'andamento del lavoro può suggerire una variazione del piano di lavoro dello Sprint.



## Scaling

- Scaling can be achieved via hierarchical scrum teams
- Specific events are scheduled to ensure overall progress (e.g. The Scrum of scrums after the daily meeting)
- LeSS (large scale Scrum) is a framework intended for many teams working together on one product. It is proposed by C. Larman and B. Vodde.

## Kanban

- Kanban is a lean scheduling method to control a production chain for just-in-time production
- Kanban is Japanese work meaning billboard
- Introduced in the software development domain by David Anderson. Open Kanban is an open source version of Kanban for agile software development

## Kanban principles

- Start with existing process
  - Kanban is a change management method that stimulates continuous, incremental and evolutionary changes
- Agree to pursue incremental, evolutionary change
- Respect the current process, roles, responsibilities and titles
- Leadership at all levels

## The pipeline

- Kanban uses a continuous delivery approach (no cycles!).
- All work is split in units and flows through a pipeline composed of stages.

## Visualization

- Visualization is a central concept in Kanban
- A Kanban board is a central artifact used to visualize the status of the work items and their progress (the pipeline)

## Limit WIP

- Kanban tries to match the WIP with team's capacity
- To accomplish this the stages of the pipeline are bounded to a limit
- The process progresses by moving work items through the pipeline, from "to do" to "done", respecting the limits
- Cycle time is the basic metric used to evaluate the team progress

## Board, progress, limits

Workflow	Index	Specification	Ready for Development	Development (e.g. unit, review and test)	Code Review	Deployed System	Pre-production Review
WIP Limit	0	In Progress	Done	In Progress	Done	In Progress	Done
Feature	User Story #001	User Story #002	User Story #003	Planned	In Progress	Done	In Progress
Login	User Story #001	User Story #002	User Story #003	User Story #004	User Story #005	User Story #006	User Story #007
Register				User Story #004	User Story #005		
Password Recovery	User Story #004				User Story #005		User Story #006
Logout							
Billing			User Story #007	User Story #008		User Story #009	

## **21. SOFTWARE TESTING**

Software testing è parte delle attività di verifica e validazione.

Verifica → Valutazione degli artefatti (incontri, recensioni). Risponde alla domanda: *Stiamo costruendo il giusto sistema?*

Validazione → Testing. Risponde alla domanda: *Il sistema che stiamo costruendo è corretto?*

Software testing consiste nel rivelare tanti difetti quanto è ragionevole.

Generalmente il testing, nei software, non può fare uso di risultati raccolti da altre discipline ingegneristiche.

**Esempio:** Un ponte testato per 100 tonnellate può sicuramente sopportare un peso inferiore, ma i programmi non beneficiano di questo tipo di proprietà continue.

Il costo per correggere i difetti in un sistema software è proporzionale al tempo per cui i difetti rimangono nel sistema. → Come conseguenza si devono fare test presto e spesso.

### **Jargon**

Defect (bug) → Il risultato di un errore.

Failure → Avviene quando un errore è esposto.

Issue → Descrive il fallimento.

Test case → Descrive il risultato che ci si aspetta da un run. Include la data, le pre/post condizioni e i risultati che ci si aspetta.

Test set → Collezione di test cases.

### **Testing levels**

Livelli di test: unità, integrazione, sistema, acceptance, classe/metodo, gruppo di moduli software, tutto il sistema (in vitro), il sistema in uso.

#### **Test statici e test dinamici**

Static test → Il codice è analizzato per trovare bugs.

Dynamic test → Il codice è eseguito per trovare bugs.

La maggior parte dei test statici sono basati su metodi di approccio formale: model checking, data-flow analysis, abstract interpretation e symbolic execution.

Noteworthy tool: Polyspace (contrassegna il codice in base ad analisi statiche: codice affidabile, codice difettoso, codice irraggiungibile, codice non provato, codice che viola le regole).

Anche recensioni umane sono un tipo di test statico.

#### **Black-box testing**

Nei black-box testing le funzionalità sono esaminate senza alcuna conoscenza dell'implementazione interna (cosa non come).

I metodi di testing includono:

- Equivalence partitioning
- Boundary value analysis
- All-pairs testing
- State transition table
- Decision table testing

#### **White-box testing**

La struttura interna del codice è usata per definire i test cases.

I metodi di testing includono:

- Control flow/data flow
- Branch testing (decision coverage)
- Path testing (code coverage)

## Pro e contro di Black e White testing

White:

- Pro: La conoscenza del codice è acquisita mentre si costruiscono i test cases, copertura più alta.
- Cons: Complessità.

Black:

- Pro: Testers che non sono programmatori informatici, più vicino ai requisiti.
- Cons: Copertura sconosciuta.

## Testare i tests

Come si stima la qualità di un test set? Attraverso i mutation testing: si crea una mutazione del tuo codice, si eseguono i test sui mutanti e se il test viene passato si ha un problema.

PIT → Mutation test utilizzabile per Java.

## Unit testing

Nello unit testing singole unità di codice (funzioni, metodi) vengono testati.

→ Lo unit testing è utilizzato per assicurare che il codice rispetti le aspettative e che continui a rispettarle (regression testing).

Le unità devono essere testate in isolamento e il test set per ogni unità deve contenere casi indipendenti.

## Isolation

Come si scioglie il codice dalle sue dipendenze? Si utilizzano stub metodi e fake/mock oggetti. Essi forniscono la stessa interfaccia con un codice semplificato.

## XUnit

Un framework per unit testing progettato originariamente da K.Beck. → L'implementazione più utilizzata è JUnit.

Le componenti dell'architettura di XUnit sono: test runners, test case (usa asserzione, una per caso), test fixtures (or contexts), test suites e test execution.

## 22. (D)VCS

Strumenti di gestione delle revisioni, appositamente pensati per gli artefatti dei progetti software e modificati durante il processo di sviluppo. Sono a supporto del team.

### Sviluppo storico

Local VCS:

- 1972 – SCCS
- 1982 – RCS

Client-server VCS:

- 1990 – CVS (client-server)
- 2000 – SVN

Distributed VCS:

- 1990s – TeamWare/1998 BitKeeper
- 2003 – Monotone/Darcs/006 – Arch
- 2005 – Bazaar/Mercurial/Git

### Nozioni base

Working copy → File del progetto memorizzati nel file system utilizzato dal team di sviluppo.

Repository → Versioni di file del progetto.

Pending change set (set di modifiche in sospeso) → I file del working copy che non sono ancora nel repository, oppure file che differiscono dall'ultima versione degli stessi file presenti nel repository.

Commit → Sottopone il pending change set al repository.

Update → Sincronizza il working copy con il repository.

### Operazioni sugli artefatti

Add → Aggiungere un elemento a quelli che sono sottoposti alla revisione.

Edit → Modifica di uno degli artefatti

Delete → Elimina una o più modifiche effettuate sugli artefatti.

Rename → Viene rinominato l'artefatto.

Gli artefatti iniziano ad evolvere, alcuni elementi possono essere tolti e altri modificati: se vengono tolti non esistono più nella storia degli artefatti prodotti, così come se viene cambiato il nome si vede nella storia degli artefatti.

Move → Sposta l'artefatto.

Status → Permette di conoscere lo stato dell'artefatto, dall'ultima sincronizzazione tra il working copy e il repository.

Diff → Operazione che permette di vedere solo le differenze, cioè solo le modifiche fatte all'artefatto originale.

Revert → Torna all'artefatto originale.

Log → Applicato alla working copy, dà informazioni sulla storia dell'artefatto, in base alle sincronizzazioni effettuate tra il working copy e il repository.

Tag → Permette di associare un nome ai vari momenti storici dell'artefatto. Il sistema viene rilasciato alla fine della versione, l'incremento della revisione si ha solo quando si arriva alla fine dell'iterazione. Durante l'iterazione si possono creare tante versioni.

**Esempio:**

    Revisione

    Revisione

FINE ITERAZIONE 1 → Tag "Versione 0.1"

    Revisione

    Revisione

FINE ITERAZIONE 2 → Tag "Versione 0.2"

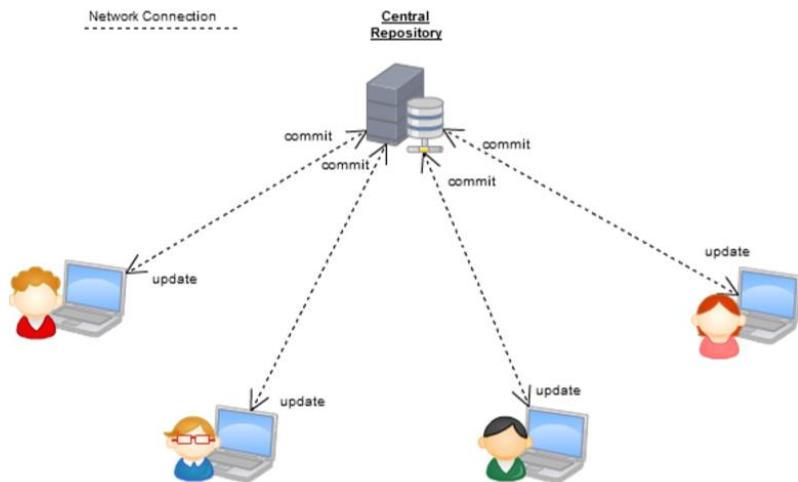
Branch → A partire dalla stessa storia si possono avere versioni distinte dell'artefatto. Queste poi possono continuare ad evolvere come versioni separate, oppure essere unite con un Merge.

Merge → Permette di unire le varie versioni di uno stesso artefatto, segnalando gli eventuali conflitti che scaturiscono.

Resolve → Risolve i conflitti che sorgono dal Merge.

Lock → All'interno del repository ci sono delle modifiche effettuabili solo da un determinato utente, quindi non è possibile effettuare un Commit se non si hanno le autorizzazioni. È un meccanismo quasi obbligatorio nelle versioni Client-Server VCS degli anni 90-2000, per questo poi è diventato facoltativo.

## Controllo Centralizzato delle Versioni



Il repository è centrale e remoto, ogni utente ha il suo working copy perciò lavora sulla propria copia locale, ma non ci sono altre repository.

Il meccanismo d'uso è:

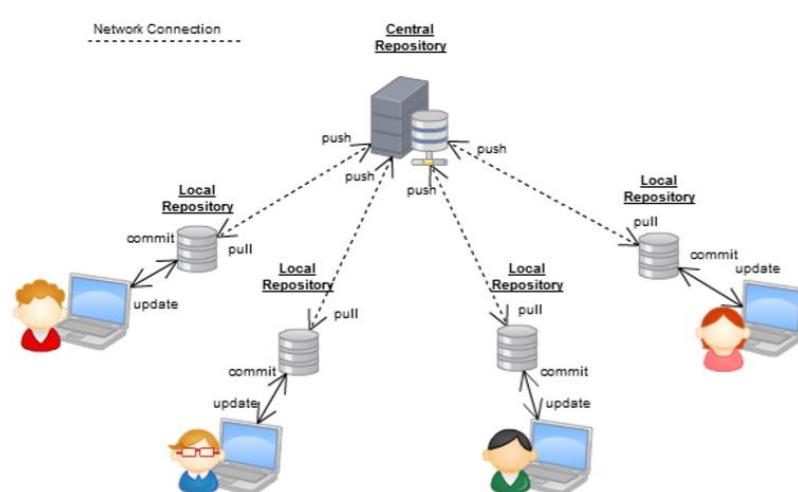
- Si inizia la sessione di lavoro e si fa l'update, così da poter lavorare sull'ultima versione disponibile
- Si effettuano le modifiche
- Si fa il commit

Problema: Due utenti modificano lo stesso file e fanno entrambi il commit del file modificato in modi diversi.

Soluzione: Per evitare che questo succeda, i meccanismi client server fanno mettere il lock dopo che l'*utente1* ha fatto l'update, così se arriva *utente2* successivamente non può modificare. → È quindi fondamentale ricordarsi di togliere il lock alla fine della sessione.

Per questo motivo il merge non può essere automatizzato, ma va sempre fatto: copy – modify – merge.

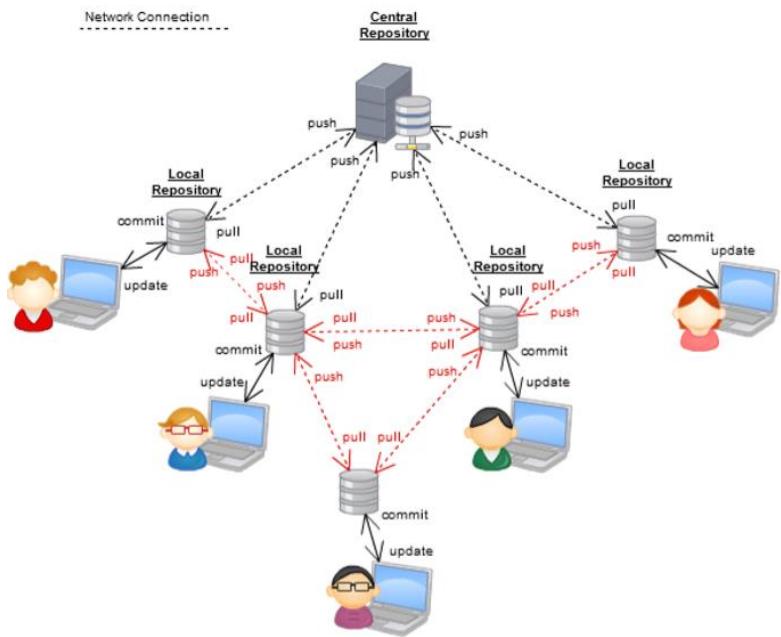
## Controllo Distribuito Simple delle Versioni



Ogni utente ha un repository locale e, attraverso commit e update, le copie di lavoro sono sincronizzate con il repository locale.

I repository locali sono sincronizzati con il repository centrale utilizzando push e pull. → Il repository centrale è sempre la fonte attendibile di verità.

## Controllo Distribuito Fully delle Versioni



→ Peer to peer

Il repository centrale potrebbe esistere o meno e non c'è una fonte di verità.

## Branches

Branches → Ramificazione, cioè duplicazione di un oggetto sotto il controllo del codice sorgente.

I rami formano gerarchie:

- Child branch
- Parent (a monte)
- Trunk (parent less branch)

I rami divergenti possono successivamente essere uniti (cioè integrati con un antenato), mentre una ramificazione che non è destinata a essere successivamente unita viene chiamata fork.

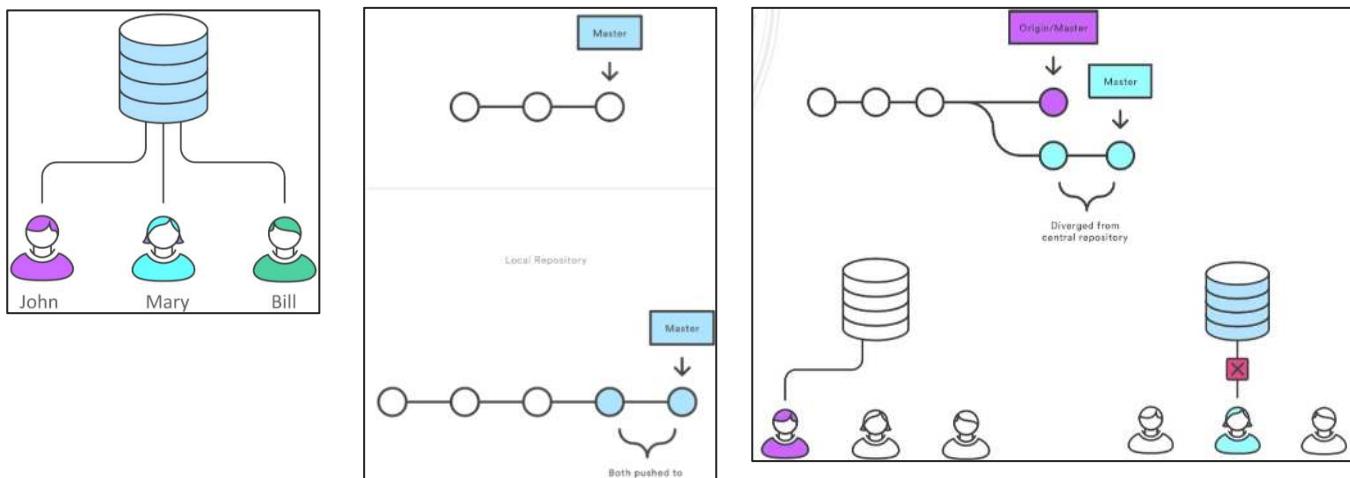
## Git

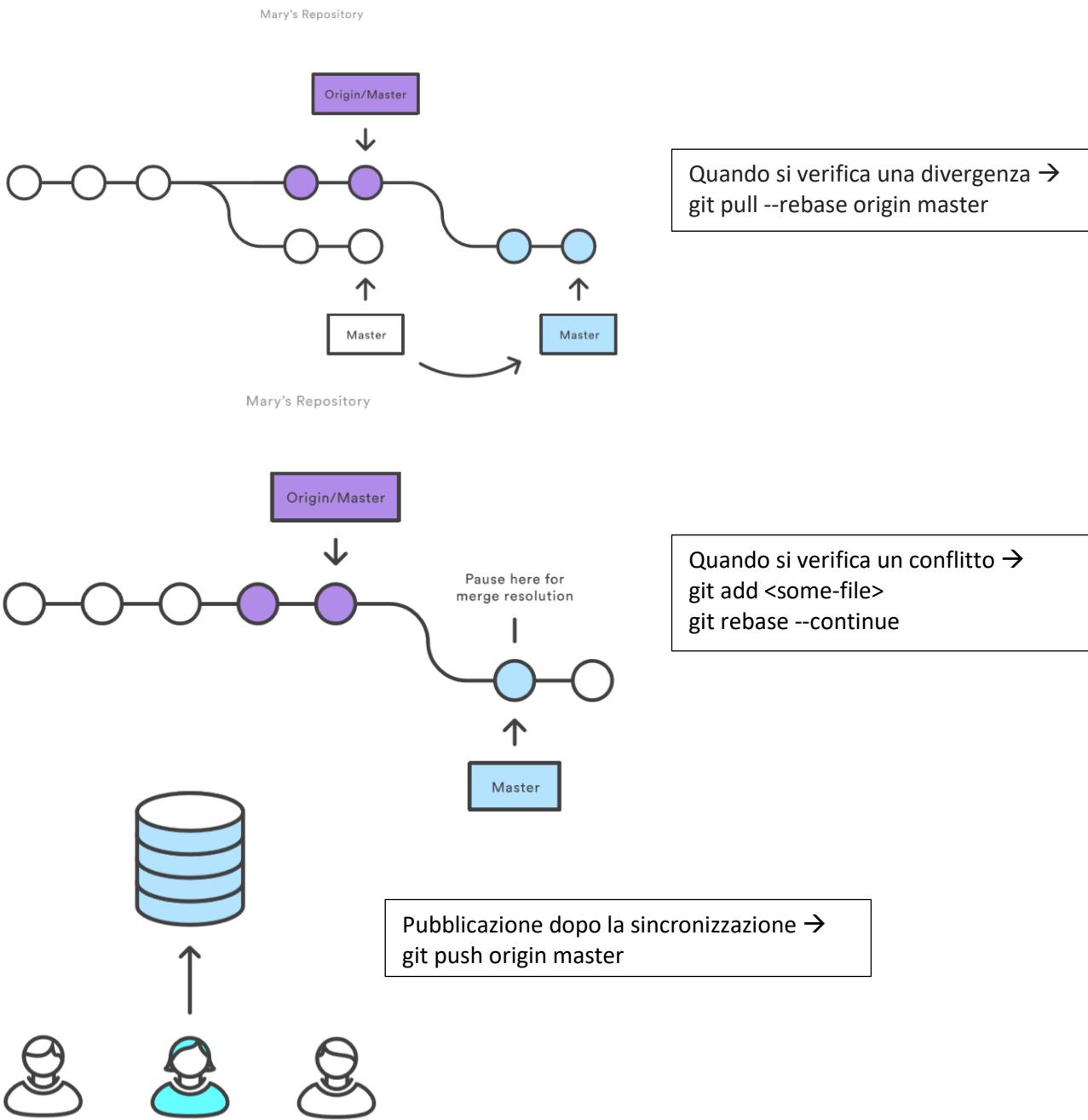
Un VCS progettato per essere:

- Affidabile
- Ad alte prestazioni
- Distribuito

→ Parola chiave: ramificazione.

## Workflows Centralizzati





### Workflows Ramificati

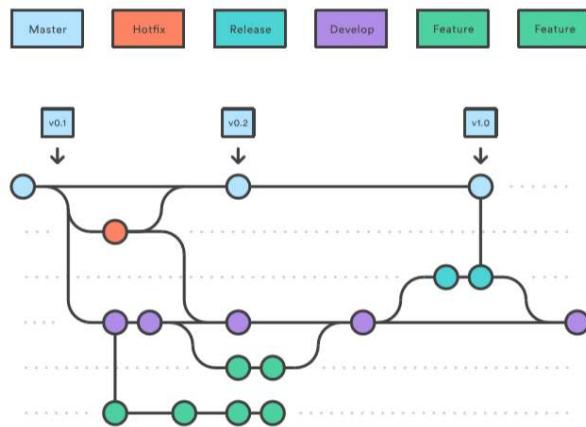
Tutte le funzionalità devono essere sviluppate in un ramo dedicato poi, una volta completata la funzione, il feature branch viene premuto e viene aperta una richiesta pull. Quando la richiesta viene accettata, viene fatto il merge sul feature branch.

### Workflows Gitflows

Ci sono due rami principali: Master e Develop.

Oltre a quelli principali, ci sono poi altri rami che sono derivati dai precedenti due:

- Rami Features → Derivano dal Develop e su questi viene fatto il merge.
- Rami Release → Vengono derivati Develop quando ci si avvicina a un rilascio, da questo ramo non viene impegnato nessuna nuova feature. Quando è pronto per la spedizione, la Release è unita al Master.
- Rami Hotfix → Sono gli unici che derivano dal Master e vengono usati solo per correggere i bug-fixing e vengono subito ricomposti.



### Conclusioni

Un (D)VCS è la colonna portante di un progetto software → Deve essere fatto il commit di tutti gli artefatti.  
È importante imparare ad usarli anche se questo può richiedere del tempo.

## 23. GWT

Se si vuole creare una web application ci sono delle tecnologie minime che bisogna saper padroneggiare per poter fare ciò:

- HTML
- DOM
- ECMA Script (è JavaScript)
- CSS
- http
- HML/JSON
- Linguaggi server-side (PHP, ASP, JSP ...)

GWT (Google Web Toolkit) → È un insieme di strumenti che ha come obiettivo quello di semplificare lo sviluppo di web application. L'idea base è che Java è il migliore linguaggio per fare ciò, per cui è altamente consigliato usarlo.

Il codice client-side è compilato da Java, il contenuto è gestito dagli oggetti (widget) e pannelli delle classi. Il codice server-side è redatto in Java, tuttavia, serve anche CSS.

### Strumenti di sviluppo

GWT è principalmente uno strumento di sviluppo, include strumenti utili per facilitare il lavoro dei programmatori di web application. Comprende un'alta interazione con Eclipse, un potente metodo di sviluppo e un designer UI.

### Client side

Il codice Java viene compilato in JavaScript, tuttavia esistono delle limitazioni:

- Non sono contemplati multithreading e sincronizzazioni;
- Non sono ammessi ripensamenti;
- Non è prevista serializzazione;
- API limitate (parti di `java.lang`, `java.lang.annotation`, `java.math`, `java.io`, `java.sql`, `java.util`, `java.util.logging` sono disponibili).

### Server side

Il codice server side è ospitato in un contenitore Servlet (non è necessaria alcuna conoscenza degli API Servlet). La comunicazione tramite codice con il client side è gestita con un meccanismo integrato di RPC. Molti servizi estendono la classe `RemoteServiceServlet` che gestisce i lavori.

### Interazione client-server

Il meccanismo fornito RPC consente al codice del cliente di invocare dei servizi in stato running sul lato server, creando un proxy dinamico che gestisce i dettagli di basso livello (**Esempio**: Eccezioni). Attenzione: le invocazioni sono asincrone ovvero i servizi devono esporre un'interfaccia asincrona (che deriva da quella "regolare") e i clients fanno riferimento a quell'interfaccia.

Ci sono altri meccanismi che possono essere usati per la comunicazione client-server (JSON, XML...), le librerie sono disponibili ma il supporto non è integrato.

### Host page ed entry point

L'host page è il punto di entrata (entry point) della web application. Il codice client può aggiungere pannelli/oggetti a specifiche aree dell'host page o sostituirne il suo intero contenuto. Una volta che la host page è caricata, il metodo `onModuleLoad` della classe principale (che deve implementare `EntryPoint`) viene invocato.

## Widget, pannelli ed eventi

Gli elementi UI sulla pagina web possono essere dinamicamente creati/modificati usando un insieme di classi UI contenenti pannelli (contenitori) e widget (elementi UI). Gli event handler sono usati per reagire alle varie gestioni degli utenti come in Swing.

Se è necessario è disponibile un'API per manipolare direttamente il DOM dell'host page.

## GWT RPC

GWT RPC è un metodo semplificato per lasciare che il codice client interagisca con il codice server. Il codice server è strutturato in servizi, ovvero classi java che estendono `RemoteServiceServlet` ed implementano un'interfaccia che descrive le operazioni disponibili al client. Deve essere fornita anche una versione asincrona dell'interfaccia.

I client creano dinamicamente un proxy per ogni servizio che vogliono usare (usando `GWT.create()`) ed interagiscono direttamente con essi usando la loro interfaccia asincrona.

## Servizi ed interfacce

```
package com.example.foo.client;

import com.google.gwt.user.client.rpc.RemoteService;

public interface MyService extends RemoteService {
    public String myMethod(String s);
}
```

```
package com.example.foo.server;

import com.google.gwt.user.server.rpc.RemoteServiceServlet;
import com.example.client.MyService;

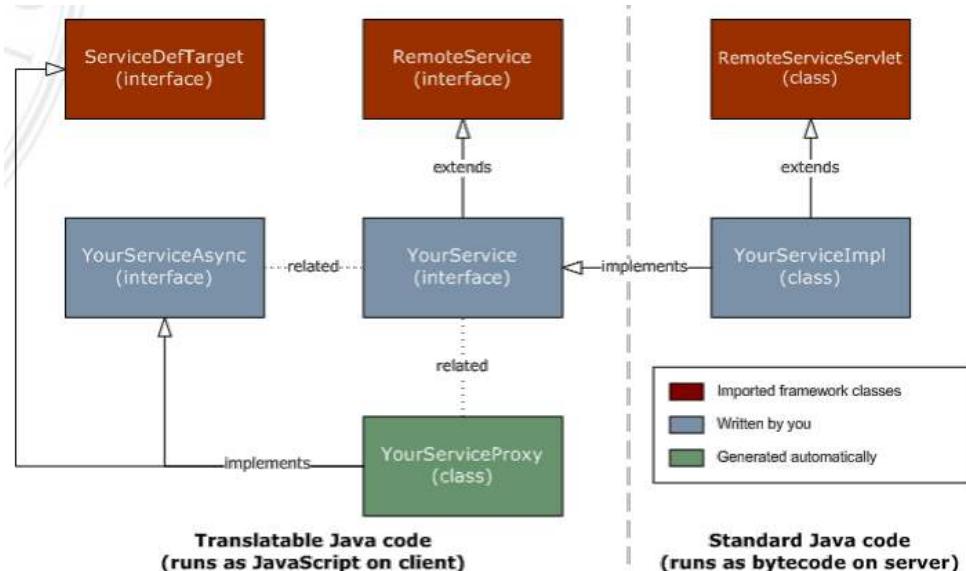
public class MyServiceImpl extends RemoteServiceServlet implements
    MyService {

    public String myMethod(String s) {
        // Do something interesting with 's' here on the server.
        return s;
    }
}
```

```
package com.example.foo.client;

interface MyServiceAsync {
    public void myMethod(String s, AsyncCallback<String> callback);
}
```

## GWT RPC



## **Sessioni**

Come in ogni web application, il codice server side non ha conoscenze su chi sta facendo le invocazioni. Quando si crea un'applicazione multi-pagine GWT (ricordando che le applicazioni GWT possono essere ospitate in una singola pagina) le sessioni possono essere usate per associare informazioni alla richiesta (**Esempio:** L'identificativo dell'user che sta facendo la chiamata).

`getThreadLocalRequest().getSession().setAttribute(name, value)` e  
`getThreadLocalRequest().getSession().getAttribute(name)` possono essere usati per accedere ai dati della sessione.